

ASL Typing Reference

DDI 0622

Arm Architecture Technology Group

September 25, 2024

Contents

1	Non-Confidential Proprietary Notice	11
2	Disclaimer	13
3	Introduction	15
4	Formal System	17
4.1	Mathematical Definitions and Notations	17
4.2	How we use Rules	21
5	Type System Building Blocks	31
5.1	Static Environments	31
5.2	Constrained Types	33
5.3	ASL Type System	33
5.4	Annotation	34
6	Domain of Values for Types	35
6.1	Native Values	35
6.2	Dynamic Domain of a Type	36
6.3	Subsumption Testing	44
7	Basic Type Attributes	47
7.1	TypingRule.BuiltinSingularType	47
7.2	TypingRule.BuiltinAggregateType	49
7.3	TypingRule.BuiltinSingularOrAggregate	50
7.4	TypingRule.NamedType	51
7.5	TypingRule.AnonymousType	52
7.6	TypingRule.SingularType	52
7.7	TypingRule.AggregateType	53
7.8	TypingRule.StructuredType	53
7.9	TypingRule.NonPrimitiveType	54
7.10	TypingRule.PrimitiveType	56
7.11	TypingRule.Structure	56
7.12	TypingRule.Anonymize	58

7.13	TypingRule.CheckConstrainedInteger	59
8	Relations Over Types	61
8.1	TypingRule.Subtype	62
8.2	TypingRule.StructuralSubtypeSatisfaction	63
8.3	TypingRule.DomainSubtypeSatisfaction	68
8.4	TypingRule.SubtypeSatisfaction	69
8.5	TypingRule.TypeSatisfaction	70
8.6	TypingRule.TypeClash	74
8.7	TypingRule.LowestCommonAncestor	78
8.8	TypingRule.CheckUnop	82
8.9	TypingRule.CheckBinop	85
8.10	TypingRule.FindNamedLCA	93
8.11	TypingRule.AnnotateConstraintBinop	94
8.12	TypingRule.BinopFilterRhs	96
8.13	TypingRule.RefineConstraintBySign	97
8.14	TypingRule.ReduceToZOpt	100
8.15	TypingRule.RefineConstraints	101
8.16	TypingRule.FilterReduceConstraintDiv	103
8.17	TypingRule.GetLiteralDivOpt	103
8.18	TypingRule.ExplodeIntervals	104
8.19	TypingRule.ExplodeConstraint	105
8.20	TypingRule.IntervalTooLarge	106
8.21	TypingRule.BinopIsExploding	106
9	Typing of Types	109
9.1	TypingRule.TString	110
9.2	TypingRule.TReal	111
9.3	TypingRule.TBool	111
9.4	TypingRule.TNamed	112
9.5	TypingRule.TInt	113
9.6	TypingRule.TBits	115
9.7	TypingRule.TTuple	116
9.8	TypingRule.TArray	117
9.9	TypingRule.TEnumDecl	119
9.10	TypingRule.TRecordExceptionDecl	120
9.11	TypingRule.TNonDecl	121
9.12	TypingRule.AnnotateConstraint	122
9.13	TypingRule.GetVariableEnum	123
9.14	TypingRule.AnnotateStaticInteger	124
10	Typing of Bitfields	127
10.1	TypingRule.TBitField	127
10.2	TypingRule.TBitFields	129

11 Typing of Expressions	131
11.1 TypingRule.ELit	133
11.2 TypingRule.ELocalVar	133
11.3 TypingRule.EGlobalVarConstantVal	134
11.4 TypingRule.EGlobalVarConstantNoVal	134
11.5 TypingRule.EGlobalVar	135
11.6 TypingRule.EUndefIdent	135
11.7 TypingRule.Binop	135
11.8 TypingRule.Unop	136
11.9 TypingRule.ECond	136
11.10 TypingRule.ESlice	137
11.11 TypingRule.ESetter	138
11.12 TypingRule.ECall	139
11.13 TypingRule.EGetArray	139
11.14 TypingRule.ESliceOrEGetArrayError	140
11.15 TypingRule.ERecord	141
11.16 TypingRule.EGetRecordField	142
11.17 TypingRule.EGetBadRecordField	143
11.18 TypingRule.EGetBadBitField	143
11.19 TypingRule.EGetBitField	144
11.20 TypingRule.EGetBitFieldNested	145
11.21 TypingRule.EGetBitFieldTyped	145
11.22 TypingRule.EGetTupleItem	146
11.23 TypingRule.EGetBadField	147
11.24 TypingRule.EConcat	147
11.25 TypingRule.ETuple	149
11.26 TypingRule.EUnknown	149
11.27 TypingRule.EPattern	150
11.28 TypingRule.ATC	150
11.29 TypingRule.Lit	151
11.30 TypingRule.ExpressionList	153
11.31 TypingRule.ReduceSlicesToCall	153
11.32 TypingRule.StaticConstrainedInteger	155
11.33 TypingRule.CheckATC	155
11.34 TypingRule.SlicesWidth	157
11.35 TypingRule.SliceWidth	158
12 Typing of Left-Hand-Side Expressions	161
12.1 TypingRule.LEDiscard	162
12.2 TypingRule.LEVar	162
12.3 TypingRule.LEDestructuring	163
12.4 TypingRule.LESlice	164
12.5 TypingRule.LESetArray	165
12.6 TypingRule.LESetStructuredField	166
12.7 TypingRule.LESetBitField	167

12.8	TypingRule.LESetBadField	170
12.9	TypingRule.LEConcat	170
12.10	TypingRule.LEBits	171
13	Typing of Slices	173
13.1	TypingRule.SliceSingle	173
13.2	TypingRule.SliceLength	174
13.3	TypingRule.SliceRange	174
13.4	TypingRule.SliceStar	175
13.5	TypingRule.Slices	175
14	Typing of Patterns	177
14.1	TypingRule.PAll	177
14.2	TypingRule.PAny	178
14.3	TypingRule.PGeq	178
14.4	TypingRule.PLeq	179
14.5	TypingRule.PNot	180
14.6	TypingRule.PRange	180
14.7	TypingRule.PSingle	181
14.8	TypingRule.PMask	183
14.9	TypingRule.PTuple	184
15	Typing of Local Declarations	185
15.1	TypingRule.LDDiscard	185
15.2	TypingRule.LDVar	186
15.3	TypingRule.LDTyped	187
15.4	TypingRule.LDTuple	188
15.5	TypingRule.CheckCanBeInitializedWith	189
16	Typing of Statements	191
16.1	TypingRule.SPass	192
16.2	TypingRule.SAssign	192
16.3	TypingRule.SReturnNone	193
16.4	TypingRule.SReturnOne	194
16.5	TypingRule.SReturnSome	194
16.6	TypingRule.SSeq	195
16.7	TypingRule.SCall	195
16.8	TypingRule.SCond	196
16.9	TypingRule.SCase	197
16.10	TypingRule.SAssert	197
16.11	TypingRule.SWhile	198
16.12	TypingRule.SRepeat	198
16.13	TypingRule.SFor	199
16.14	TypingRule.SThrowNone	201
16.15	TypingRule.SThrowSome	201

16.16	TypingRule.STry	201
16.17	TypingRule.SDeclSome	203
16.18	TypingRule.SDeclNone	204
16.19	TypingRule.CaseAlt	204
16.20	TypingRule.SForConstraints	205
16.21	TypingRule.AnnotateLoopLimit	207
16.22	TypingRule.DeclareLocalConstant	207
16.23	TypingRule.AnnotateLocalDelItemUninit	209
17	Typing of Blocks	211
17.1	TypingRule.Block	211
18	Typing of Catchers	213
18.1	TypingRule.CatcherNone	213
18.2	TypingRule.CatcherSome	214
19	Typing of Subprogram Calls	215
19.1	TypingRule.AnnotateCall	216
19.2	TypingRule.AnnotateCallArgTyped	216
19.3	TypingRule.CheckCalleeParams	218
19.4	TypingRule.RenameTyEqs	220
19.5	TypingRule.SubstExprNormalize	222
19.6	TypingRule.SubstExpr	223
19.7	TypingRule.SubstConstraint	228
19.8	TypingRule.CheckArgsTypeSat	229
19.9	TypingRule.AnnotateParameterDefining	231
19.10	TypingRule.AnnotateRetTy	233
19.11	TypingRule.SubprogramForName	234
19.12	TypingRule.DeduceEqs	236
19.13	TypingRule.FilterCallCandidates	238
19.14	TypingRule.HasArgClash	239
20	Typing of Subprograms	241
20.1	TypingRule.Subprogram	241
21	Typing of Global Declarations	243
21.1	TypingRule.TypecheckFunc	244
21.2	TypingRule.TypecheckGlobalStorage	245
21.3	TypingRule.TypecheckTypeDecl	245
21.4	TypingRule.AnnotateAndDeclareFunc	246
21.5	TypingRule.AnnotateFuncSig	246
21.6	TypingRule.UseFuncSig	248
21.7	TypingRule.GetUndeclaredDefining	249
21.8	TypingRule.ScanForParams	250
21.9	TypingRule.AnnotateParams	251

21.10	TypingRule.AnnotateOneParam	253
21.11	TypingRule.ArgsAsParams	254
21.12	TypingRule.ArgAsParam	256
21.13	TypingRule.AnnotateParamType	257
21.14	TypingRule.AnnotateArgs	258
21.15	TypingRule.AnnotateOneArg	260
21.16	TypingRule.AnnotateReturnType	261
21.17	TypingRule.DeclareOneFunc	262
21.18	TypingRule.SubprogramClash	263
21.19	TypingRule.AddNewFunc	264
21.20	TypingRule.CheckSetterHasGetter	266
21.21	TypingRule.AddSubprogram	267
21.22	TypingRule.DeclareGlobalStorage	268
21.23	TypingRule.AnnotateTypeOpt	270
21.24	TypingRule.AnnotateExprOpt	271
21.25	TypingRule.AnnotateInitType	272
21.26	TypingRule.AddGlobalStorage	273
21.27	TypingRule.DeclareType	273
21.28	TypingRule.AnnotateExtraFields	274
21.29	TypingRule.AnnotateEnumLabels	276
21.30	TypingRule.DeclareConst	277
22	Typing of Specifications	279
22.1	TypingRule.TypeCheckAST	280
22.2	TypingRule.DeclsOfComp	281
22.3	TypingRule.AnnotateDeclComps	282
22.4	TypingRule.BuildDependencies	284
22.5	TypingRule.DeclDependencies	285
22.6	TypingRule.TypeCheckMutuallyRec	285
22.7	TypingRule.FoldEnvAndFs	287
22.8	TypingRule.DefDecl	288
22.9	TypingRule.DefEnumLabels	288
22.10	TypingRule.UseDecl	289
22.11	TypingRule.UseTy	290
22.12	TypingRule.UseSubtypes	293
22.13	TypingRule.UseExpr	294
22.14	TypingRule.UseLexpr	297
22.15	TypingRule.UsePattern	299
22.16	TypingRule.UseSlice	301
22.17	TypingRule.UseBitfield	302
22.18	TypingRule.UseConstraint	303
22.19	TypingRule.UseStmt	304
22.20	TypingRule.UseLDI	307
22.21	TypingRule.UseCase	308
22.22	TypingRule.UseCatcher	308

23 Static Evaluation	311
23.1 TypingRule.StaticEval	311
23.2 TypingRule.UnopLiterals	315
23.3 TypingRule.BinopLiterals	317
23.4 TypingRule.SlicesToPositions	330
23.5 TypingRule.SliceToPositions	331
23.6 TypingRule.EvalToInt	333
23.7 TypingRule.ExtractSlice	334
24 Symbolic Subsumption Testing	335
24.1 TypingRule.SymSubsumes	337
24.2 TypingRule.SymDomOfType	337
24.3 TypingRule.SymDomOfExpr	342
24.4 TypingRule.IntSetOp	344
24.5 TypingRule.IntSetToIntConstraints	346
24.6 TypingRule.SymDomOfLiteral	348
24.7 TypingRule.SymIntSetOfConstraints	349
24.8 TypingRule.ConstraintToIntSet	350
24.9 TypingRule.NormalizeToInt	351
24.10 TypingRule.SymDomIsSubset	352
24.11 TypingRule.SymIntSetSubset	354
24.12 TypingRule.ConstraintBinop	355
24.13 TypingRule.ConstraintBinopPair	356
24.14 TypingRule.IsRightIncreasing	359
24.15 TypingRule.IsRightDecreasing	360
24.16 TypingRule.IsLeftIncreasing	361
25 Symbolic Reduction and Equivalence Testing	363
25.1 Symbolic Expressions	364
25.2 TypingRule.Normalize	366
25.3 TypingRule.ReduceConstants	367
25.4 TypingRule.ReduceConstraint	368
25.5 TypingRule.ReduceConstraints	369
25.6 TypingRule.ToIR	370
25.7 TypingRule.ToIRCase	371
25.8 TypingRule.ExprEqualNorm	377
25.9 TypingRule.ExprEqual	378
25.10 TypingRule.ExprEqualCase	379
25.11 TypingRule.TypeEqual	386
25.12 TypingRule.BitwidthEqual	390
25.13 TypingRule.BitFieldsEqual	390
25.14 TypingRule.BitFieldEqual	391
25.15 TypingRule.ConstraintsEqual	393
25.16 TypingRule.ConstraintEqual	394
25.17 TypingRule.SlicesEqual	395

25.18	TypingRule.SliceEqual	396
25.19	TypingRule.ArrayLengthEqual	397
25.20	TypingRule.LiteralEqual	398
25.21	TypingRule.ReduceIR	399
25.22	TypingRule.PolynomialToExpr	399
25.23	TypingRule.CompareMonomialBindings	400
25.24	TypingRule.MonomialsToExpr	402
25.25	TypingRule.MonomialToExpr	403
25.26	TypingRule.SymAddExpr	404
25.27	TypingRule.UnitaryMonomialsToExpr	405
25.28	TypingRule.SymMulExpr	407
26	Utility Rules	409
26.1	Checked Transitions	409
26.2	Converting a List of Pairs to a Map	409
26.3	TypingRule.CheckNoDuplicates	410
26.4	Annotating Field Initializers	411
26.5	TypingRule.BitFieldGetName	412
26.6	TypingRule.CheckVarNotInGEnv	413
26.7	TypingRule.CheckVarNotInEnv	413
26.8	TypingRule.AddLocal	414
26.9	TypingRule.DeclaredType	415
26.10	TypingRule.FindBitfieldOpt	415
26.11	TypingRule.MemBfs	416
26.12	TypingRule.BitFieldsIncluded	420
26.13	TypingRule.TypeOfArrayLength	420
26.14	TypingRule.CheckStructureInteger	421
26.15	TypingRule.CheckStructure	422
26.16	TypingRule.StorageIsPure	422
26.17	TypingRule.CheckStaticallyEvaluable	423
26.18	TypingRule.ToWellConstrained	424
26.19	TypingRule.GetWellConstrainedStructure	425
26.20	TypingRule.GetBitvectorWidth	425
26.21	TypingRule.CheckBitsEqualWidth	426
26.22	AssocOpt	427
26.23	LookupConstant	428
26.24	TypeOf	428
26.25	TypingRule.IsUndefined	429
26.26	Sorting Lists	430
27	Type Error Codes	431

Chapter 1

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof

is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at

<https://www.arm.com/company/policies/trademarks>.

Copyright © [2023,2024] Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England. 110 Fulbourn Road, Cambridge, England CB1 9NJ. (LES-PRE-20349)

Chapter 2

Disclaimer

This document is part of the ASLRef material.

This material covers ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here:

<https://github.com/herd/herdtools7>.

A list of open items being worked on can be found here:

<https://github.com/herd/herdtools7/blob/master/asllib/doc/ASLRefProgress.tex>.

This material is work in progress, more precisely at Alpha quality as per Arm's quality standards. In particular, this means that it would be premature to base any production tool development on this material.

However, any feedback, question, query and feature request would be most welcome; those can be sent to Arm's Architecture Formal Team Lead Jade Alglave (jade.alglave@arm.com) or by raising issues or PRs to the herdtools7 github repository.

Chapter 3

Introduction

The purpose of this document is to describe, in a formal and authoritative way, which ASL specifications are considered *well-typed*. Whether a specification is well-typed is defined in terms of a *type system* [3]. That is, a set of *typing rules*.

An ASL parser accepts an ASL specification and checks whether it is valid with respect to the syntax of ASL, which is defined in [1]. If the specification is syntactically valid, the parser returns an *abstract syntax tree* (AST, for short), which represents the specification as a labelled structured tree. Otherwise, it returns a syntax error. When an ASL specification is successfully parsed, we refer to the resulting AST as the *untyped AST*.

A *type checker* is an implementation of the ASL type system, which accepts an untyped AST and applies the rules of the type system to the untyped AST. If it is successful, the specification is considered *well-typed* and the result is a pair consisting of a *static environment* and a *typed AST*, which are used in defining the ASL semantics [2]. Otherwise, the type checker returns a type error.

Related documents:

- The Abstract Syntax Reference [1] defines the abstract syntax, untyped AST, and typed AST.
- The ASL Semantics Reference [2] defines all valid behaviors of a well-typed ASL specification.

Understanding the Typing Formalization: We assume basic familiarity with the ASL language. The ASL type system is defined in terms of its AST, and familiarity with the AST is required to understand it. The mathematical background needed to understand the mathematical formalization of the ASL semantics appears in Chapter 4 and Chapter 5.

Reading guide: The typing rules are organized into chapters, which roughly group the rules by their AST node type. The set of rules in each chapter is further split according to additional syntactic and semantic predicates over the AST node. For example, an expression can be a literal, or a binary operator, amongst other things. Each of those has its own evaluation rule: `TypingRule.ELit` in Section 11.1 and `Typing.Binop` in Section 11.7, respectively.

Each rule is presented using the following template:

- a Prose paragraph gives the rule in English, and corresponds as much as possible to the code of the reference implementation `ASLRef` given at [/herdtools7/asllib](#);
- one or several Example paragraphs, which as much as possible are also given as regression tests in [/herdtools7/asllib/tests/ASLTypingReference.t](#);
- Formal paragraphs which give formal definitions of the rule.
- Comments paragraphs, which provide additional details.

Note that rules use a specific language to define variables, deconstruct mathematical data types, and assert equality. See Section 4.2.3 for details.

Chapter 4

Formal System

In this chapter, we define the mathematical concepts and notations used throughout. This chapter appears in all of the ASL references as its content is used in all of them. We start by defining general mathematical concepts and then describe how sets of rules formally define functions and relations.

4.1 Mathematical Definitions and Notations

We use \triangleq to define mathematical concepts.

We define the following sets:

- \mathbb{N} is the set of natural numbers, including 0.
- \mathbb{N}^+ is the set of natural numbers, excluding 0.
- \mathbb{Z} is the set of integers.
- \mathbb{Q} is the set of rationals.
- \mathbb{B} is the set of ASL Boolean literals, which consists of **TRUE** and **FALSE**. We employ these literals to represent the corresponding mathematical truth values, which are used to denote whether logical assertions hold or not. We also employ the mathematical meaning of logical conjunction \wedge , logical disjunction \vee , and logical negation \neg , given next. For a set of Boolean values A :

$$\begin{aligned} \bigwedge A &\triangleq \begin{cases} \text{TRUE} & \text{if all values in } A \text{ are TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases} \\ \bigvee A &\triangleq \begin{cases} \text{FALSE} & \text{if all values in } A \text{ are FALSE} \\ \text{TRUE} & \text{otherwise} \end{cases} \end{aligned}$$

For a pair of Boolean values $a, b \in \mathbb{B}$, we define $a \wedge b \triangleq \bigwedge \{a, b\}$ and $a \vee b \triangleq \bigvee \{a, b\}$. Finally, $\neg \text{TRUE} \triangleq \text{FALSE}$ and $\neg \text{FALSE} \triangleq \text{TRUE}$.

- \mathbb{I} is the set of all ASL identifiers.
- \mathbb{L} is the set of all labels of Abstract Syntax Tree (AST) nodes.
- \mathbb{S} is the set of all ASCII strings.

We utilize the notation \overbrace{a}^b to enable us to name the mathematical term a as b so that we can refer to it in text. We especially use this to name the input arguments and output results of functions and relations. For example, the input argument of sign , which is defined next is named q .

Definition 1 (Sign of a Rational Number) The function $\text{sign} : \overbrace{\mathbb{Q}}^q \rightarrow \{-1, 0, 1\}$ returns the sign of q :

$$\text{sign}(q) \triangleq \begin{cases} 1 & \text{if } q > 0 \\ 0 & \text{if } q = 0 \\ -1 & \text{if } q < 0 \end{cases}$$

Definition 2 (Empty Set) The empty set — the set that does not contain any element — is denoted as \emptyset .

Definition 3 (Set Cardinality) For a set S , the notation $|S|$ stands for the number of elements in S .

Definition 4 (Powerset) The powerset of a set A , denoted as $\mathcal{P}(A)$, is the set of all subsets of A , including the empty set and A itself:

$$\mathcal{P}(A) \triangleq \{B \mid B \subseteq A\} .$$

Definition 5 (Powerset of Finite Subsets) The powerset of finite subsets of a set A , denoted as $\mathcal{P}_{\text{fin}}(A)$, is the set of all finite subsets (including the empty set) of A :

$$\mathcal{P}_{\text{fin}}(A) \triangleq \{B \mid B \subseteq A, |B| \in \mathbb{N}\} .$$

Definition 6 (Cartesian Product) The Cartesian product of sets A and B , denoted $A \times B$ is $A \times B \triangleq \{(a, b) \mid a \in A, b \in B\}$.

Definition 7 (Partial Function) A partial function, denoted $f : A \rightarrow B$, is a function from a subset of A to B . The domain of a partial function f , denoted $\text{dom}(f)$, is the subset of A for which it is defined. We write $f(x) = \perp$ to denote that x is not in the domain of f , that is, $x \notin \text{dom}(f)$.

Notice that the domain of a partial function need not be finite, which is what the following definition covers.

Definition 8 (Finite-domain Function) The notation \rightarrow_{fin} stands for a function whose domain is finite.

Definition 9 (Empty Function) The function with an empty domain is denoted as \emptyset_λ .

Definition 10 (Function Update) The function denoted as $f[x \mapsto v]$ is a function identical to f , except that x is bound to v . That is, if $g = f[x \mapsto v]$ then

$$g(z) = \begin{cases} v & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases} .$$

The notation $\{i = 1..k : a_i \mapsto b_i\}$ stands for the function formed from the corresponding input-output pairs: $\emptyset_\lambda[a_1 \mapsto b_1] \dots [a_k \mapsto b_k]$.

Definition 11 (Function Restriction) The restriction of a function $f : X \rightarrow Y$ to a subset of its domain $A \subseteq \text{dom}(f)$, denoted as $f|_A$, is defined in terms of the set of input-output pairs:

$$f|_A \triangleq \{(x, f(x)) \mid x \in A\} .$$

Definition 12 (Function Graph) The graph of a finite-domain function $f : X \rightarrow_{fn} Y$ is the list of input-output pairs for f , given in any order:

$$\text{func_graph}(f) \triangleq \{(x, f(x)) \mid x \in \text{dom}(f)\} .$$

Throughout this document, we will annotate arguments of relations and functions, wherever it is useful, by writing a name or an expression above the corresponding argument type. This makes convenient to refer to arguments by referring to the corresponding names and helps identify the expressions corresponding to the arguments. For example,

$$\text{choice} : \overbrace{\mathbb{B}}^b \times \overbrace{T}^x \times \overbrace{T}^y \rightarrow \overbrace{T}^z$$

defines a function type and lets us refer to the first argument as b , the second argument as x , the third argument as y , and to the result as z .

A *parametric function* is a function whose domain is not a priori fixed but rather parameterized by the type of its arguments. An example is the `choice` function where the type T of x , y , and z is unspecified and inferred from the context where the function is used.

Definition 13 (Choice) The parametric function $\text{choice} : \overbrace{\mathbb{B}}^b \times \overbrace{T}^x \times \overbrace{T}^y \rightarrow \overbrace{T}^z$, is defined as follows:

$$\text{choice}(b, x, y) \triangleq \begin{cases} x & \text{if } b \text{ is } \text{TRUE} \\ y & \text{otherwise} \end{cases}$$

4.1.1 Lists

In the remainder of this document, we use the term *list* and *sequence* interchangeably.

A list of elements is either empty, denoted by $[]$, or non-empty. A non-empty list is either denoted by listing the elements in sequence, $v_1 \dots v_k$, or in bracketed form,

$[v_1, \dots, v_k]$, which is used to aesthetically separate it from surrounding mathematical expressions. The commas carry no special meaning.

For a non-empty list $v_1 \dots v_k$, the **head** of the list is the first element — v_1 — and the **tail** of the list is the suffix obtained by removing v_1 from the list.

We refer to individual elements of a non-empty list V by the index notation $V[i]$ where $i \in \mathbb{N}^+$.

Definition 14 (List Length) *The length of a list is the number of elements in that list: $[[]] \triangleq 0$ and $|v_1, \dots, v_k| = k$.*

We use the notation $a..b$, where $a, b \in \mathbb{Z}$ and $a \leq b$, as a shorthand for the interval $[a \dots b]$. We write $x_{a..b}$ as a shorthand for the sequence $x_a \dots x_b$. We write $i = 1..k : V(i)$, where $V(i)$ is a mathematical expression parameterized by i , to denote the sequence of expressions $V(1) \dots V(k)$. The notation $a \in A : V(a)$, where A is a set and V is an expression parameterized by the free variable a , stands for $V(a_1) \dots V(a_k)$ where $a_{1..k}$ is an arbitrary ordering of the elements of A .

We write T^* to denote the type of a possibly-empty list of elements of type T , and T^+ for a non-empty list of elements of type T .

Definition 15 (List Concatenation) *The parametric function $+$: $T^* \times T^* \rightarrow T^*$ concatenates two lists:*

$$\begin{aligned} [] + L &\triangleq L \\ L + [] &\triangleq L \\ l_{1..k} + m_{1..n} &\triangleq [l_{1..k}, m_{1..n}] \end{aligned}$$

Definition 16 (Equating List Lengths) *The parametric function*

$$\text{equal_length} : \overbrace{L}^a \times \overbrace{L}^b \rightarrow \mathbb{B}$$

compares the length of two lists:

$$\text{equal_length}(a, b) \triangleq |a| = |b| .$$

Definition 17 (Indices of a List) *The parametric function $\text{indices} : T^* \rightarrow \mathbb{N}^*$ returns the (1-based) list of indices for a given list:*

$$\begin{aligned} \text{indices}([]) &\triangleq [] \\ \text{indices}(v_{1..k}) &\triangleq [1..k] . \end{aligned}$$

Definition 18 (Unzipping a List of Pairs) *The parametric function*

$$\text{unzip} : (T_1 \times T_2)^* \rightarrow (T_1^* \times T_2^*)$$

transforms a list of pairs into the corresponding pair of lists:

$$\text{unzip}(\text{pairs}) \triangleq \begin{cases} ([], []) & \text{if } \text{pairs} = [] \\ (a_{1..k}, b_{1..k}) & \text{else } \text{pairs} = (a_1, b_1) \dots (a_k, b_k) . \end{cases}$$

4.1.2 OCaml-style Notations

We use the following notations, which are in the style of the OCaml programming language, to facilitate correspondence with our [reference implementation](#).

The notation $L(v_{1..k})$ is a compound term where L is a label and $v_{1..k}$ is a (possibly singleton) list of mathematical values. We also write $L(T_{1..k})$, where $T_{1..k}$ denotes mathematical types of values, to stand for the type $\{L(v_{1..k}) \mid v_1 \in T_1, \dots, v_k \in T_k\}$.

Definition 19 (Optional) *The notation $\langle \cdot \rangle$ stands for either an empty set or a singleton set, where $\text{None} \triangleq \langle \rangle$ denotes an empty set and $\langle v \rangle$ denotes a set containing the single element v . The notation $\langle T \rangle$, where T denotes a mathematical type, stands for $\{\langle \rangle\} \cup \{\langle v \rangle \mid v \in T\}$.*

We refer to $\langle T \rangle$ as an optional.

4.2 How we use Rules

An *inference rule* (rule, for short) is an implication between a set of logical assertions, called the *premises* of the rule, and a *conclusion* assertion. The conclusion holds when the conjunction of its premises holds.

We use the following rule notation, where $P_{1..k}$ are the rule premises and C is the conclusion:

$$\frac{P_1 \quad \dots \quad P_k}{C}$$

For example, the rule `TypingRule.ELit` has one premise:

$$\frac{\text{annotate_literal}(v) \xrightarrow{\text{type}} t}{\text{annotate_expr}(\text{tenv}, \text{E_Literal}(v)) \xrightarrow{\text{type}} (t, \text{E_Literal}(v))}$$

and the rule `TypingRule.Binop` (somewhat simplified here) has three premises:

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t1, e1') \\ \text{annotate_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t2, e2') \\ \text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} t \end{array}}{\text{annotate_expr}(\text{tenv}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{type}} (t, \text{E_Binop}(\text{op}, e1', e2'))}$$

The free variables appearing in the premises and conclusion are interpreted universally. That is, the rules apply to any values (of the appropriate types) assigned to their free variables. For example, the rule `TypingRule.Binop` applies to any choice of values for the free variables `tenv` (a static environment), `e1`, `e2`, `e1'`, `e2'` (expressions), `t`, `t1`, and `t2` (types).

Definition 20 (Grounding) *Assertions can be grounded by substituting their free variables with values. A ground rule is a rule with all its assertions (premises and conclusion) grounded.*

For example, the following is a grounding of `TypingRule.Binop`

$$\begin{array}{c}
 \text{annotate_expr}(\emptyset_{\text{tenv}}, \text{E_Literal}(\text{L_Int}(2))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(2))) \\
 \text{annotate_expr}(\emptyset_{\text{tenv}}, \text{E_Literal}(\text{L_Int}(3))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(3))) \\
 \text{check_binop}(\emptyset_{\text{tenv}}, \text{MUL}, \text{T_Int}, \text{T_Int}) \xrightarrow{\text{type}} \text{T_Int} \\
 \hline
 \text{annotate_expr}(\emptyset_{\text{tenv}}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3)))) \xrightarrow{\text{type}} \\
 (\text{T_Int}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3))))
 \end{array}$$

obtained by the following substitutions:

free variable	value
tenv	\emptyset_{tenv}
e1	<code>E_Literal(L_Int(2))</code>
e1'	<code>E_Literal(L_Int(2))</code>
e2	<code>E_Literal(L_Int(3))</code>
e2'	<code>E_Literal(L_Int(3))</code>
t	<code>T_Int</code>
t1	<code>T_Int</code>
t2	<code>T_Int</code>
op	<code>MUL</code>

A set of rules is interpreted disjunctively. That is, each rule is used to determine whether its conclusion holds independently of other rules.

Definition 21 (Axiom) *An axiom is a rule with an empty set of premises. An axiom is denoted by simply stating its conclusion.*

An example of an axiom in the ASL type system is `TypingRule.SPass`:

$$\text{annotate_stmt}(\text{tenv}, \text{S_Pass}) \xrightarrow{\text{type}} (\text{S_Pass}, \text{tenv})$$

An example of an axiom in the ASL semantics is `SemanticsRule.PAll`:

$$\text{eval_pattern}(\text{env}, _, \text{Pattern_All}) \xrightarrow{\text{eval}} \text{Normal}(\text{Bool}(\text{TRUE}), \emptyset_g)$$

To show that a specification is correct, with respect to the set of type rules, or to show that a specification evaluates to a certain value, with respect to the set of semantic rules, we must apply rules to form a *derivation tree*.

Definition 22 (Derivation Tree) *A derivation tree is a tree whose vertices correspond to ground assertions. More specifically, the leaves of a derivation tree correspond to ground axioms, and an internal vertex corresponds to a ground conclusion of a rule with its children corresponding to the ground premises of the same rule.*

4.2.1 Transitions

We use rules as a structured way for defining relations (and therefore functions, as a special case).

To define a relation $R \subseteq X \times Y$, we use assertions of the form $tx \xrightarrow{R} ty$ where tx and ty are logical terms denoting sets of elements from X and Y , respectively. We call such assertions *transitions*. A set of rules M with transition assertions defines the relation

$$R = \{(x, y) \mid x \xrightarrow{R} y \text{ can be derived from rules in } M\} .$$

For example, the rule [TypingRule.ELit](#) defines a relation between the infinite set of elements of the form `annotate_expr(tenv, E.Literal(v))` (for the infinite choice of values for the free variables `tenv` and `v`) to the infinite set of pairs of the form `(t, E.Literal(v))`, such that the premise holds.

Mutual Exclusion Principle: Our rules follow (with very few deviations, which we point out in context) a mutual exclusion principle, where each rule defines a relation disjoint from the ones defined by the other rules. This makes it easy to determine the rule responsible for a given transition.

4.2.2 Configurations

Our relations range over compound values. That is, values that often nest tuples and lists inside other tuples and lists. We refer to such values as *configurations*. To make it easier to distinguish between different configurations, we will sometimes attach labels to tuples using the OCaml-style notation discussed earlier. We refer to those labels as *configuration domains*. The domain of a configuration $C = L(\dots)$, denoted `config_domain(C)`, is the label L .

We refer to configurations at the origin of a transition as *input configurations* and to the configurations at the destination of a transition as *output transitions*.

For example, the conclusion of the rule [TypingRule.ELit](#) has `annotate_expr(tenv, E.Literal(v))` as its input configuration and `(t, E.Literal(v))` as its output configuration. Further, `config_domain(annotate_expr(tenv, E.Literal(v))) = annotate_expr`, while the output configuration does not have a configuration domain, since it is an unlabelled pair.

Our rules always make use of labelled input configurations. This makes it easier to ensure the mutual exclusion rule principle.

Our rules always define relations whose sets of input configurations and output configurations are disjoint.

Definition 23 (Fresh Element) *Premises of the form $x \in T$ is fresh mean that in any instantiation in a derivation tree, the value of x is unique. That is, different from all other values instantiated for any other variable.*

Definition 24 (Ignore Variable) *To keep rules succinct, we write `_` for a mathematical variable whose name is irrelevant for understanding the rule, and can thus be omitted. Each occurrence of `_` represents a variable whose name is different from any other free variable in the rule.*

For example, the rule `SemanticsRule.PAll`, shown [above](#), uses an ignore variable to stand for the value being matched by a `-` pattern. Since the rule does not need to refer to the value, we do not name it and use an ignore variable instead.

4.2.3 Flavors of Equality In Rules

We now explain equality notations in rules, two of which are used in `SemanticsRule.Lit`, shown here:

$$\frac{\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(L^{\text{denv}}) \quad \mathbf{v} := L^{\text{denv}}(\mathbf{x}) \quad \mathbf{g} := \text{ReadEffect}(\mathbf{x})}{\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}((\mathbf{v}, \mathbf{g}), \text{env})}$$

Range: we write $i = 1..k$ to allow listing premises parameterized by i or constructing lists from expressions parameterized by i . For example, given two lists a and b ,

$$i = 1..k : a[i] > b[i]$$

is the list of premises

$$\begin{aligned} &a[0] > b[0] \\ &\dots \\ &a[k] > b[k] \end{aligned}$$

Predicate: we write $a = b$ as an assertion of the equality of a and b . For example, the mathematical identity $x \times (y + z) = x \times y + x \times z$.

Deconstruction / “View as”: some values, such as tuples, are compound. In order to refer to the structure of compound values, we write $v \stackrel{\text{is}}{=} f(u_{1..k})$ where the expression on the right hand side exposes the internal structure of v by introducing the variables $u_{1..k}$, allowing us to alias internal components of v . Intuitively, v is re-interpreted as $f(u_{1..k})$. For example, suppose we know that v is a pair of values. Then, $v \stackrel{\text{is}}{=} (a, b)$ allows us to alias a and b . In `SemanticsRule.Lit`, we know that the environment `env` is a pair where the first component is a static environment and the second component is a dynamic environment. Therefore, writing `env` $\stackrel{\text{is}}{=} (_, \text{denv})$ allows us to name the dynamic environment component and then refer to it, while [ignoring](#) the static environment component. Similarly, if v is a non-empty list, then $v \stackrel{\text{is}}{=} [h] + t$ deconstructs the list into the head of the list h and its tail t . Given that a variable v represents a list, we write $v \stackrel{\text{is}}{=} v_{1..k}$ to list its elements and allow referring to them by index.

Definition / “Define as”: the notation $\mathbf{x} := \mathbf{e}$ denotes that \mathbf{x} is a new name serving as an alias for the expression \mathbf{e} . For example, in the rule `SemanticsRule.Lit`, we use \mathbf{g} to name `ReadEffect(x)`. Aliases allow us to break down complex expressions, but rules can always be rewritten without them, by inlining their right-hand sides:

$$\frac{\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(L^{\text{denv}})}{\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}((L^{\text{denv}}(\mathbf{x}), \text{ReadEffect}(\mathbf{x})), \text{env})}$$

4.2.4 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_2, \dots, f_k : t_k\}$, we sometimes only care about a subset of the fields $\{f_{i_1}, \dots, f_{i_m}\} \subset \{f_{1..k}\}$. In such cases, we write $\{f_{i_1} : t_{i_1}, \dots, f_{i_m} : t_{i_m}, \dots\}$, where \dots stands for fields that are irrelevant for the rule.

For example¹, the `func` non-terminal is of a record type and has the following fields: `name`, `parameters`, `args`, `body`, `return_type`, and `subprogram_type`. The notation $\{\text{body} : \text{SB_ASL}(\text{body}), \text{args} : \text{arg_decls}, \dots\}$ allows us to deconstruct a given `func` node by matching only the `body` and `args` fields.

Recall that a subset of AST nodes are either labels or labelled tuples. The partial function `ast_label` returns the label $l \in \mathbb{L}$ an AST node, when it exists. For example, `ast_label(T_Boolean) = T_Boolean` and `ast_label(T_Named(x)) = T_Named`.

4.2.5 How to Parse Rules Efficiently

Consider the following examples, which is a simplified version of `SemanticsRule.Binop`

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \\
 \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env}) \\
 \text{m1} \stackrel{\text{is}}{=} (\text{v1}, \text{g1}) \quad \text{m2} \stackrel{\text{is}}{=} (\text{v2}, \text{g2}) \quad \text{binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \\
 \quad \quad \quad \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

To parse a rule, start by examining the conclusion and the variables appearing in the rule. In this case, the rule describes a transition from an input configuration $\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2}))$, whose configuration domain is `eval_expr`, to an output configuration $\text{Normal}((\text{v}, \text{g}), \text{new_env})$ whose configuration domain is `Normal`. A rule uses the free variables appearing in the input configuration of the conclusion (`env`, `op`, `e1`, and `e2` in our example), with the goal of assigning values to the free variables in the output configuration of the conclusion (`v`, `g`, and `new_env`, in our example).

Now, scan the premises in order to see where `env`, `op`, `e1`, and `e2` are used and how premises assign values to `v`, `g`, and `new_env`. In this case, `v` is assigned as the result of the transition assertion $\text{binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v}$, `g` is assigned the expression $\text{g1} \parallel \text{g2}$, and `new_env` is assigned as the result of the transition assertion $\text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env})$. Notice that to assign values to the variables `v`, `g`, and `new_env`, intermediate values have to be assigned first. For example, $\text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1})$ assigned values to `env1`, which is then used by the transition $\text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env})$. Similarly, `g` requires first assigning values to `g1` and `g2`, which are components of the previously assigned variables `m1` and `m2`.

¹This example is from `SemanticsRule.FCall`.

4.2.6 Short-Circuit Rule Macros

Short-circuit rule macros, or *rule macros*, for short, allow us to succinctly define sets of rules. Specifically, they allow us to capture situations where transitions have two alternative output configurations. If the transition results in the first of the alternative output configurations, the following premises are considered. However, if the result is the second, short-circuit output configuration, then the following premises are ignored and the conclusion transitions into the short-circuit output configuration. These short-circuit output configurations are typically, but not always, due to (type or dynamic) errors.

In the following, XP and XQ stand for, possibly empty, sequences of premises. A rule macro includes the special premise form $C \xrightarrow{R} C' \parallel E$, which introduces alternative output configurations C' and short-circuit E :

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \parallel E \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Such a rule macro expands to the following pair of rules:

$$\begin{array}{cc} \text{(OPTION 1)} & \text{(OPTION 2:SHORT-CIRCUITED)} \\ \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \\ XQ \end{array}}{V \xrightarrow{R} V'} & \frac{\begin{array}{c} XP \\ C \xrightarrow{R} E \\ XQ \end{array}}{V \xrightarrow{R} E} \end{array}$$

Intuitively, if C transitions to C' then $\parallel E$ can be ignored and the rule is interpreted as usual (Option 1). However, if C transitions into E (Option 2) then the premises XQ are ignored, thereby short-circuiting the rule, and the input configuration in the conclusion also transitions into E .

We allow more than one premise to include short-circuiting alternatives and also a single premise to include several alternatives. That is, a rule macro of the form

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \parallel E_{1\dots m} \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Stands for the set of rule macros

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \parallel E_1 \\ XQ \end{array}}{V \xrightarrow{R} V'} \quad \dots \quad \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \parallel E_m \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Notice that after all rule macros are expanded, in a top-to-bottom and left-to-right order, into normal rules, they behave like normal rules where the order of premises does not matter.

Alternative Outcomes Expressed in English Prose: In English prose, we use $\text{\textcolor{blue}{\textit{//}x,y,\dots}}$ to mean “if the outcome is one of x, y, \dots then the result short-circuits the rule.

As an example, consider the rule `SemanticsRule.Binop`. This time, not simplified:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \text{ \textcolor{blue}{//} \#T, \#DE} \\
 \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env}) \text{ \textcolor{blue}{//} \#T, \#DE} \\
 \text{m1} \stackrel{\text{is}}{=} (\text{v1}, \text{g1}) \quad \text{m2} \stackrel{\text{is}}{=} (\text{v2}, \text{g2}) \quad \text{binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \text{ \textcolor{blue}{//} \#DE} \\
 \quad \quad \quad \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

In this rule, $\text{\textcolor{blue}{\#T}}$ and $\text{\textcolor{blue}{\#DE}}$ are just shorthand notations for actual configurations, which are properly defined in the semantics reference. Intuitively, the alternative configurations $\text{\textcolor{blue}{\#T}}$ and $\text{\textcolor{blue}{\#DE}}$ represent situations where a transition may result in a raised exception and a dynamic error, respectively.

One may first read the rule ignoring these alternative configurations, to see how the goal of transitioning into the output configuration appearing in the conclusion — $\text{Normal}((\text{v}, \text{g}), \text{new_env})$ — is achieved. Then, re-reading the rule would indicate where exceptions and dynamic errors may result in other output configurations. For example, if the first transition assertion results in a throwing configuration $\text{\textcolor{blue}{\#T}}$ then the output configuration of the conclusion is also $\text{\textcolor{blue}{\#T}}$. This corresponds to the following rule in the expanded macro:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{\textcolor{blue}{\#T}} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{\textcolor{blue}{\#T}}
 \end{array}$$

Similarly, if the first transition assertion results in a dynamic error, the output configuration of the conclusion is that dynamic error, which corresponds to the following rule in the expansion:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{\textcolor{blue}{\#DE}} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{\textcolor{blue}{\#DE}}
 \end{array}$$

The following rules correspond to the cases where the first transition results in $\text{Normal}(\text{m1}, \text{env1})$, but the second transition assertion results in either $\text{\textcolor{blue}{\#T}}$ or $\text{\textcolor{blue}{\#DE}}$, respec-

tively:

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \\ \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \#T \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#T}$$

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \\ \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \#DE \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DE}$$

Expanding the last transition assertion, gives us the case:

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \\ \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env}) \\ \text{m1} \stackrel{\text{is}}{=} (\text{v1}, \text{g1}) \quad \text{m2} \stackrel{\text{is}}{=} (\text{v2}, \text{g2}) \quad \text{binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \#DE \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DE}$$

All these cases are succinctly encoded in a single rule with the alternative output configurations.

4.2.7 Boolean Transition Assertions

We define the following rules to allow us to treat assertions as transition assertions:

$$\frac{\text{BOOL_TRANS_TRUE}}{\text{bool_transition}(\text{TRUE}) \longrightarrow \text{TRUE}} \quad \frac{\text{BOOL_TRANS_FALSE}}{\text{bool_transition}(\text{FALSE}) \longrightarrow \text{FALSE}}$$

This is useful in that it allows us to use assertions in rule macros.

4.2.8 Rule Naming

To name a rule, we place it in a section with its name. However, some relations are defined by a group of rules. In such cases, we refer to the individual rules in a group as *case rules*, or simply *cases*. We annotate case rules by names appearing above and to the left of the rule. The name of these case rules is the name of the group, given by its section, followed by the name of the case.

For example, the ASL Semantics Reference defines the rule SemanticsRule.BaseValue using 11 cases of which two are the following:

$$\frac{\text{BOOL} \quad \text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{T_Bool}}{\text{base_value}(\text{env}, \text{t}) \xrightarrow{\text{eval}} (\text{Bool}(\text{TRUE}), \emptyset_g)} \quad \frac{\text{REAL} \quad \text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{T_Real}}{\text{base_value}(\text{env}, \text{t}) \xrightarrow{\text{eval}} (\text{Real}(0), \emptyset_g)}$$

The full name of the first case is then `SemanticsRule.BaseValue.BOOL` and the full name of the second case is `SemanticsRule.BaseValue.REAL`.

When explaining rules in English prose, we include the name of the case rules in parenthesis to make it easier to relate the prose to the corresponding mathematical definitions (see, for example, the Prose paragraph of `SemanticsRule.BaseValue` or that of `TypingRule.CheckUnop`).

4.2.9 Generic Notations

- The notation \hookrightarrow denotes that a line that is longer than the page width continues on the next line.
- The notation `***** common prefix *****` serves as a visual aid to delimit a common prefix of premises shared by rule cases.
- The notation `***** common suffix *****` serves as a visual aid to delimit a common suffix of premises shared by rule cases.
- **Missing definition:** Red hyperlinks indicate items that are yet to be defined.

Chapter 5

Type System Building Blocks

This chapter defines necessary mathematical types and concepts for the ASL system.

Types are represented by ASTs derived from the non-terminal `ty` (see [1] for the precise definition of `ty`).

5.1 Static Environments

A *static environment* (also called a *type environment*) is what the typing rules operate over: a structure, which amongst other things, associates types to variables. Throughout this document, we will use the term environment for static environment, unless otherwise stated. Intuitively, the typing of a specification makes an initial environment evolve, with new types as given by the variable declarations of the specification.

Definition 25 *Static environments, denoted as \mathbb{SE} , are defined as follows (referring to symbols defined by the abstract syntax):*

\mathbb{SE}	\triangleq	$\mathbb{G} \times \mathbb{L}$
\mathbb{G}	\triangleq	$\text{declared_types} \times \text{constant_values} \times \text{global_storage_types}$ $\times \text{subtypes} \times \text{subprograms} \times \text{subprogram_renamings}$
\mathbb{L}	\triangleq	$\text{constant_values} \times \text{local_storage_types} \times \text{return_type}$
declared_types	\triangleq	$\text{identifier} \rightarrow \text{ty}$
constant_values	\triangleq	$\text{identifier} \rightarrow \text{literal}$
$\text{global_storage_types}$	\triangleq	$\text{identifier} \rightarrow \text{ty} \times \text{global_decl_keyword}$
$\text{local_storage_types}$	\triangleq	$\text{identifier} \rightarrow \text{ty} \times \text{local_decl_keyword}$
subtypes	\triangleq	$\text{identifier} \rightarrow \text{identifier}$
subprograms	\triangleq	$\text{identifier} \rightarrow \text{func}$
$\text{subprogram_renamings}$	\triangleq	$\text{identifier} \rightarrow \mathcal{P}(\mathbb{S})$
return_type	\triangleq	$\langle \text{ty} \rangle$

We use `tenv` and similar variable names (for example, `tenv1` and `new_tenv`) to range over static environments.

A static environment $\text{tenv} = (G^{\text{tenv}}, L^{\text{tenv}})$ consists of two distinct components: the global environment $G^{\text{tenv}} \in \mathbb{G}$ — pertaining to AST nodes appearing outside of a given subprogram, and the local environment $L^{\text{tenv}} \in \mathbb{L}$ — pertaining to AST nodes appearing inside a given subprogram. This separation allows us to type-check subprograms by using an empty local environment.

The intuitive meaning of each component is as follows:

- **declared_types** assigns types to their declared names;
- **constant_values** assigns literals to their declaring (constant) names;
- **global_storage_types** associates names of global storage elements to their inferred type and how they were declared — as constants, configuration variables, **let** variables, or mutable variables;
- **local_storage_types** associates names of local storage elements to their inferred type and how they were declared — as variables, constants, or as **let** variables;
- **subtypes** associates type names to the names that their type subtypes;
- **subprograms** associates names of subprograms to the **func** AST node they were declared with;
- **subprogram_renamings** associates names of subprograms to the set of overloading subprograms — **func** AST nodes that share the same name;
- **return_type** contains the name of the type that a subprogram declares, if it is a function or a getter.

Definition 26 (Empty Static Environment) *The empty static environment, denoted as \emptyset_{tenv} , is defined as follows:*

$$\emptyset_{\text{tenv}} \triangleq \left(\begin{array}{c} \overbrace{\text{declared_types} \quad \text{constant_values} \quad \text{global_storage_types} \quad \text{subtypes} \quad \text{subprograms} \quad \text{subprogram_renamings}}^G \\ \underbrace{\text{constant_values} \quad \text{local_storage_types} \quad \text{return_type}}_L \\ \left(\underbrace{\emptyset_\lambda}_{\text{declared_types}}, \underbrace{\emptyset_\lambda}_{\text{constant_values}}, \underbrace{\emptyset_\lambda}_{\text{global_storage_types}}, \underbrace{\emptyset_\lambda}_{\text{subtypes}}, \underbrace{\emptyset_\lambda}_{\text{subprograms}}, \underbrace{\emptyset_\lambda}_{\text{subprogram_renamings}} \right), \\ \left(\underbrace{\emptyset_\lambda}_{\text{constant_values}}, \underbrace{\emptyset_\lambda}_{\text{local_storage_types}}, \underbrace{\text{None}}_{\text{return_type}} \right) \end{array} \right)$$

The global environment and local environment consist of various components. We use the notation $G^{\text{tenv}}.m$ and $L^{\text{tenv}}.m$ to access the m component of a given environment.

To update a function component f (e.g., **declared_types**) of a global or local environment E with a new mapping $x \mapsto v$, we use the notation $\text{tenv}.f[x \mapsto v]$ to stand for $E[f \mapsto E.f[x \mapsto v]]$.

5.2 Constrained Types

- A *constrained type* is a type whose definition is parameterized by an expression. In ASL only integer types and bitvector types can be constrained.
- A type which is not constrained is *unconstrained*.
- A constrained type with a non-empty constraint is *well-constrained*.
- A *parameterized integer type* is an implicit type of a subprogram parameter.

The widths of bitvector storage elements are constrained integers.

We use the following helper predicates to classify integer types:

$$\begin{aligned}
 \text{is_unconstrained_integer}(\overset{t}{\text{ty}}) &\longrightarrow \mathbb{B} \\
 \text{is_parameterized_integer}(\overset{t}{\text{ty}}) &\longrightarrow \mathbb{B} \\
 \text{is_well_constrained_integer}(\overset{t}{\text{ty}}) &\longrightarrow \mathbb{B}
 \end{aligned}$$

Those are defined as follows:

$$\begin{aligned}
 \text{is_unconstrained_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{Unconstrained} \\
 \text{is_parameterized_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{Parameterized} \\
 \text{is_well_constrained_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{WellConstrained}
 \end{aligned}$$

5.3 ASL Type System

The type system of ASL is given by the relation `type`, which is defined as the disjoint union of the functions and relations defined in this document. The functions and relations in this document are defined, in turn, via type system rules.

The output configurations of type assertions have two flavors:

Normal Outputs. Configurations are typically tuples with different combinations of *static environments*, types, and Boolean values.

Type Errors. Configurations in `TypeError(S)` represent type errors, for example, using an integer type as a condition expression, as in `if 5 then 1 else 2`. The ASL type system is designed such that when these *type error configurations* appear, the typing of the entire specification terminates by outputting them.

We define the mathematical type of type error configurations (which is needed to define the types of functions in the ASL type system) as follows:

$$\text{TTypeError} \triangleq \{\text{TypeError}(s) \mid s \in \mathbb{S}\}.$$

and the shorthand $\#TE \triangleq \text{TypeError}(s)$ for type error configurations.

When several *case rules* for the same function use the same short-circuiting transition assertion, we do not repeat the $\#TE$, but rather include it only in the first rule.

5.4 Annotation

Typing a specification consists of annotating the root of its AST with the rules defined in the remainder of this document.

Shorthand Notations: We use the shorthand notation `unconstrained_integer` to denote the unconstrained integer type: `T.Int(Unconstrained)`.

We employ the following abbreviations for various AST nodes:

Abbreviation	Meaning
\overline{n} <small>E.Literal(L.Int)</small>	literal integer expression: <code>E.Literal(L.Int(<i>n</i>))</code>
\overline{e} <small>Constraint.Exact</small>	<code>Constraint.Exact(<i>e</i>)</code>
$\overline{e1..e2}$ <small>Constraint.Range</small>	<code>Constraint.Range(<i>e1</i>, <i>e2</i>)</code>
$\overline{e1 \text{ op } e2}$ <small>E.Binop</small>	<code>E.Binop(<i>op</i>, <i>e1</i>, <i>e2</i>)</code>
$\overline{\text{array } [i] \text{ of } t}$ <small>T.Array</small>	<code>T.Array(ArrayLength.Expr(<i>i</i>), <i>t</i>)</code>
$\overline{\text{array } [e] \text{ of } t}$ <small>T.Array(ArrayLength.Expr)</small>	<code>T.Array(ArrayLength.Expr(<i>e</i>), <i>t</i>)</code>
$\overline{\text{array } [e\#s] \text{ of } t}$ <small>T.Array(ArrayLength.Enum)</small>	<code>T.Array(ArrayLength.Enum(<i>e</i>, <i>s</i>), <i>t</i>)</code>

Note that throughout this document we use `ty` to denote a type variable, which should not be confused with the abstract syntax variable `ty`.

Chapter 6

Domain of Values for Types

This chapter formalizes the concept of the set of values for a given type. The formalism is given in the form of rules. The chapter also defines the concept of checking whether the set of values for one type is included in the set of values for another type.

6.1 Native Values

Types define sets of values that variables can take when a specification is semantically evaluated. To formalize this, we define the set of *native values*, denoted \mathbb{V} (NV stands for Native Value).

6.1.1 Prose

The set of native values \mathbb{V} is the minimal set satisfying all of the following rules:

- BASIS SET: if v is a literal then $\text{NV_Literal}(v)$ is a native value;
- TUPLE VALUES AND ARRAY VALUES: if l is a list of native values then $\text{NV_Vector}(l)$ is a native value;
- RECORD VALUES: if r is a finite function from identifiers to native values then $\text{NV_Record}(r)$ is a native value.

6.1.2 Formally

(BASIS SET: INTEGERS, REALS, BOOLEANS, STRINGS, AND BITVECTORS)

$$\frac{v \in \text{literal}}{\text{NV_Literal}(v) \in \mathbb{V}}$$

(TUPLE VALUES AND ARRAY VALUES)

$$\frac{vl \in \mathbb{V}^*}{\text{NV_Vector}(vl) \in \mathbb{V}}$$

(RECORD VALUES)

$$\frac{r : \mathbb{I} \rightarrow_{\text{fin}} \mathbb{V}}{\text{NV_Record}(r) \in \mathbb{V}}$$

We define the following shorthands for native value literals:

$$\begin{aligned}
\text{Int}(z) &\triangleq \text{NV_Literal}(\text{L_Int}(z)) \\
\text{Bool}(b) &\triangleq \text{NV_Literal}(\text{L_Bool}(b)) \\
\text{Real}(r) &\triangleq \text{NV_Literal}(\text{L_Real}(r)) \\
\text{String}(s) &\triangleq \text{NV_Literal}(\text{L_String}(s)) \\
\text{Bitvector}(v) &\triangleq \text{NV_Literal}(\text{L_Bitvector}(v))
\end{aligned}$$

We define the following types of native values:

$$\begin{aligned}
\mathcal{Z} &\triangleq \{\text{Int}(z) \mid z \in \mathbb{Z}\} \\
\mathcal{B} &\triangleq \{\text{Bool}(\text{TRUE}), \text{Bool}(\text{FALSE})\} \\
\mathcal{R} &\triangleq \{\text{Real}(r) \mid r \in \mathbb{Q}\} \\
\text{STR} &\triangleq \{\text{String}(s) \mid s \in \mathbb{S}\} \\
\mathcal{BV} &\triangleq \{\text{Bitvector}(bits) \mid bits \in \{0, 1\}^*\} \\
\mathcal{VEC} &\triangleq \{\text{NV_Vector}(vals) \mid vals \in \mathbb{V}^*\} \\
\mathcal{REC} &\triangleq \{\text{NV_Record}(field_map) \mid field_map \in \mathbb{I} \rightarrow_{\text{fin}} \mathbb{V}\}
\end{aligned}$$

6.2 Dynamic Domain of a Type

We now define the concept of a *dynamic domain* of a type and the *static domain* of a type. Intuitively, domains assign potentially infinite sets of native values to types. Dynamic domains are used by the semantics to evaluate expressions of the form `UNKNOWN: t` by choosing a single value from the dynamic domain of `t`. Static domains are used to define (domain) subtype satisfaction in Section 8.3.

The definition of a dynamic domain refers to *dynamic environments*, denoted \mathbb{DE} , which assigns native values to identifiers [2].

We define *environments* as pairs of static environments and dynamic environments: $\mathbb{E} \triangleq \mathbb{SE} \times \mathbb{DE}$.

Formally, the partial function

$$\text{dyn_dom} : \overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\text{ty}}^{\text{t}} \rightarrow \overbrace{\mathcal{P}(\mathbb{V})}^{\text{d}}$$

assigns the set of values that a type `t` can hold in a given environment `env`. We say that $\text{dyn_dom}(\text{env}, \text{t})$ is the *dynamic domain* of `t` in the environment `env`. The *static domain* of a type is the set of values which storage elements of that type may hold across all possible dynamic environments. The reason for this distinction is that the sets of values of integer types, bitvector types, and array types can depend on the dynamic values of variables.

Types that do not refer to variables whose values are only known dynamically have a static domain that is equal to any of their dynamic domains. In those cases, we simply refer to their *domain*.

Associating a set of values to a type is done by evaluating any expression appearing in the type definitions. Evaluation is defined by the ASL semantics [2] via the relation

$$\text{eval_expr_sef}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{expr}}^{\text{e}}) \times \text{Normal}(\overbrace{\mathbb{V}}^{\text{v}}, \overbrace{\mathcal{G}}^{\text{g}}) \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

which evaluates side-effect-free expressions and either returns a configuration of the form $\text{Normal}(\text{v}, \text{g})$ or a dynamic error configuration \#DE . In the first case, v is a native value and g is an *execution graph*. Execution graphs are related to the concurrent semantics and can be ignored in the context of defining dynamic domains. In the latter case (which can occur if, for example, an expression attempts to divide 8 by 0), a dynamic error configuration, for which we use the notation \#DE , is returned. The dynamic domain is empty in cases where evaluating side-effect-free expressions results in a dynamic error. The dynamic domain is undefined if the type t is not well-typed in tenv . That is, if $\text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{\#TE}$.

As part of the definition, we also associate dynamic domains to integer constraints by overloading dyn_dom :

$$\text{dyn_dom} : \overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\text{int_constraint}}^{\text{c}} \rightarrow \overbrace{\mathcal{P}(\mathbb{V})}^{\text{d}}$$

6.2.1 Prose

For an environment $\text{env} \in \mathbb{E}$ and a type t , the domain is d and one of the following applies:

- All of the following apply (T_BOOL):
 - * t is the Boolean type, T_Bool ;
 - * d is the set of native Boolean values, \mathcal{B} .
- All of the following apply (T_STRING):
 - * t is the string type, T_String ;
 - * d is the set of all native string values, $\text{\textit{STR}}$.
- All of the following apply (T_REAL):
 - * t is the real type, T_Real ;
 - * d is the set of all native real values, \mathcal{R} .
- All of the following apply (T_ENUMERATION):
 - * t is the enumeration type with labels $\text{id}_{1..k}$, that is $\text{T_Enum}(\text{id}_{1..k})$;

- * d is the set of all native integer values for $1..k$.

Why represent enumeration domains via integers: Conceptually, enumeration labels carry two pieces of information — the identifiers themselves and their position in the list of identifiers, which are used for accessing arrays. For the purpose of type-checking, we use the identifiers, but for the purpose of the semantics and the domain of values, only the positions are relevant.

- All of the following apply (`T_INT_UNCONSTRAINED`):
 - * t is the unconstrained integer type, `unconstrained_integer`;
 - * d is the set of all native integer values, \mathbb{Z} .
- All of the following apply (`T_INT_WELL_CONSTRAINED`):
 - * t is the well-constrained integer type `T_Int(WellConstrained(c1..k))`;
 - * d is the union of the dynamic domains of each of the constraints $c_{1..k}$ in `env`.
- All of the following apply (`CONSTRAINT_EXACT_OKAY`):
 - * c is a constraint consisting of a single side-effect-free expression e , that is, `Constraint_Exact(e)`;
 - * evaluating e in `env`, results in a configuration with the native integer for n ;
 - * d is the set containing the single native integer value for n .
- All of the following apply (`CONSTRAINT_EXACT_DYNAMIC_ERROR`):
 - * c is a constraint consisting of a single side-effect-free expression e , that is, `Constraint_Exact(e)`;
 - * evaluating e in `env`, results in a dynamic error configuration;
 - * d is the empty set.
- All of the following apply (`CONSTRAINT_RANGE_OKAY`):
 - * c is a range constraint consisting of a two side-effect-free expressions $e1$ and $e2$, that is, `Constraint.Range(e1, e2)`;
 - * evaluating $e1$ in `env`, results in a configuration with the native integer for a ;
 - * evaluating $e2$ in `env`, results in a configuration with the native integer for b ;
 - * d is the set containing all native integer values for integers greater or equal to a and less than or equal to b .
- All of the following apply (`CONSTRAINT_RANGE_DYNAMIC_ERROR1`):
 - * c is a range constraint consisting of a two side-effect-free expressions $e1$ and $e2$, that is, `Constraint.Range(e1, e2)`;
 - * evaluating $e1$ in `env`, results in a dynamic error configuration;
 - * d is the empty set.

- All of the following apply (CONSTRAINT_RANGE_DYNAMIC_ERROR2):
 - * c is a range constraint consisting of a two side-effect-free expressions $e1$ and $e2$, that is, `Constraint.Range($e1$, $e2$)`;
 - * evaluating $e1$ in env , results in a configuration with the native integer for a ;
 - * evaluating $e2$ in env , results in a dynamic error configuration;
 - * d is the empty set.
- All of the following apply (T_INT_PARAMETERIZED):
 - * t is a [parameterized integer type](#) for parameter id , `T_Int(Parameterized(id))`;
 - * the native value associated with id in the local dynamic environment is the native integer value for n ;
 - * d is the set containing the single integer value for n .
- All of the following apply (T_BITS_DYNAMIC_ERROR):
 - * t is a bitvector type with size expression e , `T_Bits(e , _)`;
 - * evaluating e in env , results in a dynamic error configuration;
 - * d is the empty set.
- All of the following apply (T_BITS_NEGATIVE_WIDTH_ERROR):
 - * t is a bitvector type with size expression e , `T_Bits(e , _)`;
 - * evaluating e in env , results in a configuration with the native integer for k ;
 - * k is negative;
 - * d is the empty set.
- All of the following apply (T_BITS_EMPTY):
 - * t is a bitvector type with size expression e , `T_Bits(e , _)`;
 - * evaluating e in env , results in a configuration with the native integer for 0;
 - * d is the set containing the single native value for an empty bitvector.
- All of the following apply (T_BITS_NON_EMPTY):
 - * t is a bitvector type with size expression e , `T_Bits(e , _)`;
 - * evaluating e in env , results in a configuration with the native integer for k ;
 - * k is greater than 0;
 - * d is the set containing all native values for bitvectors of size exactly k .
- All of the following apply (T_TUPLE):
 - * t is a tuple type over types t_i , for $i = 1..k$, `T_Tuple($t_{1..k}$)`;

- * the domain of each element \mathbf{t}_i is D_i , for $i = 1..k$;
 - * evaluating \mathbf{e} in \mathbf{env} , results in a configuration with the native integer for k ;
 - * \mathbf{d} is the set containing all native vectors of k values, where the value at position i is from D_i .
- All of the following apply (T_ARRAY_DYNAMIC_ERROR):
 - * \mathbf{t} is an array type with length expression \mathbf{e} and element type \mathbf{t}_i , for $i = 1..k$, $\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})$;
 - * evaluating \mathbf{e} in \mathbf{env} , results in a dynamic error configuration;
 - * \mathbf{d} is the empty set.
 - All of the following apply (T_ARRAY_NEGATIVE_LENGTH_ERROR):
 - * \mathbf{t} is an array type with length expression \mathbf{e} and element type \mathbf{t}_i , for $i = 1..k$, $\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})$;
 - * evaluating \mathbf{e} in \mathbf{env} , results in a configuration with the native integer for k ;
 - * k is negative;
 - * \mathbf{d} is the empty set.
 - All of the following apply (T_ARRAY_OKAY):
 - * \mathbf{t} is an array type with length expression \mathbf{e} and element type \mathbf{t}_i , for $i = 1..k$, $\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})$;
 - * evaluating \mathbf{e} in \mathbf{env} , results in a configuration with the native integer for k ;
 - * k is greater than or equal to 0;
 - * the domain of $\mathbf{t1}$ is $D_{\mathbf{t1}}$;
 - * \mathbf{d} is the set containing all native vectors of k values taken from $D_{\mathbf{t1}}$.
 - All of the following apply (T_STRUCTURED):
 - * \mathbf{t} is a **structured type** with typed fields $(\mathbf{id}_i, \mathbf{t}_i)$, for $i = 1..k$, that is $L([i = 1..k : (\mathbf{id}_i, \mathbf{t}_i)])$ where $L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\}$;
 - * the domain of each type \mathbf{t}_i is D_i , for $i = 1..k$;
 - * \mathbf{d} is the set containing all native records where \mathbf{id}_i is mapped to a value taken from D_i .
 - All of the following apply (T_NAMED):
 - * \mathbf{t} is a named type with name \mathbf{id} , $\mathbf{T_Named}(\mathbf{id})$;
 - * the type associated with \mathbf{id} in \mathbf{tenv} is \mathbf{ty} ;
 - * \mathbf{d} is the domain of \mathbf{ty} in \mathbf{env} .

6.2.2 Formally

$$\begin{array}{c}
\text{T_BOOL} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bool}}^t) = \overbrace{\mathcal{B}}^d \\
\\
\text{T_STRING} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{T_String}}^t) = \overbrace{STR}^d \\
\\
\text{T_REAL} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{T_Real}}^t) = \overbrace{\mathcal{R}}^d \\
\\
\text{T_ENUMERATION} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{T_Enum}(\text{id}_{1..k})}^t) = \overbrace{\{\text{Int}(1), \dots, \text{Int}(k)\}}^d \\
\\
\text{T_INT_UNCONSTRAINED} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{unconstrained_integer}}^t) = \overbrace{\mathcal{Z}}^d \\
\\
\text{T_INT_WELL_CONSTRAINED} \\
\text{dyn_dom}(\text{env}, \overbrace{\text{T_Int}(\text{WellConstrained}(c_{1..k}))}^t) = \bigcup_{i=1}^k \overbrace{\text{dyn_dom}(\text{env}, c_i)}^d \\
\\
\text{CONSTRAINT_EXACT_OKAY} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(n), _)}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Exact}(e)}^c) = \overbrace{\{\text{Int}(n)\}}^d} \\
\\
\text{CONSTRAINT_EXACT_DYNAMIC_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Exact}(e)}^c) = \overbrace{\emptyset}^d} \\
\\
\text{CONSTRAINT_RANGE_OKAY} \\
\frac{\begin{array}{l} \text{eval_expr_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(a), _) \\ \text{eval_expr_sef}(\text{env}, e2) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(b), _) \end{array}}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\{\text{Int}(n) \mid a \leq n \wedge n \leq b\}}^d} \\
\\
\text{CONSTRAINT_RANGE_DYNAMIC_ERROR1} \\
\frac{\text{eval_expr_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\emptyset}^d} \\
\\
\text{CONSTRAINT_RANGE_DYNAMIC_ERROR2} \\
\frac{\text{eval_expr_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(_, _) \quad \text{eval_expr_sef}(\text{env}, e2) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\emptyset}^d}
\end{array}$$

The notation $L^{\text{denv}}(\text{id})$ denotes the native value associated with the identifier id in the *local dynamic environment* of denv .

$$\frac{\text{T_INT_PARAMETERIZED} \quad L^{\text{denv}}(\text{id}) = \text{Int}(n)}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{id}))}^{\text{t}}) = \overbrace{\{\text{Int}(n)\}}^{\text{d}}}$$

$$\frac{\text{T_BITS_DYNAMIC_ERROR} \quad \text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(\text{e}, _)}^{\text{t}}) = \overbrace{\emptyset}^{\text{d}}}$$

$$\frac{\text{T_BITS_NEGATIVE_WIDTH_ERROR} \quad \text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k < 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(\text{e}, _)}^{\text{t}}) = \overbrace{\emptyset}^{\text{d}}}$$

$$\frac{\text{T_BITS_EMPTY} \quad \text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(0), _)}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(\text{e}, _)}^{\text{t}}) = \overbrace{\{\text{Bitvector}([\])\}}^{\text{d}}}$$

$$\frac{\text{T_BITS_NON_EMPTY} \quad \text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k > 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(\text{e}, _)}^{\text{t}}) = \overbrace{\{\text{Bitvector}(\text{b}_{1..k}) \mid \text{b}_1, \dots, \text{b}_k \in \{0, 1\}\}}^{\text{d}}}$$

$$\frac{\text{T_TUPLE} \quad i = 1..k : \text{dyn_dom}(\text{env}, \text{t}_i) = D_i}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Tuple}(\text{t}_{1..k})}^{\text{t}}) = \overbrace{\{\text{NV_Vector}(\text{v}_{1..k}) \mid \text{v}_i \in D_i\}}^{\text{d}}}$$

$$\begin{array}{c}
\text{T_ARRAY_DYNAMIC_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(e, t_1)}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_ARRAY_NEGATIVE_LENGTH_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k < 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(e, t_1)}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_ARRAY_OKAY} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k \geq 0 \quad \text{dyn_dom}(\text{env}, t_1) = D_{t_1}}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(e, t_1)}^t) = \overbrace{\{\text{NV_Vector}(v_{1..k}) \mid v_{1..k} \in D_{t_1}\}}^d} \\
\\
\text{STRUCTURED} \\
\frac{L \in \{\text{T_Record}, \text{T_Exception}\} \quad i = 1..k : \text{dyn_dom}(\text{env}, t_i) = D_i}{\text{dyn_dom}(\text{env}, \overbrace{L([i = 1..k : (\text{id}_i, t_i)])}^t) = \overbrace{\{\text{NV_Record}(\{i = 1..k : \text{id}_i \mapsto v_i\}) \mid v_i \in D_i\}}^d} \\
\\
\text{T_NAMED} \\
\frac{G^{\text{env}}.\text{declared_types}(\text{id}) = \text{ty}}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Named}(\text{id})}^t) = \overbrace{\text{dyn_dom}(\text{env}, \text{ty})}^d}
\end{array}$$

6.2.3 Example

The domain of `integer` is the infinite set of all integers.

The domain of `integer {2,16}` is the set $\{\text{Int}(2), \text{Int}(16)\}$.

The domain of `integer{1..3}` is the set $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$.

The domain of `integer{10..1}` is the empty set as there are no integers that are both greater than 10 and smaller than 1.

The domain of `bits(2)` is the set $\{\text{Bitvector}(00), \text{Bitvector}(01), \text{Bitvector}(10), \text{Bitvector}(11)\}$.

The domain of `enumeration {GREEN, ORANGE, RED}` is the set $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$ and so is the domain of type `TrafficLights` of `enumeration {GREEN, ORANGE, RED}`.

The domain of `bits(2,16)` is the set containing native bitvectors of all 2-bit and all 16-bit binary sequences.

The domain of `(integer, integer)` is the set containing all pairs of native integer values.

The domain of `record {a: integer; b: boolean}` contains all native records that map `a` to a native integer value and `b` to a native Boolean value.

The dynamic domain of a subprogram parameter `N: integer` is the (singleton) set containing the native integer value `c`, which is assigned to `N` by a given dynamic environment. The static domain of that parameter is the infinite set of all native integer values.

6.3 Subsumption Testing

Whether an assignment statement is well-typed depends on whether the dynamic domain of the right hand side type is contained in the dynamic domain of the left hand side type, for any given dynamic environment (see Section 8.3 where this is checked).

Definition 27 (Subsumption) *For any given types t and s and static environment $tenv$, we say that t subsumes s in $tenv$, if the following condition holds:*

$$subsumes(tenv, t, s) \triangleq \forall denv \in \mathbb{DE}. \text{dyn_dom}((tenv, denv), t) \supseteq \text{dyn_dom}((tenv, denv), s) . \quad (6.1)$$

For example, consider the assignment

```
var x : integer{1,2,3} = UNKNOWN : integer{1,2};
```

It is legal, since (in any static environment), the domain of `integer{1,2,3}` is $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$, which subsumes the domain of `integer{1,2}`, which is $\{\text{Int}(1), \text{Int}(2)\}$.

Since dynamic domains are potentially infinite, this requires *symbolic reasoning*. Furthermore, since any (statically evaluable) expressions may appear inside integer and bitvector types, testing subsumption is undecidable. We therefore approximate subsumption testing *conservatively* via the predicate `sym_subsumes(tenv, t, s)`.

Definition 28 (Sound Subsumption Test) *A predicate*

$$sym_subsumes(\overbrace{\text{SE}}^{tenv}, \overbrace{ty}^t, \overbrace{ty}^s) \longrightarrow \mathbb{B}$$

is sound if the following condition holds:

$$\forall t, s \in ty. \text{tenv} \in \text{SE}. \quad sym_subsumes(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE} \implies subsumes(tenv, t, s) . \quad (6.2)$$

That is, if a sound subsumption test returns a positive answer, it means that `t` definitely *subsumes* `s` in the static environment `tenv`. This is referred to as a *true positive*. However, a negative answer means one of two things:

True Negative: indeed, t does not subsume s in the static environment tenv ; or

False Negative: the symbolic reasoning is unable to decide.

In other words, `sym_subsumes(tenv, t, s)` errs on the *safe side* — it never answers **TRUE** when the real answer is **FALSE**, which would (undesirably) determine the following statement as well-typed:

```
var x : integer{1,2} = UNKNOWN: integer;
```

A sound but trivial subsumption test is one that always returns **FALSE**. However, that would make all assignments be considered as not well-typed. Indeed, it has the maximal set of false negatives. Reducing the set of false negatives requires stronger symbolic reasoning algorithms, which inevitably leads to higher computational complexity. The symbolic subsumption test in Chapter 24 attempts to accept a large enough set of true positives, based on empirical trial and error, while maintaining the computational complexity of the symbolic reasoning relatively low. In particular, it serves as the definitive subsumption test that must be utilized by any implementation of the ASL type system.

Chapter 7

Basic Type Attributes

This chapter defines some basic predicates for classifying types as well as functions that inspect the structure of types:

- Builtin singular types (Section 7.1)
- Builtin aggregate types (Section 7.2)
- Builtin types (Section 7.3)
- Named types (Section 7.4)
- Anonymous types (Section 7.5)
- Singular types (Section 7.6)
- Aggregate types (Section 7.7)
- Non-primitive types (Section 7.9)
- Primitive types (Section 7.10)
- The structure of a type (Section 7.11)
- The underlying type of a type (Section 7.12)
- Checked constrained integers (Section 7.13)

7.1 TypingRule.BuiltinSingularType

The predicate

$$\text{textfuncis_builtin_singular}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin singular type*.

7.1.1 Prose

The *builtin singular types* are:

- integer;
- real;
- string;
- boolean;
- bits (which also represents bit, as a special case);
- enumeration.

7.1.2 Example

In this example:

```
let i : integer = 0;
let r : real = 0.0;
let s : string = "0.0";
let b : boolean = TRUE;
let z4 : bits(4) = '0000';
let o2 : bits(2) = '11';
```

Variables of builtin singular types `integer`, `real`, `boolean`, `bits(4)`, and `bits(2)` are defined.

7.1.3 Example

```
type Color of enumeration { RED, BLACK } ;
```

```
func main () => integer
begin
  assert (RED != BLACK);
  return 0;
end
```

The builtin singular type `Color` consists in two constants `RED`, and `BLACK`.

7.1.4 Formally

$$\frac{b := \textit{ast_label}(\textit{ty}) \in \{\textit{T_Real}, \textit{T_String}, \textit{T_Bool}, \textit{T_Bits}, \textit{T_Enum}, \textit{T_Int}\}}{\textit{extfuncis_builtin_singular}(\textit{ty}) \xrightarrow{\textit{type}} b}$$

7.2 TypingRule.BuiltinAggregateType

The predicate

$$\text{textfuncis_builtin_aggregate}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin aggregate type*.

7.2.1 Prose

The builtin aggregate types are:

- tuple;
- array;
- record;
- exception.

7.2.2 Example

```

type Pair of (integer, boolean);

type T of array [3] of real;
type Coord of enumeration { CX, CY, CZ };
type PointArray of array [Coord] of real;

type PointRecord of record
  { x : real, y : real, z : real };

func main () => integer
begin
  let p = (0, FALSE);

  var t1 : T; var t2 : PointArray;
  t1[0] = t2[CX];

  let o = PointRecord { x=0.0, y=0.0, z=0.0 };
  t2[CZ] = o.z;

  return 0;
end

```

Type `Pair` is the type of integer and boolean pairs.

Arrays are declared with indices that are either integer-typed or enumeration-typed. In the example above, `T` is declared as an array with an integer-typed index (as indicated by the used of the integer-typed constant 3) whereas `PointArray` is declared with the index of `Coord`, which is an enumeration type.

Arrays declared with integer-typed indices can be accessed only by integers ranging from 0 to the size of the array minus 1. In the example above, `T` can be accessed with one of 0, 1, and 2.

Arrays declared with an enumeration-typed index can only be accessed with labels from the corresponding enumeration. In the example above, `PointArray` can only be accessed with one of the labels `CX`, `CY`, and `CZ`.

The (builtin aggregate) type `{ x : real, y : real, z : real }` is a record type with three fields `x`, `y` and `z`.

7.2.3 Example

```
type Not_found of exception;
type SyntaxException of exception { message:string };

func main () => integer
begin
  if UNKNOWN : boolean then
    throw Not_found {};
  else
    throw SyntaxException { message="syntax" };
  end

  return 0;
end
```

Two (builtin aggregate) exception types are defined:

- `exception{}` (for `Not_found`), which carries no value; and
- `exception { message:string }` (for `SyntaxException`), which carries a message.

Notice the similarity with record types and that the empty field list `{}` can be omitted in type declarations, as is the case for `Not_found`.

7.2.4 Formally

$$\frac{b := \text{ast_label}(\text{ty}) \in \{\text{T_Tuple}, \text{T_Array}, \text{T_Record}, \text{T_Exception}\}}{\text{extfuncis_builtin_aggregate}(\text{ty}) \xrightarrow{\text{type}} b}$$

7.3 TypingRule.BuiltinSingularOrAggregate

The predicate

$$\text{is_builtin}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin type*.

7.3.1 Prose

`ty` is a builtin type and one of the following applies:

- `ty` is singular;
- `ty` is builtin aggregate.

7.3.2 Example

In the specification

```
type ticks of integer;
```

the type `integer` is a builtin type but the type of `ticks` is not.

7.3.3 Formally

$$\frac{\text{extfuncis_builtin_singular}(\text{ty}) \xrightarrow{\text{type}} \text{b1}}{\text{is_builtin}(\text{ty}) \xrightarrow{\text{type}} \text{b1} \vee \text{b2}}$$

7.4 TypingRule.NamedType

The predicate

$$\text{is_named}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *named type*.

7.4.1 Prose

A named type is a type that is declared by using the `type of` syntax.

7.4.2 Example

In the specification

```
type ticks of integer;
```

`ticks` is a named type.

7.4.3 Formally

$$\frac{\text{b} := \text{ast_label}(\text{ty}) = \text{T.Named}}{\text{is_named}(\text{ty}) \xrightarrow{\text{type}} \text{b}}$$

7.5 TypingRule.AnonymousType

The predicate

$$\text{is_anonymous}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type ty is an *anonymous type*.

7.5.1 Prose

An anonymous type is a type that is not declared using the type syntax.

7.5.2 Example

The tuple type $(\text{integer}, \text{integer})$ is an anonymous type.

7.5.3 Formally

$$\frac{b := \text{ast_label}(\text{ty}) \neq \text{T_Named}}{\text{is_anonymous}(\text{ty}) \xrightarrow{\text{type}} b}$$

7.6 TypingRule.SingularType

The predicate

$$\text{is_singular}(\overbrace{\text{SIE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether the type ty is a *singular type* in the static environment tenv .

7.6.1 Prose

A type ty is singular if and only if all of the following apply:

- obtaining the *underlying type* of ty in the environment tenv yields $\text{t1} \#TE$;
- t1 is a builtin singular type.

7.6.2 Example

In the following example, the types A, B, and C are all singular types:

```
type A of integer;
type B of A;
type C of B;
```

7.6.3 Formally

$$\frac{\begin{array}{c} \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t1} \text{ // } \#TE \\ \text{extfuncis_builtin_singular}(\text{t1}) \xrightarrow{\text{type}} \text{b} \end{array}}{\text{is_singular}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{b}}$$

7.7 TypingRule.AggregateType

The predicate

$$\text{is_aggregate}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether the type `ty` is an *aggregate type* in the static environment `tenv`.

7.7.1 Prose

A type `ty` is aggregate in an environment `tenv` if and only if all of the following apply:

- obtaining the *underlying type* of `ty` in the environment `tenv` yields `t1` // `#TE`;
- `t1` is a builtin aggregate.

7.7.2 Example

In the following example, the types A, B, and C are all aggregate types:

```
type A of (integer, integer);
type B of A;
type C of B;
```

7.7.3 Formally

$$\frac{\begin{array}{c} \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t1} \text{ // } \#TE \\ \text{extfuncis_builtin_aggregate}(\text{t1}) \xrightarrow{\text{type}} \text{b} \end{array}}{\text{is_aggregate}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{b}}$$

7.8 TypingRule.StructuredType

A *structured type* is any type that consists of a list of field identifiers that denote individual storage elements. In ASL there are two such types — record types and exception types.

The predicate

$$\text{is_structured}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether the type `ty` is a *structured type* and yields the result in `b`.

7.8.1 Prose

The result b is **TRUE** if and only if ty is either a record type or an exception type, which is determined via the AST label of ty .

7.8.2 Example

In the following example, the types `SyntaxException` and `PointRecord` are each an example of a **structured type**:

```
type SyntaxException of exception {message: string };
type PointRecord of Record {x : real, y: real, z: real};
```

7.8.3 Formally

$$is_structured(ty) \xrightarrow{\text{type}} \overbrace{ast_label(ty) \in \{T_Record, T_Exception\}}^b$$

7.9 TypingRule.NonPrimitiveType

The predicate

$$is_non_primitive(\overbrace{ty}^{ty}) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether the type ty is a *non-primitive type*.

7.9.1 Prose

One of the following applies:

- All of the following apply (SINGULAR):
 - * ty is a builtin singular type;
 - * b is **FALSE**.
- All of the following apply (NAMED):
 - * ty is a named type;
 - * b is **TRUE**.
- All of the following apply (TUPLE):
 - * ty is a tuple type li ;
 - * b is **TRUE** if and only if there exists a non-primitive type in li .
- All of the following apply (ARRAY):
 - * ty is an array of type ty'

- * b is **TRUE** if and only if ty' is non-primitive.
- All of the following apply (STRUCTURED):
 - * ty is a **structured type** with fields **fields**;
 - * b is **TRUE** if and only if there exists a non-primitive type in **fields**.

7.9.2 Example

The following types are non-primitive:

Type definition	Reason for being non-primitive
type A of integer	Named types are non-primitive
(integer, A)	The second component, A, has non-primitive type
array[6] of A	Element type A has a non-primitive type
record { a : A }	The field a has a non-primitive type

7.9.3 Formally

The cases TUPLE and STRUCTURED below, use the notation b_t to name Boolean variables by using the types denoted by t as a subscript.

$$\begin{array}{c}
 \text{SINGULAR} \\
 \hline
 \text{ast_label}(ty) \in \{T_Real, T_String, T_Bool, T_Bits, T_Enum, T_Int\} \\
 \hline
 is_non_primitive(ty) \xrightarrow{\text{type}} \text{FALSE} \\
 \\
 \text{NAMED} \\
 \hline
 \text{ast_label}(ty) = T_Named \\
 \hline
 is_non_primitive(ty) \xrightarrow{\text{type}} \text{TRUE} \\
 \\
 \text{TUPLE} \\
 \hline
 t \in \text{tys} : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in \text{tys}} b_t \\
 \hline
 is_non_primitive(\overbrace{T_Tuple(\text{tys})}^{ty}) \xrightarrow{\text{type}} b \\
 \\
 \text{ARRAY} \\
 \hline
 is_non_primitive(ty') \xrightarrow{\text{type}} b \\
 \hline
 is_non_primitive(\overbrace{T_Array(_, ty')}^{ty}) \xrightarrow{\text{type}} b \\
 \\
 \text{STRUCTURED} \\
 \hline
 L \in \{T_Record, T_Exception\} \\
 (_, t) \in \text{fields} : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in li} b_t \\
 \hline
 is_non_primitive(\overbrace{L(\text{fields})}^{ty}) \xrightarrow{\text{type}} b
 \end{array}$$

7.10 TypingRule.PrimitiveType

The predicate

$$\text{is_primitive}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type ty is a *primitive type*.

7.10.1 Prose

A type ty is primitive if it is not non-primitive.

7.10.2 Example

The following types are primitive:

Type definition	Reason for being primitive
<code>integer</code>	Integers are primitive
<code>(integer, integer)</code>	All tuple elements are primitive
<code>array[5] of integer</code>	The array element type is primitive
<code>record {ticks : integer}</code>	The single field <code>ticks</code> has a primitive type

7.10.3 Formally

$$\frac{\text{is_non_primitive}(\text{ty}) \xrightarrow{\text{type}} \text{b}}{\text{is_primitive}(\text{ty}) \xrightarrow{\text{type}} \neg \text{b}}$$

7.11 TypingRule.Structure

The function

$$\text{get_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

assigns a type to its *structure*, which is the type formed by recursively replacing named types by their type definition in the static environment `tenv`. If a named type is not associated with a declared type in `tenv`, a type error is returned.

`TypingRule.Specification` ensures the absence of circular type definitions, which ensures that `TypingRule.Structure` terminates¹.

7.11.1 Prose

One of the following applies:

- All of the following apply (NAMED):

* ty is a named type x ;

¹In mathematical terms, this ensures that `TypingRule.Structure` is a proper *structural induction*.

- * obtaining the declared type associated with x in the static environment tenv yields $\mathbf{t1}$ *//TE*;
- * obtaining the structure of $\mathbf{t1}$ static environment tenv yields \mathbf{t} *//TE*;
- All of the following apply (BUILTIN_SINGULAR):
 - * \mathbf{ty} is a builtin singular type;
 - * \mathbf{t} is \mathbf{ty} .
- All of the following apply (TUPLE):
 - * \mathbf{ty} is a tuple type with list of types \mathbf{tys} ;
 - * the types in \mathbf{tys} are indexed as \mathbf{t}_i , for $i = 1..k$;
 - * obtaining the structure of each type \mathbf{t}_i , for $i = 1..k$, in \mathbf{tys} in the static environment tenv , yields \mathbf{t}'_i *//TE*;
 - * \mathbf{t} is a tuple type with the list of types \mathbf{t}'_i , for $i = 1..k$.
- All of the following apply (ARRAY):
 - * \mathbf{ty} is an array type of length \mathbf{e} with element type \mathbf{t} ;
 - * obtaining the structure of \mathbf{t} yields $\mathbf{t1}$ *//TE*;
 - * \mathbf{t} is is an array type with of length \mathbf{e} with element type $\mathbf{t1}$.
- All of the following apply (STRUCTURED):
 - * \mathbf{ty} is a *structured type* with fields \mathbf{fields} ;
 - * obtaining the structure for each type \mathbf{t} associated with field \mathbf{id} yields a type $\mathbf{t}_{\mathbf{id}}$ *//TE*;
 - * \mathbf{t} is a record or an exception, in correspondence to \mathbf{ty} , with the list of pairs $(\mathbf{id}, \mathbf{t}_{\mathbf{id}})$;

7.11.2 Example

In this example: `type T1 of integer;` is the named type T1 whose structure is `integer`.

In this example: `type T2 of (integer, T1);` is the named type T2 whose structure is `(integer, integer)`. In this example, `(integer, T1)` is non-primitive since it uses T1, which is builtin aggregate.

In this example: `var x: T1;` the type of x is the named (hence non-primitive) type T1, whose structure is `integer`.

In this example: `var y: integer;` the type of y is the anonymous primitive type `integer`.

In this example: `var z: (integer, T1);` the type of z is the anonymous non-primitive type `(integer, T1)` whose structure is `(integer, integer)`.

7.11.3 Formally

NAMED

$$\frac{\text{declared_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \text{ // } \#TE \quad \text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t \text{ // } \#TE}{\text{get_structure}(\text{tenv}, T_Named(x)) \xrightarrow{\text{type}} t} \quad \text{BUILTIN_SINGULAR} \quad \frac{\text{extfuncis_builtin_singular}(ty) \xrightarrow{\text{type}} \text{TRUE}}{\text{get_structure}(\text{tenv}, ty) \xrightarrow{\text{type}} ty}$$

TUPLE

$$\frac{\text{tys} \stackrel{\text{is}}{=} t_{1..k} \quad i = 1..k : \text{get_structure}(\text{tenv}, t_i) \xrightarrow{\text{type}} t'_i \text{ // } \#TE}{\text{get_structure}(\text{tenv}, T_Tuple(\text{tys})) \xrightarrow{\text{type}} T_Tuple(i = 1..k : t'_i)}$$

ARRAY

$$\frac{\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \text{ // } \#TE}{\text{get_structure}(\text{tenv}, T_Array(e, t)) \xrightarrow{\text{type}} T_Array(e, t1)}$$

STRUCTURED

$$\frac{L \in \{T_Record, T_Exception\} \quad (id, t) \in \text{fields} : \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_{id} \text{ // } \#TE}{\text{get_structure}(\text{tenv}, L(\text{fields})) \xrightarrow{\text{type}} L([(id, t) \in \text{fields} : (id, t_{id})])}$$

7.12 TypingRule.Anonymize

The function

$$\text{make_anonymous}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^{\text{ty}}) \longrightarrow \overbrace{ty}^t \cup \overbrace{T_TypeError}^{\#TE}$$

returns the *underlying type* — t — of the type ty in the static environment tenv or a type error. Intuitively, ty is the first non-named type that is used to define ty . Unlike *get_structure*, *make_anonymous* replaces named types by their definition until the first non-named type is found but does not recurse further.

7.12.1 Prose

One of the following applies:

- All of the following apply (NAMED):
 - * ty is a named type x ;
 - * obtaining the type declared for x yields $t1 \text{ // } \#TE$;
 - * the *underlying type* of $t1$ is t .
- All of the following apply (NON-NAMED):
 - * ty is not a named type x ;
 - * t is ty .

7.12.2 Example

Consider the following example:

```
type T1 of integer;
type T2 of T1;
type T3 of (integer, T2);
```

The underlying types of `integer`, `T1`, and `T2` is `integer`.

The underlying type of `(integer, T2)` and `T3` is `(integer, T2)`. Notice how the underlying type does not replace `T2` with its own underlying type, in contrast to the structure of `T2`, which is `(integer, integer)`.

7.12.3 Formally

$$\begin{array}{c}
 \text{NAMED} \\
 \text{ty} \stackrel{\text{is}}{=} \text{T_Named}(x) \quad \text{declared_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \quad \text{// } \#TE \\
 \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t \\
 \hline
 \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} t \\
 \\
 \text{NON-NAMED} \\
 \text{ast_label}(\text{ty}) \neq \text{T_Named} \\
 \hline
 \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty}
 \end{array}$$

7.13 TypingRule.CheckConstrainedInteger

The function

$$\text{check_constrained_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the type `t` is a `constrained integer`. If so, the result is `TRUE`, otherwise a type error is returned.

7.13.1 Prose

One of the following applies:

- All of the following apply (WELL-CONSTRAINED):
 - * `t` is a well-constrained integer;
 - * the result is `TRUE`.
- All of the following apply (PARAMETERIZED):
 - * `t` is a `parameterized integer type`;
 - * the result is `TRUE`.

- All of the following apply (UNCONSTRAINED):
 - * t is an unconstrained integer;
 - * the result is a type error indicating that a constrained integer type is expected.
- All of the following apply (CONFLICTING_TYPE):
 - * t is not an integer type;
 - * the result is a type error indicating the type conflict.

7.13.2 Formally

WELL-CONSTRAINED

$check_constrained_integer(tenv, T_Int(WellConstrained(_))) \xrightarrow{type} TRUE$

PARAMETERIZED

$check_constrained_integer(tenv, T_Int(Parameterized(_))) \xrightarrow{type} TRUE$

UNCONSTRAINED

$check_constrained_integer(tenv, T_Int(Unconstrained(_))) \xrightarrow{type} \\ TypeError(ConstrainedIntegerExpected)$

CONFLICTING_TYPE

$\frac{ast_label(t) \neq T_Int}{check_constrained_integer(tenv, t) \xrightarrow{type} TypeError(TypeConflict)}$

Chapter 8

Relations Over Types

We define the following relations over types and operators:

- Subtype (Section [8.1](#))
- Structural Subtype Satisfaction (Section [8.2](#))
- Domain Subtype Satisfaction (Section [8.3](#))
- Subtype Satisfaction (Section [8.4](#))
- Type Satisfaction (Section [8.5](#))
- Type Clash (Section [8.6](#))
- Lowest Common Ancestor (Section [8.7](#))
- Checking adequacy of a unary operator for a type (Section [8.8](#))
- Checking adequacy of a binary operator for a pair of types (Section [8.9](#))

Finally, we define the following helper functions:

- `TypingRule.AnnotateConstraintBinop` (see Section [8.11](#))
- `TypingRule.BinopFilterRhs` (see Section [8.12](#))
- `TypingRule.RefineConstraintBySign` (see Section [8.13](#))
- `TypingRule.ReduceToZOpt` (see Section [8.14](#))
- `TypingRule.RefineConstraints` (see Section [8.15](#))
- `TypingRule.FilterReduceConstraintDiv` (see Section [8.16](#))
- `TypingRule.GetLiteralDivOpt` (see Section [8.17](#))
- `TypingRule.ExplodeIntervals` (see Section [8.18](#))

- `TypingRule.ExplodeConstraint` (see Section 8.19)
- `TypingRule.IntervalTooLarge` (see Section 8.20)
- `TypingRule.BinopIsExploding` (see Section 8.21)

We also define the helper rule `TypingRule.FindNamedLCA` (Section 8.10).

8.1 TypingRule.Subtype

The *subtype* relation is a partial order over named types. The *supertype* is the inverse relation. That is, `ty` is a supertype of `sy` if and only if `sy` is a subtype of `ty`.

The predicate

$$\text{is_subtype}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

8.1.1 Prose

One of the following applies:

- all of the following apply (REFLEXIVE):
 - * `t1` and `t2` are both the same named type;
 - * `b` is `TRUE`.
- all of the following apply (TRANSITIVE):
 - * `t1` is a named type with name `id1`, that is `T_Named(id1)`;
 - * `t2` is a named type with name `id2`, that is `T_Named(id2)`, such that `id1` is different from `id2`;
 - * the global static environment maintains that `id1` is a subtype of `id3`;
 - * testing whether the type named `id3` is a subtype of `t2` in the static environment `tenv` gives `b`.
- all of the following apply (NO_SUPERTYPE):
 - * `t1` is a named type with name `id1`, that is `T_Named(id1)`;
 - * `t2` is a named type with name `id2`, that is `T_Named(id2)`, such that `id1` is different from `id2`;
 - * the global static environment maintains that `id1` does subtype any named type;
 - * `b` is `FALSE`.
- all of the following apply (NOT_NAMED):
 - * at least one of `t1` and `t2` is not a named type;
 - * `b` is `FALSE`.

8.1.2 Example

In the following example `subInt` is a subtype of itself and of `superInt`:

```
type superInt of integer;
type subInt of integer subtypes superInt;
```

8.1.3 Formally

$$\begin{array}{c}
\text{REFLEXIVE} \\
\text{is_subtype}(\text{tenv}, T_Named(id), T_Named(id)) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{TRANSITIVE} \\
\frac{id1 \neq id2 \quad G^{\text{tenv}}.\text{subtypes}(id1) = id3 \quad \text{is_subtype}(\text{tenv}, T_Named(id3), t2) \xrightarrow{\text{type}} b}{\text{is_subtype}(\text{tenv}, T_Named(id1), T_Named(id2)) \xrightarrow{\text{type}} b} \\
\\
\text{NO_SUPERTYPE} \\
\frac{id1 \neq id2 \quad G^{\text{tenv}}.\text{subtypes}(id1) = \perp}{\text{is_subtype}(\text{tenv}, T_Named(id1), T_Named(id2)) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{NOT_NAMED} \\
\frac{(\text{ast_label}(t1) \neq T_Named \vee \text{ast_label}(t2) \neq T_Named)}{\text{is_subtype}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE}}
\end{array}$$

8.2 TypingRule.StructuralSubtypeSatisfaction

The predicate

$$\text{structural_subtype_satisfies}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether a type t *structurally-subtype-satisfies* a type s in environment tenv , returning the result b or a type error, if one is detected. The function assumes that both t and s are well-typed according to Chapter 9.

8.2.1 Prose

One of the following applies:

- All of the following apply (ERROR1):
 - * obtaining the [underlying type](#) of t gives a type error;
 - * the rule results in a type error.
- All of the following apply (ERROR2):
 - * obtaining the [underlying type](#) of t gives a type $t2$;

- * obtaining the **underlying type** of **s** gives a type error;
- * the rule results in a type error.
- All of the following apply (DIFFERENT_LABELS):
 - * the underlying types of **t** and **s** have different AST labels (for example, **integer** and **real**);
 - * **b** is **FALSE**.
- All of the following apply (SIMPLE):
 - * the **underlying type** of **t**, **t2**, is either **integer** (any kind), **real**, **string**, or **bool**;
 - * the **underlying type** of **s**, **s2**, is either **integer** (any kind), **real**, **string**, or **bool**;
 - * **b** is **TRUE** if and only if both **t2** and **s2** have the same ASL label.
- All of the following apply (T_ENUM):
 - * the **underlying type** of **t** is an enumeration type with list of labels **lis_t**, that is, **T_Enum(lis_t)**;
 - * the **underlying type** of **s** is an enumeration type with list of labels **lis_s**, that is, **T_Enum(lis_s)**;
 - * **b** is **TRUE** if and only if **lis_t** is equal to **lis_s**.
- All of the following apply (T_BITS):
 - * the **underlying type** of **s** is a bitvector type with width **w_s** and bit fields **bfs_s**, that is **T_Bits(w_s, bfs_s)**;
 - * the **underlying type** of **t** is a bitvector type with width **w_t** and bit fields **bfs_t**, that is **T_Bits(w_t, bfs_t)**;
 - * determining whether the bitwidth **w_s** is equivalent to **w_t** in **tenv** either yields **TRUE** or yields **FALSE**, which short-circuits the entire rule;
 - * determining whether the bit fields **bfs_s** are included in the bit fields **bfs_t** in **tenv** yields **b^{//TE}**
- All of the following apply (T_ARRAY_EXPR):
 - * **s** has the **underlying type** of an array with index **length_s** and element type **ty_s**, that is **T_Array(length_s, ty_s)**;
 - * **t** has the **underlying type** of an array with index **length_t** and element type **ty_t**, that is **T_Array(length_t, ty_t)**;
 - * determining whether **ty_s** and **ty_t** are equivalent in **tenv** is either **TRUE** or **FALSE**, which short-circuits the entire rule with **b = FALSE**;

- * either the AST labels of `length_s` and `length_t` are the same or the rule short-circuits with `b = FALSE`;
 - * `length_s` is an array length expression with `length_expr_s`, that is `ArrayLength.Expr(length_expr_s)`;
 - * `length_t` is an array length expression with `length_expr_t`, that is `ArrayLength.Expr(length_expr_t)`;
 - * determining whether expressions `length_expr_s` and `length_expr_t` are equivalent gives `b`.
- All of the following apply (`T_ARRAY_ENUM`):
 - * `s` has the **underlying type** of an array with index `length_s` and element type `ty_s`, that is `T_Array(length_s, ty_s)`;
 - * `t` has the **underlying type** of an array with index `length_t` and element type `ty_t`, that is `T_Array(length_t, ty_t)`;
 - * determining whether `ty_s` and `ty_t` are equivalent in `tenv` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b = FALSE`;
 - * either the AST labels of `length_s` and `length_t` are the same or the rule short-circuits with `b = FALSE`;
 - * `length_s` is an array with indices taken from the enumeration `name_s`, that is `ArrayLength.Enum(name_s, _)`;
 - * `length_t` is an array with indices taken from the enumeration `name_t`, that is `ArrayLength.Enum(name_t, _)`;
 - * `b` is `TRUE` if and only if `name_s` and `name_t` are the same.
 - All of the following apply (`T_TUPLE`):
 - * `s` has the **underlying type** of a tuple with type list `lis_s`, that is `T_Tuple(lis_s)`;
 - * `t` has the **underlying type** of a tuple with type list `lis_t`, that is `T_Tuple(lis_t)`;
 - * equating the lengths of `lis_s` and `lis_t` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b = FALSE`;
 - * checking at each index `i` of the list `lis_s` whether the type `lis_t[i]` **type-satisfies** the type `lis_s[i]` yields `bi` //^{#TE};
 - * `b` is `TRUE` if and only if all `bi` are `TRUE`;
 - All of the following apply (`STRUCTURED`):
 - * `s` has the **underlying type** `L(fields_s)`, which is a **structured type**;
 - * `t` has the **underlying type** `L(fields_t)`, which is a **structured type**;
 - * since both underlying types have the same AST label they are either both record types or both exception types;
 - * `b` is `TRUE` if and only if for each field in `fields_s` with type `ty_s` there exists a field in `fields_t` with type `ty_t` such that both `ty_s` and `ty_t` are determined to be **type-equivalent** in `tenv`.

8.2.2 Formally

ERROR1

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \#TE}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

ERROR2

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \#TE}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

DIFFERENT_LABELS

$$\frac{\begin{array}{l} \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \\ \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad \text{ast_label}(t2) \neq \text{ast_label}(s2) \end{array}}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE}}$$

SIMPLE

$$\frac{\begin{array}{l} \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \\ \text{ast_label}(t2) \in \{T_Int, T_Real, T_String, T_Bool\} \quad b := \text{ast_label}(s2) = \text{ast_label}(t2) \end{array}}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_ENUM

$$\frac{\begin{array}{l} \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} T_Enum(lis_t) \\ \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} T_Enum(lis_s) \quad b := lis_t = lis_s \end{array}}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_BITS

$$\frac{\begin{array}{l} \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} T_Bits(w_s, bfs_s) \\ \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} T_Bits(w_t, bfs_t) \\ \text{bitwidth_equal}(\text{tenv}, w_s, w_t) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\ \text{bitfields_included}(\text{tenv}, bfs_s, bfs_t) \xrightarrow{\text{type}} b \parallel \#TE \end{array}}{\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

$$\begin{array}{c}
\text{T_ARRAY_EXPR} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, \text{ty_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, \text{ty_t}) \\
\text{type_equal}(\text{tenv}, \text{ty_s}, \text{ty_t}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_s}) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_t}) \\
\text{expr_equal}(\text{tenv}, \text{length_expr_s}, \text{length_expr_t}) \xrightarrow{\text{type}} b
\end{array} \\
\hline
\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{T_ARRAY_ENUM} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, \text{ty_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, \text{ty_t}) \\
\text{type_equal}(\text{tenv}, \text{ty_s}, \text{ty_t}) \xrightarrow{\text{type}} \text{TRUE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \xrightarrow{\text{type}} \text{TRUE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_s}, _) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_t}, _) \quad b := \text{name_s} = \text{name_t}
\end{array} \\
\hline
\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{T_TUPLE} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_t}) \\
\text{equal_length}(\text{lis_s}, \text{lis_t}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{lis_s}) : \text{type_satisfies}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} b_i \parallel \text{T_TypeError} \\
b := \bigwedge_{i=1}^k b_i
\end{array} \\
\hline
\text{structural_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

For a list of typed fields **fields**, we define the set of its field identifiers as:

$$\text{field_names}(\text{fields}) \triangleq \{\text{id} \mid (\text{id}, t) \in \text{fields}\}$$

We define the type associated with the field name **id** in a list of typed fields **fields**, if there is a unique one, as follows:

$$\text{field_type}(\text{fields}, \text{id}) \triangleq \begin{cases} t & \text{if } \{t' \mid (\text{id}, t') \in \text{fields}\} = \{t\} \\ \perp & \text{otherwise} \end{cases}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
L \in \{\text{T_Record}, \text{T_Exception}\} \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} L(\text{fields_s}) \\
\quad \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} L(\text{fields_t}) \\
\text{names_s} := \text{field_names}(\text{fields_s}) \quad \text{names_t} := \text{field_names}(\text{fields_t}) \\
\quad \text{bool_transition}(\text{names_s} \subseteq \text{names_t}) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
(\text{id}, \text{ty_s}) \in \text{fields_s} : \text{type_equal}(\text{tenv}, \text{ty_s}, \text{field_type}(\text{fields_t}, \text{id})) \xrightarrow{\text{type}} b_{\text{id}} \\
\quad b := \bigwedge_{\text{id} \in \text{names_s}} b_{\text{id}} \\
\hline
\text{structural_subtype_satisfies}(\text{tenv}, s, t) \xrightarrow{\text{type}} b
\end{array}$$

8.3 TypingRule.DomainSubtypeSatisfaction

The predicate

$$\text{domain_subtype_satisfies}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether a type t *domain-subtype-satisfies* a type s in environment tenv , assuming that t structurally-subtype-satisfies s , returning the result b or a type error, if one is detected. The function assumes that both t and s are well-typed according to Chapter 9.

8.3.1 Prose

One of the following applies:

- All of the following apply (ERROR1):
 - * obtaining the **structure** of s results in a type error;
 - * the rule gives a type error.
- All of the following apply (ERROR2):
 - * obtaining the **structure** of s results in a type s_struct ;
 - * obtaining the **structure** of t results in a type error;
 - * the rule gives a type error.
- All of the following apply (SIMPLE):
 - * obtaining the **structure** of s results in a type s_struct ;
 - * the AST label of s_struct is either T_Tuple , T_Array , T_Record , or T_Exception ;
 - * b is **TRUE**.
- All of the following apply (SYMBOLIC):

- * obtaining the `structure` of `s` results in a type `s_struct`;
- * obtaining the `structure` of `t` results in a type `t_struct`;
- * the AST label of `s_struct` is either `T_Real`, `T_String`, `T_Bool`, `T_Enum`, or `T_Int`;
- * determining whether `s` subsumes `t` in `tenv` via symbolic reasoning results in `b`.

8.3.2 Formally

ERROR

$$\frac{(\text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} \#TE) \vee (\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \#TE)}{\text{domain_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

SIMPLE

$$\frac{\left(\begin{array}{l} \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\ \text{ast_label}(s_struct) \in \{T_Tuple, T_Array, T_Record, T_Exception\} \\ s_struct \in \{T_Real, T_String, T_Bool, T_Enum, \text{unconstrained_integer}\} \end{array} \vee \right)}{\text{domain_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}$$

CONSTRAINED_INT_ENUM

$$\frac{\begin{array}{l} \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\ (s_struct = T_Int(c) \wedge c \neq \text{Unconstrained}) \vee \text{ast_label}(s_struct) = T_Enum \\ \text{sym_subsumes}(\text{tenv}, s_struct, t_struct) \xrightarrow{\text{type}} b \end{array}}{\text{domain_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

BITS

$$\frac{\begin{array}{l} \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\ \text{ast_label}(s_struct) = T_Bits \quad \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\ \text{sym_subsumes}(\text{tenv}, s_struct, t_struct) \xrightarrow{\text{type}} b \end{array}}{\text{domain_subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

8.4 TypingRule.SubtypeSatisfaction

The predicate

$$\text{subtype_satisfies}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether a type `t` *subtype-satisfies* a type `s` in environment `tenv`, returning the result `b` or a type error, if one is detected. The function assumes that both `t` and `s` are well-typed according to Chapter 9.

8.4.1 Prose

All of the following apply:

- determining whether \mathbf{t} structurally-subtype-satisfies \mathbf{s} yields $\mathbf{b1} \# \mathbf{TE}$;
- determining whether \mathbf{t} domain-subtype-satisfies \mathbf{s} yields $\mathbf{b2}$;
- \mathbf{b} is **TRUE** if and only if both $\mathbf{b1}$ and $\mathbf{b2}$ are **TRUE**.

$$\frac{\begin{array}{l} \text{structural_subtype_satisfies}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{b1} \# \mathbf{TE} \\ \text{domain_subtype_satisfies}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{b2} \end{array}}{\text{subtype_satisfies}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{b} \quad \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2}}$$

8.5 TypingRule.TypeSatisfaction

The predicate

$$\text{type_satisfies}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{ty}}^{\mathbf{t}}, \overbrace{\mathbf{ty}}^{\mathbf{s}}) \longrightarrow \overbrace{\mathbf{B}}^{\mathbf{b}} \cup \overbrace{\mathbf{TTypeError}}^{\# \mathbf{TE}}$$

tests whether a type \mathbf{t} *type-satisfies* a type \mathbf{s} in environment tenv , returning the result \mathbf{b} or a type error, if one is detected.

We also define

$$\text{checked_typesat}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{ty}}^{\mathbf{t}}, \overbrace{\mathbf{ty}}^{\mathbf{s}}) \longrightarrow \{\mathbf{TRUE}\} \cup \overbrace{\mathbf{TTypeError}}^{\# \mathbf{TE}}$$

which is the same as *type-satisfies*, but yields a type error when *type-satisfies*($\text{tenv}, \mathbf{t}, \mathbf{s}$) is **FALSE**.

These functions assume that both \mathbf{t} and \mathbf{s} are well-typed according to Chapter 9.

8.5.1 Prose

One of the following applies:

- All of the following apply (SUBTYPES):
 - * \mathbf{t} subtypes \mathbf{s} in tenv ;
 - * \mathbf{b} is **TRUE**.
- All of the following apply (ANONYMOUS):
 - * \mathbf{t} does not subtype \mathbf{s} in tenv ;
 - * at least one of \mathbf{t} and \mathbf{s} is an anonymous type in tenv ;
 - * determining whether \mathbf{t} *subtype-satisfies* \mathbf{s} in tenv yields $\mathbf{TRUE} \# \mathbf{TE}$;

- * b is **TRUE**.
- All of the following apply (T_BITS):
 - * t does not subtype s in $tenv$;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t **subtype-satisfies** s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * t is a bitvector type with width $width_t$ and no bitfields;
 - * obtaining the **structure** of s in $tenv$ yields a bitvector type with width $width_s \#TE$;
 - * determining whether $width_t$ and $width_s$ are **bitwidth-equivalent** yields b .
- All of the following apply ($OTHERWISE1$):
 - * t does not subtype s in $tenv$;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t **subtype-satisfies** s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the **structure** of s in $tenv$ yields a $s_struct \#TE$;
 - * at least one of t and s_struct is not a bitvector type;
- All of the following apply ($OTHERWISE2$):
 - * t does not subtype s in $tenv$;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t **subtype-satisfies** s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the **structure** of s in $tenv$ yields a $s_struct \#TE$;
 - * both t and s_struct are bitvector types;
 - * the bitvector type t has a non-empty list of bitfields;
 - * b is **FALSE**;

8.5.2 Example

In the specification:

```

type T1 of integer;
  // the named type 'T1' whose structure is integer
type T2 of integer;
  // the named type 'T2' whose structure is integer
type pairT of (integer, T1);
  // the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);
  // legal since the right hand side has anonymous, non-primitive type (integer, T1)
return 0;
end

var pair: pairT = (1, dataT1) is legal since the right-hand-side has anonymous,
non-primitive type (integer, T1).

```

8.5.3 Example

In the specification:

```

type T1 of integer;
  // the named type 'T1' whose structure is integer
type T2 of integer;
  // the named type 'T2' whose structure is integer
type pairT of (integer, T1);
  // the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataAsInt: integer = dataT1;
  pair = (1, dataAsInt);
  // legal since the right-hand-side has anonymous,
  // primitive type (integer, integer)
  return 0;
end

pair = (1, dataAsInt); is legal since the right-hand-side has anonymous, primitive
type (integer, integer).

```

8.5.4 Example

In the specification:


```

type T1 of integer;
  // the named type 'T1' whose structure is integer
type T2 of integer;
  // the named type 'T2' whose structure is integer
type pairT of (integer, T1);
  // the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataT2: T2 = 10;
  pair = (1, dataT2);
  // illegal since the right-hand-side has anonymous,
  // non-primitive type (integer, T2)
  // which does not subtype-satisfy named type pairT
  return 0;
end

```

`pair = (1, dataT2);` is illegal since the right-hand-side has anonymous, non-primitive type `(integer, T2)` which does not subtype-satisfy named type `pairT`.

8.5.5 Formally

SUBTYPES

$$\frac{\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}$$

ANONYMOUS

$$\frac{\begin{array}{l} \text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\ \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad b1 \vee b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \end{array}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}$$

T_BITS

$$\frac{\begin{array}{l} \text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \quad \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \\ \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \quad \neg((b1 \vee b2) \wedge b3) \\ t \stackrel{\text{is}}{=} \text{T_Bits}(\text{width_t}, []) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Bits}(\text{width_s}, _) \quad \text{// \#TE} \\ \text{bitwidth_equal}(\text{tenv}, \text{width_t}, \text{width_s}) \xrightarrow{\text{type}} b \end{array}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

OTHERWISE1

$$\begin{array}{c}
\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\
\text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\
\neg((b1 \vee b2) \wedge b3) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\
ast_label(t) \neq T_Bits \vee ast_label(s_struct) \neq T_Bits \\
\hline
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

OTHERWISE2

$$\begin{array}{c}
\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\
\text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\
\neg((b1 \vee b2) \wedge b3) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\
ast_label(t) = T_Bits \wedge ast_label(s_struct) = T_Bits \\
t \stackrel{\text{is}}{=} T_Bits(\text{width}_t, \text{bitfields}) \quad \text{bitfields} \neq [] \\
\hline
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

The rules for the checked type-satisfy predicate are:

TRUE

$$\begin{array}{c}
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \quad \text{//} \quad \text{\#TE} \\
\hline
checked_typesat(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

ERROR

$$\begin{array}{c}
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\hline
checked_typesat(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
\end{array}$$

8.5.6 Comments

Since the subtype relation is a partial order, it is reflexive. Therefore every type t is a subtype of itself, and as a consequence, every type t *type-satisfies* itself.

8.6 TypingRule.TypeClash

The predicate

$$type_clashes(\overbrace{\text{\textcolor{blue}{SIE}}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\text{\textcolor{blue}{B}}}^b \cup \overbrace{\text{\textcolor{blue}{TTypeError}}}^{\text{\#TE}}$$

tests whether a type t *type-clashes* with a type s in environment tenv , returning the result b or a type error, if one is detected.

8.6.1 Prose

One of the following applies:

- All of the following apply (SUBTYPE):
 - * either s subtypes t or t subtypes s ;
 - * b is **TRUE**.
- All of the following apply (SIMPLE):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the **structure** of t in tenv yields $t_struct\#\text{TE}$;
 - * obtaining the **structure** of s in tenv yields $s_struct\#\text{TE}$;
 - * both t_struct and s_struct are one of the following types: `integer`, `real`, `string`;
 - * b is **TRUE**.
- All of the following apply (T_ENUM):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the **structure** of t in tenv yields an enumeration type with labels `lis_t`;
 - * obtaining the **structure** of s in tenv yields an enumeration type with labels `lis_s`;
 - * b is **TRUE** if and only if `lis_s` and `lis_t` are equal.
- All of the following apply (T_ARRAY):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the **structure** of t in tenv yields an array type with element type `ty_t`;
 - * obtaining the **structure** of s in tenv yields an array type with element type `ty_s`;
 - * b is **TRUE** if and only if `ty_t` and `ty_s` type-clash.
- All of the following apply (T_TUPLE):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the **structure** of t in tenv yields a tuple type with element types $t_{1..k}$;
 - * obtaining the **structure** of s in tenv yields a tuple type with element types $s_{1..n}$;
 - * if $n \neq k$ the rule short-circuits with $b = \text{FALSE}$;
 - * b is **TRUE** if and only if t_i type-clashes with s_i , for all $i = 1..k$.

- All of the following apply (OTHERWISE_DIFFERENT_LABELS):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the `structure` of t in $tenv$ yields t_struct ;
 - * obtaining the `structure` of s in $tenv$ yields s_struct ;
 - * s_struct and t_struct have different AST labels;
 - * b is `FALSE`;
- All of the following apply (OTHERWISE_STRUCTURED):
 - * neither s subtypes t nor t subtypes s ;
 - * obtaining the `structure` of t in $tenv$ yields t_struct ;
 - * obtaining the `structure` of s in $tenv$ yields s_struct ;
 - * s_struct and t_struct have the same AST label;
 - * t_struct (and thus s_struct) is a `structured type`;
 - * b is `FALSE`;

8.6.2 Formally

SUBTYPE

$$\frac{(is_subtype(tenv, s, t) \xrightarrow{type} \text{TRUE}) \vee (is_subtype(tenv, t, s) \xrightarrow{type} \text{TRUE})}{type_clashes(tenv, t, s) \xrightarrow{type} \overbrace{\text{TRUE}}^b}$$

SIMPLE

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{type} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{type} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{type} t_struct \quad // \quad \#TE \\ get_structure(tenv, s) \xrightarrow{type} s_struct \quad // \quad \#TE \\ ast_label(t_struct) = ast_label(s_struct) \\ ast_label(t_struct) \in \{T_Int, T_Real, T_String, T_Bits\} \end{array}}{type_clashes(tenv, t, s) \xrightarrow{type} \overbrace{\text{TRUE}}^b}$$

T_ENUM

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{type} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{type} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{type} T_Enum(_, lis_s) \\ get_structure(tenv, s) \xrightarrow{type} T_Enum(_, lis_t) \end{array}}{type_clashes(tenv, t, s) \xrightarrow{type} \overbrace{lis_s = lis_t}^b}$$

T_ARRAY

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} T_Array(_, ty_t) \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} T_Array(_, ty_s) \quad type_clashes(\text{tenv}, ty_t, ty_s) \xrightarrow{\text{type}} b \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_TUPLE

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} T_Tuple(t_{1..k}) \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} T_Tuple(s_{1..n}) \quad bool_transition(n = k) \longrightarrow \text{TRUE} // \text{FALSE} \\
i = 1..k : type_clashes(\text{tenv}, t_i, s_i) \xrightarrow{\text{type}} b_i \quad b := \bigwedge_{i=1}^k b_i \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

OTHERWISE_DIFFERENT_LABELS

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) \neq ast_label(s_struct) \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

OTHERWISE_STRUCTURED

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) = ast_label(s_struct) \\
b := ast_label(t_struct) \in \{T_Record, T_Exception\} \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

8.6.3 Comments

Note that if t subtype-satisfies s then t and s type-clash, but not the other way around.

Note that type-clashing is an equivalence relation. Therefore if t type-clashes with A and B then it is also the case that A and B type-clash.

8.7 TypingRule.LowestCommonAncestor

Annotating a conditional expression (see Section 11.9), requires finding a single type that can be used to annotate the results of both subexpressions. The function

$$\text{lowest_common_ancestor}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{ty}}^{\text{s}}) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the *lowest common ancestor* of types t and s in tenv — ty . The result is a type error if a lowest common ancestor does not exist or a type error is detected.

8.7.1 Prose

One of the following applies:

- All of the following apply (TYPE_EQUAL):
 - * t is *type_equal* to s in tenv ;
 - * ty is s (can as well be t).
- All of the following apply:
 - * t is not *type_equal* to s in tenv and one of the following applies:
 - * All of the following apply (NAMED_SUBTYPE1):
 - t is a named type with identifier name_t , that is, $\text{T_Named}(\text{name_t})$;
 - s is a named type with identifier name_s , that is, $\text{T_Named}(\text{name_s})$;
 - there is no *named lowest common ancestor* of name_s and name_t in tenv ;
 - obtaining the *underlying type* of s yields $\text{anon_s}\text{\#TE}$;
 - obtaining the *underlying type* of t yields $\text{anon_t}\text{\#TE}$;
 - obtaining the lowest common ancestor of anon_s and anon_t in tenv yields $\text{ty}\text{\#TE}$.
 - * All of the following apply (NAMED_SUBTYPE2):
 - t is a named type with identifier name_t , that is, $\text{T_Named}(\text{name_t})$;
 - s is a named type with identifier name_s , that is, $\text{T_Named}(\text{name_s})$;
 - the *named lowest common ancestor* of name_s and name_t in tenv is $\text{name}\text{\#TE}$;
 - ty is the named type with identifier name , that is, $\text{T_Named}(\text{name})$.
 - * All of the following apply (ONE_NAMED1):
 - only one of t or s is a named type;
 - obtaining the *underlying type* of s yields $\text{anon_s}\text{\#TE}$;
 - obtaining the *underlying type* of t yields $\text{anon_t}\text{\#TE}$;
 - anon_t is *type_equal* to anon_s ;

- ty is t if it is a named type (that is, $ast_label(t) = T_Named$), and s otherwise.
- * All of the following apply (ONE_NAMED2):
 - only one of t or s is a named type;
 - obtaining the *underlying type* of s yields $anon_s\#TE$;
 - obtaining the *underlying type* of t yields $anon_t\#TE$;
 - $anon_t$ is not *type-equal* to $anon_s$;
 - the lowest common ancestor of $anon_t$ and $anon_s$ in $tenv$ is $ty\#TE$.
- * All of the following apply (T_INT_UNCONSTRAINED):
 - both t and s are integer types;
 - at least one of t or s is an unconstrained integer type;
 - ty is the unconstrained integer type.
- * All of the following apply (T_INT_PARAMETERIZED):
 - neither t nor s are the unconstrained integer type;
 - one of t and s is a *parameterized integer type*;
 - the *well-constrained version* of t is $t1$;
 - the *well-constrained version* of s is $s1$;
 - ty the lowest common ancestor of $t1$ and $s1$ in $tenv$ is $ty\#TE$.
- * All of the following apply (T_INT_WELLCONSTRAINED):
 - t is a well-constrained integer type with constraints cs_t ;
 - s is a well-constrained integer type with constraints cs_s ;
 - ty is the well-constrained integer type with constraints $cs_t + cs_s$.
- * All of the following apply (T_BITS):
 - t is a bitvector type with with length expression e_t , that is, $T_Bits(e_t, _)$;
 - s is a bitvector type with with length expression e_s , that is, $T_Bits(e_s, _)$;
 - applying *type-equal* to t and s in $tenv$ yields **FALSE**;
 - applying *expr-equal* to e_t and e_s in $tenv$ yields b_equal ;
 - checking whether b_equal is **TRUE** yields $TRUE\#TE_LCA$;
 - ty is a bitvector type with length expression e_t and an empty bitfield list, that is, $T_Bits(e_t, [])$.
- * All of the following apply (T_ARRAY):
 - t is an array type with width expression $width_t$ and element type ty_t ;
 - s is an array type with width expression $width_s$ and element type ty_s ;
 - applying *array-length-equal* to $width_t$ and $width_s$ in $tenv$ to equate the array lengths, yields $b_equal_length\#TE$;
 - checking that b_equal_length is **TRUE** yields $TRUE\#TE_LCA$;
 - the lowest common ancestor of ty_t and ty_s is $t1\#TE$;
 - ty is an array type with width expression $width_s$ and element type $t1$.

- * All of the following apply (T_TUPLE):
 - \mathbf{t} is a tuple type with type list $\mathbf{lis_t}$;
 - \mathbf{s} is a tuple type with type list $\mathbf{lis_s}$;
 - checking whether $\mathbf{lis_t}$ and $\mathbf{lis_s}$ have the same number of elements yields **TRUE** or a type error, which short-circuits the entire rule (indicating that the number of elements in both tuples is expected to be the same and thus there is no lowest common ancestor);
 - applying *lowest_common_ancestor* to $\mathbf{lis_t}[i]$ and $\mathbf{lis_s}[i]$ in \mathbf{tenv} , for every position of $\mathbf{lis_t}$, yields $\mathbf{t_i}$ // **#TE**;
 - define \mathbf{li} to be the list of types $\mathbf{t_i}$, for every position of $\mathbf{lis_t}$;
 - define \mathbf{ty} as the tuple type with list of types \mathbf{li} , that is, $\mathbf{T_Tuple(li)}$.
- * All of the following apply (ERROR):
 - either the AST labels of \mathbf{t} and \mathbf{s} are different, or one of them is $\mathbf{T_Enum}$, $\mathbf{T_Record}$, or $\mathbf{T_Exception}$;
 - the result is a type error indicating the lack of a lowest common ancestor.

8.7.2 Formally

Since we do not impose a canonical representation on types (e.g., `integer {1, 2}` is equivalence to `integer {1..2}`), the lowest common ancestor is not unique. We define *lowest_common_ancestor*($\mathbf{tenv}, \mathbf{t}, \mathbf{s}$) to be any type $\mathbf{t'}$ that is *type-equivalent* to the lowest common ancestor of \mathbf{t} and \mathbf{s} .

$$\begin{array}{c}
 \text{TYPE_EQUAL} \\
 \hline
 \frac{\text{type_equal}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{TRUE}}{\text{lowest_common_ancestor}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \overbrace{\mathbf{s}}^{\mathbf{ty}}} \\
 \\
 \text{NAMED_SUBTYPE1} \\
 \hline
 \frac{
 \begin{array}{l}
 \mathbf{t} = \mathbf{T_Named}(\mathbf{name_s}) \quad \mathbf{s} = \mathbf{T_Named}(\mathbf{name_t}) \quad \text{type_equal}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{FALSE} \\
 \text{named_lowest_common_ancestor}(\mathbf{tenv}, \mathbf{name_s}, \mathbf{name_t}) \xrightarrow{\text{type}} \mathbf{None} \quad // \quad \mathbf{\#TE} \\
 \text{make_anonymous}(\mathbf{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{anon_s} \quad // \quad \mathbf{\#TE} \\
 \text{make_anonymous}(\mathbf{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{anon_t} \quad // \quad \mathbf{\#TE} \\
 \text{lowest_common_ancestor}(\mathbf{tenv}, \mathbf{anon_t}, \mathbf{anon_s}) \xrightarrow{\text{type}} \mathbf{ty} \quad // \quad \mathbf{\#TE}
 \end{array}
 }{\text{lowest_common_ancestor}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{ty}} \\
 \\
 \text{NAMED_SUBTYPE2} \\
 \hline
 \frac{
 \begin{array}{l}
 \mathbf{t} = \mathbf{T_Named}(\mathbf{name_s}) \quad \mathbf{s} = \mathbf{T_Named}(\mathbf{name_t}) \quad \text{type_equal}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{FALSE} \\
 \text{named_lowest_common_ancestor}(\mathbf{tenv}, \mathbf{name_s}, \mathbf{name_t}) \xrightarrow{\text{type}} \langle \mathbf{name} \rangle \quad // \quad \mathbf{\#TE}
 \end{array}
 }{\text{lowest_common_ancestor}(\mathbf{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \overbrace{\mathbf{T_Named}(\mathbf{name})}^{\mathbf{ty}}}
 \end{array}$$

ONE_NAMED1

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad (\text{ast_label}(t) = \text{T_Named} \vee \text{ast_label}(s) = \text{T_Named}) \\
\text{ast_label}(t) \neq \text{ast_label}(s) \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{anon_s} \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{anon_t} \quad \# \text{TE} \\
\text{type_equal}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{TRUE} \\
\text{ty} := \text{choice}(\text{ast_label}(t) = \text{T_Named}, t, s) \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}
\end{array}$$

ONE_NAMED2

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad (\text{ast_label}(t) = \text{T_Named} \vee \text{ast_label}(s) = \text{T_Named}) \\
\text{ast_label}(t) \neq \text{ast_label}(s) \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{anon_s} \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{anon_t} \quad \# \text{TE} \\
\text{type_equal}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{lowest_common_ancestor}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{ty} \quad \# \text{TE} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}
\end{array}$$

T_INT_UNCONSTRAINED

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \\
\text{is_unconstrained_integer}(t) \vee \text{is_unconstrained_integer}(s) \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{ty}}
\end{array}$$

T_INT_PARAMETERIZED

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \quad \neg \text{is_unconstrained_integer}(t) \\
\neg \text{is_unconstrained_integer}(s) \quad \text{is_parameterized_integer}(t) \vee \text{is_parameterized_integer}(s) \\
\text{to_well_constrained}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \text{to_well_constrained}(\text{tenv}, s) \xrightarrow{\text{type}} s1 \\
\text{lowest_common_ancestor}(\text{tenv}, t1, s1) \xrightarrow{\text{type}} \text{ty} \quad \# \text{TE} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}
\end{array}$$

T_INT_WELLCONSTRAINED

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
t \stackrel{\text{is}}{=} \text{T_Int}(\text{WellConstrained}(\text{cs_t})) \quad s \stackrel{\text{is}}{=} \text{T_Int}(\text{WellConstrained}(\text{cs_s})) \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_t} + \text{cs_s}))}^{\text{ty}}
\end{array}$$

T_BITS

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{expr_equal}(\text{tenv}, e_t, e_s) \xrightarrow{\text{type}} \text{b_equal} \quad \text{check}(\text{b_equal}, \text{TE_LCA}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Bits}(e_t, _)}^t, \overbrace{\text{T_Bits}(e_s, _)}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(e_t, [_])}^{\text{ty}}
\end{array}$$

T_ARRAY

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{array_length_equal}(\text{tenv}, \text{width_t}, \text{width_s}) \xrightarrow{\text{type}} \text{b_equal_length} \parallel \text{\#TE} \\
\text{check}(\text{b_equal_length}, \text{TE_LCA}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\
\text{lowest_common_ancestor}(\text{tenv}, \text{ty_t}, \text{ty_s}) \xrightarrow{\text{type}} t1 \parallel \text{\#TE} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Array}(\text{width_t}, \text{ty_t})}^t, \overbrace{\text{T_Array}(\text{width_s}, \text{ty_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Array}(\text{width_t}, t1)}^{\text{ty}}
\end{array}$$

T_TUPLE

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{equal_length}(\text{lis_t}, \text{lis_s}) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_LCA}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
i \in \text{indices}(\text{lis_t}) : \text{lowest_common_ancestor}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} t_i \parallel \text{\#TE} \\
\text{li} := [i \in \text{indices}(\text{lis_t}) : t_i] \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Tuple}(\text{lis_t})}^t, \overbrace{\text{T_Tuple}(\text{lis_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{li})}^{\text{ty}}
\end{array}$$

ERROR

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
(\text{ast_label}(t) \neq \text{ast_label}(s)) \vee \text{ast_label}(t) \in \{\text{T_Enum}, \text{T_Record}, \text{T_Exception}\} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_LCA})
\end{array}$$

8.8 TypingRule.CheckUnop

The function

$$\text{check_unop}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{unop}}^{\text{op}}, \overbrace{\text{ty}}^t) \longrightarrow \overbrace{\text{ty}}^s \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

determines the result type of applying a unary operator when the type of its operand is known. Similarly, we determine the negation of integer constraints. Otherwise, the result is a type error.

8.8.1 Prose

One of the following applies:

- All of the following apply (BNOT_T_BOOL):
 - * `op` is BNOT;
 - * determining whether `t` *type-satisfies* `T_Bool` yields `TRUE`//*#TE*;
 - * `s` is `T_Bool`;
- All of the following apply (NEG_ERROR):
 - * `op` is NEG;
 - * determining whether `t` *type-satisfies* `T_Real` yields `FALSE`//*#TE*;
 - * determining whether `t` *type-satisfies* `unconstrained_integer` yields `FALSE`//*#TE*;
 - * the result is a type error indicating the NEG is appropriate only for `real` and `integer` types;
- All of the following apply (NEG_T_REAL):
 - * `op` is NEG;
 - * determining whether `t` *type-satisfies* `T_Real` yields `TRUE`;
 - * `s` is `T_Real`;
- All of the following apply (NEG_T_INT_UNCONSTRAINED):
 - * `op` is NEG;
 - * obtaining the *well-constrained structure* of `t` yields `unconstrained_integer`//*#TE*;
 - * `s` is `unconstrained_integer`;
- All of the following apply (NEG_T_INT_WELL_CONSTRAINED):
 - * `op` is NEG;
 - * obtaining the *well-constrained structure* of `t` yields the well-constrained integer type with constraints `vcs`//*#TE*;
 - * negating the constraints in `vcs` (see *negate_constraint*) yields `cs_new`;
 - * `s` is the well-constrained integer type with constraints `cs_new`, that is, `T_Int(WellConstrained(cs_new))`;
- All of the following apply (NOT_T_BITS):
 - * `op` is NOT;
 - * `t` has the structure of a bitvector;
 - * `s` is `t`.

8.8.2 Formally

$$\frac{\text{BNOT_T_BOOL} \quad \text{checked_typesat}(\text{tenv}, t1, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \text{BNOT}, t1) \xrightarrow{\text{type}} \text{T_Bool}}$$

We now define the helper function

$$\text{negate_constraint}(\text{int_constraint}) \longrightarrow \text{int_constraint}$$

which takes an integer constraint and returns the constraint that corresponds to the negation of all the values it represents:

$$\text{negate_constraint}(\text{Constraint_Exact}(e)) \xrightarrow{\text{type}} \text{Constraint_Exact}(\text{E_Unop}(\text{MINUS}, e))$$

$$\frac{\text{negate_constraint}(\text{Constraint_Range}(\text{top}, \text{bot})) \xrightarrow{\text{type}}}{\text{Constraint_Range}(\text{E_Unop}(\text{MINUS}, \text{bot}), \text{E_Unop}(\text{MINUS}, \text{top}))}$$

$$\frac{\text{NEG_ERROR} \quad \begin{array}{l} \text{type_satisfies}(\text{tenv}, t, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \#TE \\ \text{type_satisfies}(\text{tenv}, t, \text{T_Real}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \#TE \end{array}}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \text{TypeError}(\text{InappropriateTypeForNeg})}$$

$$\frac{\text{NEG_T_REAL} \quad \text{type_satisfies}(\text{tenv}, t, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE}}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{T_Real}}^{\text{s}}}$$

$$\frac{\text{NEG_T_INT_UNCONSTRAINED} \quad \text{get_well_constrained_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{unconstrained_integer} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{s}}}$$

$$\frac{\text{NEG_T_INT_WELL_CONSTRAINED} \quad \begin{array}{l} \text{get_well_constrained_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{vcs})) \\ c \in \text{vcs} : \text{negate_constraint}(c) \xrightarrow{\text{type}} \text{neg}_c \quad \text{cs_new} := [c \in \text{vcs} : \text{neg}_c] \end{array}}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_new}))}^{\text{s}}}$$

$$\frac{\text{NOT_T_BITS} \quad \text{check_structure}(\text{tenv}, t, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \overbrace{\text{NOT}}^{\text{op}}, t) \xrightarrow{\text{type}} t}$$

8.9 TypingRule.CheckBinop

The function

$$\text{check_binop}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

determines the result type t of applying the binary operator op to operands of type t1 and t2 in the static environment tenv . Otherwise, the result is a type error.

8.9.1 Prose

One of the following applies:

- All of the following apply (BOOLEAN):
 - * op is AND, OR, EQ_OP or IMPL;
 - * determining whether t1 `type-satisfies` `T_Bool` in tenv yields `TRUE//\#TE`;
 - * determining whether t2 `type-satisfies` `T_Bool` in tenv yields `TRUE//\#TE`;
 - * t is `T_Bool`.
- All of the following apply (BITS_BOOL):
 - * op is AND, OR, or XOR;
 - * checking whether t1 and t2 have the `structure` of bitvector types of the same width in tenv yields `TRUE//\#TE`;
 - * the bitvector width of t1 in tenv is w ;
 - * t is the bitvector type of width w and empty list of bitfields, that is, `T_Bits(w, [])`.
- All of the following apply (PLUS_MINUS_ERROR):
 - * op is PLUS or MINUS;
 - * obtaining the `structure` of t1 in tenv is `t1_struct//\#TE`;
 - * `t1_struct` is neither a bitvector type nor an integer type;
 - * the result is a type error indicating that the type of t1 is inappropriate for op .
- All of the following apply (PLUS_MINUS_BITS_INT):
 - * op is PLUS or MINUS;
 - * obtaining the `structure` of t1 in tenv is `t1_struct//\#TE`;
 - * `t1_struct` is a bitvector type;
 - * t2 `type-satisfies` the unconstrained integer type in tenv ;
 - * obtaining the bitwidth of t1 in tenv yields w .

- * t is the bitvector type of width w and empty list of bitfields, that is, $T_Bits(w, [])$.
- All of the following apply (PLUS_MINUS_BITS_BITS):
 - * op is PLUS or MINUS;
 - * obtaining the *structure* of $t1$ in $tenv$ is a bitvector of width $e1$, that is, $T_Bits(e1, _)$;
 - * $t2$ does not *type-satisfy* the unconstrained integer type in $tenv$
 - * obtaining the *structure* of $t2$ in $tenv$ is $t2_struct \#TE$;
 - * determining whether $t2_struct$ has a bitvector type yields $TRUE \#TE$;
 - * $t2_struct$ is a bitvector of width $w2$, that is, $T_Bits(w2, _)$;
 - * determining whether $w1$ and $w2$ are equal bitwidths yields b ;
 - * b is $TRUE \#TE$;
 - * t is the bitvector type of width $w1$ and empty list of bitfields, that is, $T_Bits(w1, [])$.
- All of the following apply (EQ_NEQ_ERROR):
 - * op is either EQ_OP or NEQ;
 - * the *underlying type* of $t1$ in $tenv$ is $t1_anon \#TE$;
 - * the *underlying type* of $t2$ in $tenv$ is $t2_anon \#TE$;
 - * the AST labels of $t1_anon$ and $t2_anon$ are different or one of them is not in $\{T_Int, T_Real, T_Bool, T_Bits, T_Enum\}$;
 - * the result is a type error indicating that the types of $t1$ and $t2$ are inappropriate for op .
- All of the following apply (EQ_NEQ_BITS):
 - * op is either EQ_OP or NEQ;
 - * the *underlying type* of $t1$ in $tenv$ is $t1_anon \#TE$;
 - * $t1_anon$ is a bitvector type;
 - * the *underlying type* of $t2$ in $tenv$ is $t2_anon \#TE$;
 - * $t2_anon$ is a bitvector type;
 - * checking whether the bitwidth of $t1_anon$ and $t2_anon$ is the same yields $TRUE \#TE$;
 - * t is T_Bool .
- All of the following apply (EQ_NEQ_BOOL):
 - * op is either EQ_OP or NEQ;
 - * the *underlying type* of $t1$ in $tenv$ is $T_Bool \#TE$;

- * the **underlying type** of `t2` in `tenv` is `T.Bool`//**#TE**;
- * checking whether `t1_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
- * checking whether `t2_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
- * `t` is `T.Bool`.
- All of the following apply (`EQ_NEQ_REAL`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the **underlying type** of `t1` in `tenv` is `T.Real`//**#TE**;
 - * the **underlying type** of `t2` in `tenv` is `T.Real`//**#TE**;
 - * checking whether `t1_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
 - * checking whether `t2_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
 - * `t` is `T.Bool`.
- All of the following apply (`EQ_NEQ_STRING`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the **underlying type** of `t1` in `tenv` is `T.String`//**#TE**;
 - * the **underlying type** of `t2` in `tenv` is `T.String`//**#TE**;
 - * checking whether `t1_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
 - * checking whether `t2_anon` **type-satisfies** `T.Bool` yields **TRUE**//**#TE**;
 - * `t` is `T.Bool`.
- All of the following apply (`EQ_NEQ_ENUM`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the **underlying type** of `t1` in `tenv` is `T.Enum(li1)`//**#TE**;
 - * the **underlying type** of `t2` in `tenv` is `T.Enum(li2)`//**#TE**;
 - * checking whether `li1` is equal to `li2` yields **TRUE**//**#TE**;
 - * `t` is `T.Bool`.
- All of the following apply (`RELATIONAL`):
 - * `op` is one of `LT`, `LEQ`, `GT`, and `GEQ`;
 - * determining whether both `t1` and `t2` **type-satisfy** the unconstrained integer type in `tenv` or both `t1` and `t2` **type-satisfy** the real type in `tenv` yields **TRUE**//**#TE**;
 - * `t` is `T.Bool`.
- All of the following apply (`ARITH_ERROR`):
 - * obtaining the **structure** of `t1` in `tenv` yields `t1_struct`//**#TE**;

- * obtaining the [structure](#) of `t2` in `tenv` yields `t2_struct` [//#TE](#);
- * the operator and the AST labels of `t1_struct` and `t2_struct` do not match any of the rows in the following table:

op	ast_label (<code>t1_struct</code>)	ast_label (<code>t2_struct</code>)
MUL	T_Int	T_Int
DIV	T_Int	T_Int
DIVRM	T_Int	T_Int
MOD	T_Int	T_Int
SHL	T_Int	T_Int
SHR	T_Int	T_Int
POW	T_Int	T_Int
PLUS	T_Int	T_Int
MINUS	T_Int	T_Int
PLUS	T_Real	T_Real
MINUS	T_Real	T_Real
MUL	T_Real	T_Real
RDIV	T_Real	T_Real
POW	T_Real	T_Int

- * the result is a type error indicating that the types of `t1` and `t2` are inappropriate for `op`.
- All of the following apply (`ARITH_T_INT_UNCONSTRAINED1`, `ARITH_T_INT_UNCONSTRAINED2`):
 - * `op` is one of {`MUL`, `DIV`, `DIVRM`, `MOD`, `SHL`, `SHR`, `POW`, `PLUS`, `MINUS`};
 - * the [well-constrained structure](#) of `t1` or `t2` in `tenv` is that of the unconstrained integer type;
 - * `t` is the unconstrained integer type;
- All of the following apply (`ARITH_T_INT_WELLCONSTRAINED`):
 - * `op` is one of {`MUL`, `POW`, `PLUS`, `MINUS`, `DIVRM`, `DIV`, `MOD`, `SHL`, `SHR`};
 - * the [well-constrained structure](#) of `t1` in `tenv` is that of a well-constrained integer type with constraints `cs1`;
 - * the [well-constrained structure](#) of `t2` in `tenv` is that of a well-constrained integer type with constraints `cs2`;
 - * applying [annotate_constraint_binop](#) to `op`, `cs1`, and `cs2` in `tenv` yields `vcs`;
 - * `t` is the integer type with constraints `vcs`;
- All of the following apply (`PLUS_MINUS_MUL_REAL`):
 - * `op` is one of {`PLUS`, `MINUS`, `MUL`};
 - * obtaining the [structure](#) of `t1` in `tenv` yields `T_Real`;

- * obtaining the [structure](#) of `t2` in `tenv` yields `T_Real`;
 - * `t` is `T_Real`.
- All of the following apply (`POW_REAL_INT`):
 - * `op` is one of `{PLUS, MINUS, MUL}`;
 - * obtaining the [structure](#) of `t1` in `tenv` yields `T_Real`;
 - * obtaining the [structure](#) of `t2` in `tenv` yields an integer type;
 - * `t` is `T_Real`.
 - All of the following apply (`RDIV`):
 - * `op` is one of `{RDIV}`;
 - * determining whether `t1` [type-satisfies](#) `T_Real` yields [TRUE](#) // [#TE](#);
 - * determining whether `t2` [type-satisfies](#) `T_Real` yields [TRUE](#) // [#TE](#);
 - * `t` is `T_Real`.

8.9.2 Formally

BOOLEAN

$$\frac{
 \begin{array}{l}
 \text{op} \in \{\text{BAND}, \text{BOR}, \text{IMPL}, \text{EQ_OP}\} \quad \text{checked_typesat}(\text{tenv}, \text{t1}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
 \text{checked_typesat}(\text{tenv}, \text{t2}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE}
 \end{array}
 }{
 \text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^{\text{t}}
 }$$

BITS_BOOL

$$\frac{
 \begin{array}{l}
 \text{op} \in \{\text{AND}, \text{OR}, \text{XOR}\} \quad \text{check_bits_equal_width}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
 \text{get_bitvector_width}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} w
 \end{array}
 }{
 \text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(w, [\])}^{\text{t}}
 }$$

$$\begin{array}{c}
\text{PLUS_MINUS_ERROR} \\
\frac{\text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \begin{array}{c} \text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_struct \quad // \quad \#TE \\ \text{ast_label}(t1_struct) \notin \{T_Bits, T_Int\} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})} \\
\\
\text{PLUS_MINUS_BITS_INT} \\
\frac{\text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \begin{array}{c} \text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_struct \quad // \quad \#TE \\ \text{ast_label}(t1_struct) = T_Bits \\ \text{type_satisfies}(\text{tenv}, t2, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{get_bitvector_width}(\text{tenv}, t1) \xrightarrow{\text{type}} w \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{T_Bits(w, [])}^t} \\
\\
\text{PLUS_MINUS_BITS_BITS} \\
\frac{\begin{array}{c} \text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \begin{array}{c} \text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} T_Bits(w1, _) \\ \text{type_satisfies}(\text{tenv}, t2, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{FALSE} \\ \text{get_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_struct \quad // \quad \#TE \\ \text{check}(\text{ast_label}(t2_struct) = T_Bits, \text{TE_EBT}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\ t2_struct \stackrel{\text{is}}{=} T_Bits(w2, _) \end{array} \\ \text{bitwidth_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{DifferentBitwidths}) \longrightarrow \text{TRUE} \quad // \quad \#TE \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{T_Bits(w1, [])}^t}
\end{array}$$

EQ_NEQ_ERROR

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\
\left(\begin{array}{l} \text{ast_label}(t1_anon) \neq \text{ast_label}(t2_anon) \\ \text{ast_label}(t1_anon) \notin \{\text{T_Int}, \text{T_Real}, \text{T_Bool}, \text{T_Bits}, \text{T_Enum}\} \end{array} \quad \vee \right) \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})
\end{array}$$

EQ_NEQ_BITS

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\
\text{ast_label}(t1_anon) = \text{T_Bits} \quad \text{ast_label}(t2_anon) = \text{T_Bits} \\
\text{check_bits_equal_width}(\text{tenv}, t1_anon, t2_anon) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

EQ_NEQ_BOOL

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t1_anon, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t2_anon, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

EQ_NEQ_REAL

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t1_anon, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t2_anon, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

EQ_NEQ_STRING

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t1_anon, \text{T_String}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, t2_anon, \text{T_String}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

EQ_NEQ_ENUM

$$\begin{array}{c}
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{T_Enum}(li1) \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{T_Enum}(li2) \\
\text{check}(li1 = li2, \text{DifferentEnumLabels}) \rightarrow \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

RELATIONAL

$$\begin{array}{c}
\text{op} \in \{\text{LT}, \text{LEQ}, \text{GT}, \text{GEQ}\} \quad \text{type_satisfies}(\text{tenv}, t1, \text{unconstrained_integer}) \xrightarrow{\text{type}} b1 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t2, \text{unconstrained_integer}) \xrightarrow{\text{type}} b2 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t1, \text{T_Real}) \xrightarrow{\text{type}} b3 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t2, \text{T_Real}) \xrightarrow{\text{type}} b4 \quad \# \text{TE} \\
\text{check}(b1 \wedge b2 \vee b3 \wedge b4, \text{TE_OTB}) \longrightarrow \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

ARITH_ERROR

$$\begin{array}{c}
\text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_struct \quad \# \text{TE} \\
\text{get_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_struct \quad \# \text{TE} \\
(\text{op}, \text{ast_label}(t1_struct), \text{ast_label}(t2_struct)) \notin \left\{ \begin{array}{l} (\text{MUL}, \text{T_Int}, \text{T_Int}) \\ (\text{DIV}, \text{T_Int}, \text{T_Int}) \\ (\text{DIVRM}, \text{T_Int}, \text{T_Int}) \\ (\text{MOD}, \text{T_Int}, \text{T_Int}) \\ (\text{SHL}, \text{T_Int}, \text{T_Int}) \\ (\text{SHR}, \text{T_Int}, \text{T_Int}) \\ (\text{POW}, \text{T_Int}, \text{T_Int}) \\ (\text{PLUS}, \text{T_Int}, \text{T_Int}) \\ (\text{MINUS}, \text{T_Int}, \text{T_Int}) \\ (\text{PLUS}, \text{T_Real}, \text{T_Real}) \\ (\text{MINUS}, \text{T_Real}, \text{T_Real}) \\ (\text{MUL}, \text{T_Real}, \text{T_Real}) \\ (\text{RDIV}, \text{T_Real}, \text{T_Real}) \\ (\text{POW}, \text{T_Real}, \text{T_Int}) \end{array} \right\} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})
\end{array}$$

The following two rules are not mutually exclusive, but both yield the same result when they are both active.

ARITH_T_INT_UNCONSTRAINED1

$$\begin{array}{c}
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{get_well_constrained_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{unconstrained_integer} \\
\text{get_well_constrained_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{T_Int}(_) \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{unconstrained_integer}
\end{array}$$

ARITH_T_INT_UNCONSTRAINED2

$$\begin{array}{c}
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{get_well_constrained_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{T_Int}(_) \\
\text{get_well_constrained_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{unconstrained_integer} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{unconstrained_integer}
\end{array}$$

$$\begin{array}{c}
\text{ARITH_T_INT_WELLCONSTRAINED} \\
\text{op} \in \{\text{MUL}, \text{POW}, \text{PLUS}, \text{MINUS}, \text{DIVRM}, \text{DIV}, \text{MOD}, \text{SHL}, \text{SHR}\} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, \mathfrak{t}1) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{cs1})) \quad \text{get_well_constrained_structure}(\text{tenv}, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{cs2})) \quad \text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{vcs}}{\text{check_binop}(\text{tenv}, \text{op}, \mathfrak{t}1, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Int}(\text{vcs})}
\end{array}$$

$$\begin{array}{c}
\text{PLUS_MINUS_MUL_REAL} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}, \text{MUL}\} \\
\frac{\text{get_structure}(\text{tenv}, \mathfrak{t}1) \xrightarrow{\text{type}} \text{T_Real} \quad \text{get_structure}(\text{tenv}, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Real}}{\text{check_binop}(\text{tenv}, \text{op}, \mathfrak{t}1, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

$$\begin{array}{c}
\text{POW_REAL_INT} \\
\frac{\text{get_structure}(\text{tenv}, \mathfrak{t}1) \xrightarrow{\text{type}} \text{T_Real} \quad \text{ast_label}(\text{get_structure}(\text{tenv}, \mathfrak{t}2)) \xrightarrow{\text{type}} \text{T_Int}}{\text{check_binop}(\text{tenv}, \text{POW}, \mathfrak{t}1, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

$$\begin{array}{c}
\text{RDIV} \\
\frac{\text{checked_typesat}(\text{tenv}, \mathfrak{t}1, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \quad \text{checked_typesat}(\text{tenv}, \mathfrak{t}2, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{check_binop}(\text{tenv}, \text{RDIV}, \mathfrak{t}1, \mathfrak{t}2) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

8.10 TypingRule.FindNamedLCA

The function

$$\text{named_lowest_common_ancestor}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\mathfrak{t}}, \overbrace{\text{ty}}^{\mathfrak{s}}) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

returns the lowest common named super type — ty — of the types \mathfrak{t} and \mathfrak{s} in tenv .

The helper function

$$\text{supers}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\mathfrak{t}}) \longrightarrow \mathcal{P}(\text{ty})$$

returns the set of *named supertypes* given via the `subtypes` function of the global environment:

$$\text{supers}(\text{tenv}, \mathfrak{t}) \triangleq \begin{cases} \{\mathfrak{t}\} \cup \text{supers}(\mathfrak{s}) & \text{if } G^{\text{tenv}}.\text{subtypes}(\mathfrak{t}) = \mathfrak{s} \\ \{\mathfrak{t}\} & \text{otherwise (that is, } G^{\text{tenv}}.\text{subtypes}(\mathfrak{t}) = \perp) \end{cases}$$

8.10.1 Prose

One of the following holds:

- `t_supers` is in the set of named supertypes of `t`;
- All of the following hold (FOUND):
 - * `s` is in `t_supers`;
 - * `ty` is `s`;
- All of the following hold (SUPER):
 - * `s` is not in `t_supers`;
 - * `s` has a named super type in `tenv` — `s'`;
 - * `ty` is the lowest common named supertype of `t` and `s'` in `tenv`.
- All of the following hold (NONE):
 - * `s` is not in `t_supers`;
 - * `s` has no named super type in `tenv`;
 - * `ty` is `None`.

8.10.2 Formally

$$\begin{array}{c}
 \text{FOUND} \\
 \frac{\text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \in t_supers}{\text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} s} \\
 \\
 \text{SUPER} \\
 \frac{G^{\text{tenv}}.\text{subtypes}(s) = s' \quad \text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \quad \text{named_lowest_common_ancestor}(\text{tenv}, t, s') \xrightarrow{\text{type}} ty}{\text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} ty} \\
 \\
 \text{NONE} \\
 \frac{\text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \quad G^{\text{tenv}}.\text{subtypes}(s) = \text{None}}{\text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{None}}
 \end{array}$$

8.11 TypingRule.AnnotateConstraintBinop

The function

$$\text{annotate_constraint_binop}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \underbrace{\text{int_constraints} \cup \text{\#TE}}_{\text{ics}}$$

annotates the application of the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding an integer constraints element `ics`. Otherwise, the result is a type error.

The operator `op` is assumed to be only one of the operators in the following set: `{SHL, SHR, POW, MOD, DIVRM, MINUS, MUL, PLUS, DIV}`.

Annotating the constraints involves applying symbolic reasoning and in particular filtering out values that will definitely result in a dynamic error.

8.11.1 Prose

All of the following apply:

- applying *binop_filter_rhs* to `op cs2` in `tenv`, to filter out constraints that will definitely fail dynamically, yields `cs2_f`;
- applying *binop_is_exploding* to `op` yields `b_exploding`;
- applying *explode_intervals* to `cs1` in `tenv` yields `cs1_e`;
- applying *explode_intervals* to `cs2` in `tenv` yields `cs2`;
- define `(cs1_arg, cs2_arg)` as `(cs1_e, cs2_e)` if `b_exploding` is `TRUE` and `(cs1, cs2_f)`, otherwise;
- applying *constraint_binop* to `op`, `cs1_arg`, and `cs2_arg` yields `cs_vanilla`;
- applying *reduce_constraints* to `cs_vanilla` in `tenv` yields `cs`;
- one of the following applies:
 - * All of the following apply (DIV_WELLCONSTRAINED):
 - `op` is `DIV` and `cs` is a `WellConstrained` constraint;
 - view `cs` as `WellConstrained(cs_list)`;
 - applying *refine_constraints* to *filter_reduce_constraint_div* and `cs_list` yields `cs_list_filtered`;
 - applying *reduce_constants* to `cs_list_filtered` yields `ics`.
 - * All of the following apply (ELSE):
 - either `op` is not `DIV` or `cs` is not a `WellConstrained` constraint;
 - `ics` is `cs`.

8.11.2 Formally

DIV_WELLCONSTRAINED

$$\begin{array}{c}
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \quad \text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{b_exploding} \\
\text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} \text{cs1_e} \quad \text{explode_intervals}(\text{tenv}, \text{cs2_f}) \xrightarrow{\text{type}} \text{cs2_e} \\
(\text{cs1_arg}, \text{cs2_arg}) := \text{choice}(\text{b_exploding}, (\text{cs1_e}, \text{cs2_e}), (\text{cs1}, \text{cs2_f})) \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{reduce_constraints}(\text{tenv}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{cs} \\
\text{***** common prefix *****} \\
\text{op} = \text{DIV} \wedge \text{ast_label}(\text{cs}) = \text{WellConstrained} \quad \text{cs} \stackrel{\text{is}}{=} \text{WellConstrained}(\text{cs_list}) \\
\text{refine_constraints}(\text{filter_reduce_constraint_div}, \text{cs_list}) \xrightarrow{\text{type}} \text{cs_list_filtered} \\
\text{reduce_constraints}(\text{cs_list_filtered}) \xrightarrow{\text{type}} \text{ics} \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{ics}
\end{array}$$

ELSE

$$\begin{array}{c}
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \quad \text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{b_exploding} \\
\text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} \text{cs1_e} \quad \text{explode_intervals}(\text{tenv}, \text{cs2_f}) \xrightarrow{\text{type}} \text{cs2_e} \\
(\text{cs1_arg}, \text{cs2_arg}) := \text{choice}(\text{b_exploding}, (\text{cs1_e}, \text{cs2_e}), (\text{cs1}, \text{cs2_f})) \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{reduce_constraints}(\text{tenv}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{cs} \\
\text{***** common prefix *****} \\
\neg(\text{op} = \text{DIV} \wedge \text{ast_label}(\text{cs}) = \text{WellConstrained}) \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{ics}}
\end{array}$$

8.12 TypingRule.BinopFilterRhs

The function

$$\text{binop_filter_rhs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

8.12.1 Prose

One of the following applies:

- All of the following apply (GREATER_OR_EQUAL):
 - * `op` is one of `SHL`, `SHR`, and `POW`;
 - * define `f` as the specialization of *refine_constraint_by_sign* for the predicate $\lambda x. x \geq 0$, which is `TRUE` if and only if the tested number is greater or equal to 0;

- * refining the list of constraints `cs` with `p` via *refine_constraints* yields `new_cs`;
 - * checking whether `new_cs` is empty yields `TRUE` *TE.OFC*.
- All of the following apply (GREATER_THAN):
 - * `op` is one of MOD, DIV, and DIVRM;
 - * define `f` as the specialization of *refine_constraint_by_sign* for the predicate $\lambda x. x > 0$, which is `TRUE` if and only if the tested number is greater than 0;
 - * refining the list of constraints `cs` with `p` via *refine_constraints* yields `new_cs`;
 - * checking whether `new_cs` is empty yields `TRUE` *TE.OFC*.
 - All of the following apply (NO_FILTERING):
 - * `op` is one of MINUS, MUL, and PLUS;
 - * `new_cs` is `cs`.

8.12.2 Formally

GREATER_OR_EQUAL

$$\frac{\begin{array}{c} \text{op} \in \{\text{SHL}, \text{SHR}, \text{POW}\} \\ f := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x \geq 0, c) \\ \text{refine_constraints}(\text{cs}, p) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE.OFC}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

GREATER_THAN

$$\frac{\begin{array}{c} \text{op} \in \{\text{MOD}, \text{DIV}, \text{DIVRM}\} \\ f := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x > 0, c) \\ \text{refine_constraints}(\text{cs}, p) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE.OFC}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

NO_FILTER

$$\frac{\text{op} \in \{\text{MINUS}, \text{MUL}, \text{PLUS}\}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{new_cs}}}$$

8.13 TypingRule.RefineConstraintBySign

The function

$$\text{refine_constraint_by_sign}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z} \rightarrow \mathbb{B}}^p, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\langle \text{int_constraint} \rangle}^{\text{c_opt}}$$

takes a predicate p that returns **TRUE** based on the sign of its input. The function conservatively refines the constraint c in tenv by applying symbolic reasoning to yield a new constraint (inside an optional) that represents the values that satisfy the c and for which p holds. In this context, conservatively means that the new constraint may represent a superset of the values that a more precise reasoning may yield. If the set of those values is empty the result is **None**.

8.13.1 Prose

One of the following applies:

- All of the following apply (**EXACT_REDUCES_TO_Z**):
 - * c is an exact constraint for the expression e , that is, `Constraint.Exact(e)`;
 - * applying *reduce_to_z_opt* to e in tenv , in order to symbolically simplify e to an integer, yields $\langle z \rangle$;
 - * c_opt is $\langle c \rangle$ if p holds for z and **None** otherwise.
- All of the following apply (**EXACT_DOES_NOT_REDUCE_TO_Z**):
 - * c is an exact constraint for the expression e , that is, `Constraint.Exact(e)`;
 - * applying *reduce_to_z_opt* to e in tenv , in order to symbolically simplify e to an integer, yields **None**;
 - * c_opt is $\langle c \rangle$.
- All of the following apply (**RANGE_BOTH_REDUCE_TO_Z**):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, `Constraint.Range(e1, e2)`;
 - * applying *reduce_to_z_opt* to $e1$ in tenv , in order to symbolically simplify $e1$ to an integer, yields $\langle z1 \rangle$;
 - * applying *reduce_to_z_opt* to $e2$ in tenv , in order to symbolically simplify $e2$ to an integer, yields $\langle z2 \rangle$;
 - * One of the following applies (defining c_opt):
 - if p is **TRUE** for both $z1$ and $z2$, define c_opt as $\langle c \rangle$;
 - if p is **FALSE** for $z1$ and **TRUE** for $z2$, define c_opt as the optional range constraint where the bottom expression is the literal expression for 0 if p holds for 0 and the literal expression for 1 otherwise, and the top expression is $e2$;
 - if p is **TRUE** for $z1$ and **FALSE** for $z2$, define c_opt as the optional range constraint where the bottom expression is $e1$ and the top expression is the literal expression for 0 if p holds for 0 and the literal expression for -1 otherwise;
 - if p is **FALSE** for both $z1$ and $z2$, define c_opt as **None**.

- All of the following apply (`ONLY_E1_REDUCES_TO_Z`):
 - * `c` is a range constraint for the expressions `e1` and `e2`, that is, `Constraint.Range(e1, e2)`;
 - * applying `reduce_to_z_opt` to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields `<z1>`;
 - * applying `reduce_to_z_opt` to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `None`;
 - * One of the following applies (defining `c_opt`):
 - if `p` is `TRUE` for `z1`, define `c_opt` as `<c>`;
 - if `p` is `FALSE` for `z1`, define `c_opt` as the optional range constraint with the bottom expression as the literal expression for 0 if `p` holds for 0 and the literal expression for 1 otherwise, and the top expression `e2`.
- All of the following apply (`ONLY_E2_REDUCES_TO_Z`):
 - * `c` is a range constraint for the expressions `e1` and `e2`, that is, `Constraint.Range(e1, e2)`;
 - * applying `reduce_to_z_opt` to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields `None`;
 - * applying `reduce_to_z_opt` to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `<z2>`;
 - * One of the following applies (defining `c_opt`):
 - if `p` is `TRUE` for `z2`, define `c_opt` as `<c>`;
 - if `p` is `FALSE` for `z2`, define `c_opt` as the optional range constraint with the bottom expression `e1` and the top expression the literal expression for 0 if `p` holds for 0 and the literal expression for `-1` otherwise.

8.13.2 Formally

$$\begin{array}{c}
 \text{EXACT_REDUCES_TO_Z} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \langle z \rangle \quad c_opt := \text{choice}(p(z), \langle c \rangle, \text{None})}{\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} c_opt} \\
 \\
 \text{EXACT_DOES_NOT_REDUCE_TO_Z} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None}}{\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt}}
 \end{array}$$

RANGE_BOTH_REDUCE_TO_Z

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle \\
\text{c_opt} := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z1) \wedge p(z2) \\ \langle \text{Constraint_Range}(\text{choice}(p(0), \overset{\text{E_Literal(L.Int)}{0}, \overset{\text{E_Literal(L.Int)}{1}}{1}}, e2) \rangle & \text{if } \neg p(z1) \wedge p(z2) \\ \langle \text{Constraint_Range}(e1, \text{choice}(p(0), \overset{\text{E_Literal(L.Int)}{0}, \overset{\text{E_Literal(L.Int)}{-1}}{-1}})) \rangle & \text{if } p(z1) \wedge \neg p(z2) \\ \text{None} & \text{if } \neg p(z1) \wedge \neg p(z2) \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \text{c_opt}
\end{array}$$

ONLY_E1_REDUCES_TO_Z

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
\text{c_opt} := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z1) \\ \langle \text{Constraint_Range}(\text{choice}(p(0), \overset{\text{E_Literal(L.Int)}{0}, \overset{\text{E_Literal(L.Int)}{1}}{1}}, e2) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \text{c_opt}
\end{array}$$

ONLY_E2_REDUCES_TO_Z

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle \\
\text{c_opt} := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z2) \\ \langle \text{Constraint_Range}(e1, \text{choice}(p(0), \overset{\text{E_Literal(L.Int)}{0}, \overset{\text{E_Literal(L.Int)}{-1}}{-1}})) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \text{c_opt}
\end{array}$$

NONE_REDUCE_TO_Z

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{c_opt}}^c
\end{array}$$

8.14 TypingRule.ReduceToZOpt

The function

$$\text{reduce_to_z_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{\text{z_opt}}$$

returns an integer inside an optional if e can be symbolically simplified into an integer in tenv and `None` otherwise. The expression e is assumed to appear in a constraint for a type that has been successfully annotated, which means that applying *normalize* to it should not yield a type error.

8.14.1 Prose

One of the following applies:

- All of the following apply (`NORMALIZES_TO_Z`):
 - * symbolically simplifying e in tenv via *normalize* yields a literal expression for the integer z ;
 - * define `z_opt` as $\langle z \rangle$.
- All of the following apply (`DOES_NOT_NORMALIZE_TO_Z`):
 - * symbolically simplifying e in tenv via *normalize* yields an expression that is not an integer literal;
 - * define `z_opt` as `None`.

8.14.2 Formally

$$\begin{array}{c}
 \text{NORMALIZES_TO_Z} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \frac{\text{E_Literal(L_Int)}}{\mathbb{Z}} \\
 \hline
 \text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\langle z \rangle}_{\text{z_opt}}
 \end{array}$$

$$\begin{array}{c}
 \text{DOES_NOT_NORMALIZE_TO_Z} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e' \quad \forall z \in \mathbb{Z}. e' \neq \frac{\text{E_Literal(L_Int)}}{\mathbb{Z}} \\
 \hline
 \text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\text{None}}_{\text{z_opt}}
 \end{array}$$

8.15 TypingRule.RefineConstraints

The function

$$\text{refine_constraints}(\underbrace{\text{SE}}_{\text{tenv}}, \underbrace{\text{int_constraint} \rightarrow \langle \text{int_constraint} \rangle}_{\text{f}}, \underbrace{\text{int_constraint}^*}_{\text{cs}} \rightarrow \underbrace{\text{int_constraint}^*}_{\text{new_cs}}$$

refines a list of constraints `cs` by applying the refinement function `f` to each constraint and retaining the constraints that do not refine to `None` in tenv . The resulting list of constraints is given in `new_cs`.

8.15.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `cs` is the empty list;
 - * `new_cs` is the empty list.
- All of the following apply (NON_EMPTY_NONE):
 - * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
 - * applying `f` to `c` in `tenv` yields `None`;
 - * applying *refine_constraints* to `f` and `cs1` in `tenv` yields `cs1'`;
 - * `new_cs` is `cs1'`.
- All of the following apply (NON_EMPTY_SOME):
 - * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
 - * applying `f` to `c` in `tenv` yields `<c'>`;
 - * applying *refine_constraints* to `f` and `cs1` in `tenv` yields `cs1'`;
 - * `new_cs` is the list with `c'` as its `head` and `cs1'` as its `tail`.

8.15.2 Formally

EMPTY

$$\text{refine_constraints}(\text{tenv}, f, \overbrace{[]^{\text{cs}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{new_cs}}}$$

NON_EMPTY_NONE

$$\frac{f(\text{tenv}, c) \xrightarrow{\text{type}} \text{None} \quad \text{refine_constraints}(\text{tenv}, f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}'}{\text{refine_constraints}(\text{tenv}, f, \overbrace{[c] + \text{cs1}}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{\text{cs1}'}^{\text{new_cs}}}$$

NON_EMPTY_SOME

$$\frac{f(\text{tenv}, c) \xrightarrow{\text{type}} \langle c' \rangle \quad \text{refine_constraints}(\text{tenv}, f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}'}{\text{refine_constraints}(\text{tenv}, f, \overbrace{[c] + \text{cs1}}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{[c'] + \text{cs1}'}^{\text{new_cs}}}$$

8.16 TypingRule.FilterReduceConstraintDiv

The function

$$\text{filter_reduce_constraint_div}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\langle \text{int_constraint} \rangle}^{c_opt}$$

returns **None** if c is an exact constraint for a binary expression for dividing two integer literals where the denominator does not divide the numerator and an optional containing c . The result is returned in c_opt . This is used to conservatively test whether c would always fail dynamically.

8.16.1 Prose

If c is an exact constraint for a binary expression for the division operation and two integer literal expressions for the integers $z1$ and $z2$ such that $z2$ does not divide $z1$ then c_opt is **None**. Otherwise c_opt is $\langle c \rangle$.

8.16.2 Formally

$$\begin{array}{c} \text{EXACT_DIV_LITERALS} \\ \hline c = \text{Constraint_Exact}(e) \quad \text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \text{None} \\ \hline \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt} \\ \\ \text{EXACT_NOT_DIV_LITERALS} \\ \hline c = \text{Constraint_Exact}(e) \\ \text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \langle \langle z1, z2 \rangle \rangle c_opt := \text{choice}(\frac{z1}{z2} \in \mathbb{Z}, \langle c \rangle, \text{None}) \\ \hline \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} c_opt \\ \\ \text{RANGE} \\ \hline \text{ast_label}(c) = \text{Constraint_Range} \\ \hline \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt} \end{array}$$

8.17 TypingRule.GetLiteralDivOpt

The function

$$\text{get_literal_div_opt}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \times \mathbb{Z} \rangle}^{\text{range_opt}}$$

matches the expression e to a binary operation expression over the division operation and two literal integer expressions. If e matches this pattern the result range_opt is an optional containing the pair of integers appearing in the operand expressions. Otherwise, the result is **None**.

8.17.1 Prose

The value `range_opt` is $\langle\langle z1, z2 \rangle\rangle$ if `e` is a binary operation expression over the division operation and two literal integer expressions for the integers `z1` and `z2` and `None` otherwise.

8.17.2 Formally

$$\frac{\text{range_opt} := \text{choice}(e = \text{E.Binop}(\text{DIV}, \overset{\text{E.Literal(L.Int)}}{\boxed{z1}}, \overset{\text{E.Literal(L.Int)}}{\boxed{z2}}), \langle\langle z1, z2 \rangle\rangle, \text{None})}{\text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \text{range_opt}}$$

8.18 TypingRule.ExplodeIntervals

The function

$$\text{explode_intervals}(\overset{\text{tenv}}{\boxed{\text{SE}}}, \overset{\text{cs}}{\text{int_constraint}^*}) \longrightarrow \overset{\text{new_cs}}{\text{int_constraint}^*}$$

applies `exploded_interval` to each constraint of `cs` in `tenv` and concatenates the resulting list, yielding the result in `new_cs`.

8.18.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `cs` is the empty list;
 - * `new_cs` is the empty list.
- All of the following apply (NON_EMPTY):
 - * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
 - * applying `explode_constraint` to `c` in `tenv` yields `c'` (a list of constraints);
 - * applying `explode_intervals` to `cs1` in `tenv` yields `cs1'`;
 - * `new_cs` is the concatenation of `c'` and `cs1'`.

8.18.2 Formally

$$\frac{\text{EMPTY}}{\text{explode_intervals}(\text{tenv}, \overset{\text{cs}}{\boxed{[]}}) \xrightarrow{\text{type}} \overset{\text{new_cs}}{\boxed{[]}}}$$

$$\frac{\text{NON_EMPTY} \quad \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} c' \quad \text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} \text{cs1'}}{\text{explode_intervals}(\text{tenv}, \overset{\text{cs}}{[c] + \text{cs1}}) \xrightarrow{\text{type}} \overset{\text{new_cs}}{c' + \text{cs1}'}}$$

8.19 TypingRule.ExplodeConstraint

The function

$$\text{explode_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{vcs}}$$

expands the constraint c into the equivalent list of exact constraints if c matches a n ascending range constraint that is not too large in tenv and the singleton list for c otherwise.

8.19.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an exact constraint;
 - * vcs is the singleton list for c .
- All of the following apply (RANGE_REDUCED):
 - * c is a range constraint for the expressions a and b ;
 - * applying *reduce_to_z_opt* to a in tenv yields $\langle \text{za} \rangle$;
 - * applying *reduce_to_z_opt* to b in tenv yields $\langle \text{zb} \rangle$;
 - * define `exploded_interval` as the list of exact constraints for each integer literal in the range starting at za and ending at zb , inclusively, which is empty if $\text{zb} < \text{za}$;
 - * applying *interval_too_large* to za and zb yields b_too_large ;
 - * define vcs as the singleton list for c if b_too_large is `TRUE` and `exploded_interval` otherwise.
- All of the following apply (RANGE_NOT_REDUCED):
 - * c is a range constraint for the expressions a and b ;
 - * applying *reduce_to_z_opt* to a in tenv yields za_opt ;
 - * applying *reduce_to_z_opt* to b in tenv yields zb_opt ;
 - * at least one of za_opt and zb_opt is `None`;
 - * vcs is the singleton list for c .

8.19.2 Formally

$$\begin{array}{c}
\text{EXACT} \\
\hline
\text{ast_label}(c) = \text{Constraint.Exact} \\
\hline
\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{[c]}^{\text{vcs}}
\end{array}$$

$$\begin{array}{c}
\text{RANGE_REDUCED} \\
\hline
c = \text{Constraint.Range}(a, b) \\
\text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \langle \text{za} \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \langle \text{zb} \rangle \\
\text{exploded_interval} := [z \in \text{za}.. \text{zb} : \text{Constraint.Exact}(\text{E.Literal}(\text{L.Int}) \text{ } \overbrace{\text{z}}^{\text{vcs}})] \\
\text{interval_too_large}(\text{za}, \text{zb}) \xrightarrow{\text{type}} \text{b_too_large} \\
\text{vcs} := \text{choice}(\text{b_too_large}, [c], \text{exploded_interval}) \\
\hline
\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \text{vcs}
\end{array}$$

$$\begin{array}{c}
\text{RANGE_NOT_REDUCED} \\
\hline
c = \text{Constraint.Range}(a, b) \quad \text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \text{za_opt} \\
\text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \text{zb_opt} \quad \text{za_opt} = \text{None} \vee \text{zb_opt} = \text{None} \\
\hline
\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{[c]}^{\text{vcs}}
\end{array}$$

8.20 TypingRule.IntervalTooLarge

The function

$$\text{interval_too_large}(\overbrace{\mathbb{Z}}^{z1}, \overbrace{\mathbb{Z}}^{z2}) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether the set of numbers between $z1$ and $z2$, inclusive, contains more than 2^{14} integers, yielding the result in b .

8.20.1 Prose

The value b is **TRUE** if and only if the absolute value of $z1 - z2$ is greater than 2^{14} .

8.20.2 Formally

$$\text{interval_too_large}(z1, z2) \xrightarrow{\text{type}} \overbrace{|\text{z1} - \text{z2}| > 2^{14}}^b$$

8.21 TypingRule.BinopIsExploding

The function

$$\text{binop_is_exploding}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^b$$

determines whether the binary operation `op` should lead to applying *explode_intervals* when the `op` is applied to a pair of constraint lists. It is assumed that `op` is one of MUL, SHL, POW, PLUS, DIV, MINUS, MOD, SHR, and DIVRM.

8.21.1 Prose

The value `b` is `TRUE` if and only if `op` is one of MUL, SHL, and POW.

8.21.2 Formally

$$\textit{binop_is_exploding}(\textit{op}) \xrightarrow{\textit{type}} \overbrace{\textit{op} \in \{\textit{MUL}, \textit{SHL}, \textit{POW}\}}^{\textit{b}}$$

Chapter 9

Typing of Types

The function

$$\text{annotate_type}(\overbrace{\mathbb{B}}^{\text{decl}}, \overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{new_ty}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a type `ty` in an environment `tenv`, resulting in a annotated type `new_ty`. The flag `decl` indicates whether `ty` is a type currently being declared. Otherwise, the result is a type error.

The flag `decl` makes a difference only when `ty` is an enumeration type or a [structured type](#).

One of the following applies:

- `TypingRule.TString` (see [Section 9.1](#));
- `TypingRule.TReal` (see [Section 9.2](#));
- `TypingRule.TBool` (see [Section 9.3](#));
- `TypingRule.TNamed` (see [Section 9.4](#));
- `TypingRule.TInt` (see [Section 9.5](#));
- `TypingRule.TBits` (see [Section 9.6](#));
- `TypingRule.TTuple` (see [Section 9.7](#));
- `TypingRule.TArray` (see [Section 9.8](#));
- `TypingRule.TEnumDecl` (see [Section 9.9](#));
- `TypingRule.TRecordExceptionDecl` (see [Section 9.10](#));
- `TypingRule.TNonDecl` (see [Section 9.11](#));

Otherwise, the result is a type error.

We also define the following helper functions:

- `TypingRule.AnnotateConstraint` (see Section 9.12)
- `TypingRule.GetVariableEnum` (see Section 9.13)
- `TypingRule.AnnotateStaticInteger` (see Section 9.14)

9.1 TypingRule.TString

9.1.1 Prose

All of the following apply:

- `ty` is the string type `T.String`.
- `new_ty` is the string type `T.String`.

9.1.2 Example

In the following example, all the uses of `string` are valid:

```
type MyType of string;

func foo (x: string) => string
begin
  return x;
end

func main () => integer
begin
  var x: string;

  x = "foo";
  x = foo (x as string);

  let y: string = x;

  assert x as string == x;

  return 0;
end
```

9.1.3 Formally

$$\text{annotate_type}(\overbrace{\quad}^{\text{decl}}, \text{tenv}, \overbrace{\text{T.String}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T.String}}^{\text{new_ty}}$$

9.2 TypingRule.TReal

9.2.1 Prose

All of the following apply:

- `ty` is the real type `T_Real`.
- `new_ty` is the real type `T_Real`.

9.2.2 Example

In the following example, all the uses of `real` are valid:

```
type MyType of real;

func foo (x: real) => real
begin
  return x + 1.0;
end

func main () => integer
begin
  var x: real;

  x = 3.141592;
  x = foo (x as real);

  let y: real = x + x;

  assert x as real == x;

  return 0;
end
```

9.2.3 Formally

$$\text{annotate_type}(\overbrace{\text{—}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Real}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Real}}^{\text{new_ty}}$$

9.3 TypingRule.TBool

9.3.1 Prose

All of the following apply:

- `ty` is the boolean type, `T_Bool`;
- `new_ty` is the boolean type, `T_Bool`.

9.3.2 Example

In the following example, all the uses of `boolean` are valid:

```
type MyType of boolean;

func foo (x: boolean) => boolean
begin
  return FALSE --> x;
end

func main () => integer
begin
  var x: boolean;

  x = TRUE;
  x = foo (x as boolean);

  let y: boolean = x && x;

  assert x as boolean == x;

  return 0;
end
```

9.3.3 Formally

$$\text{annotate_type}(\overbrace{\text{---}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Bool}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^{\text{new_ty}}$$

9.4 TypingRule.TNamed

9.4.1 Prose

All of the following apply:

- `ty` is the named type `x`, that is `T_Named(x)`;
- checking whether `x` is associated with a declared type in `tenv` yields `TRUE//#TE`;
- `new_ty` is `ty`.

9.4.2 Example

In the following example, all the uses of `MyType` are valid:

```
type MyType of integer;

func foo (x: MyType) => MyType
```



```

begin
  return x;
end

func main () => integer
begin
  var x: MyType;

  x = 4;
  x = foo (x as MyType);

  let y: MyType = x;

  assert x as MyType == x;

  return 0;
end

```

9.4.3 Formally

$$\frac{\text{check}(G^{\text{tenv}}(x) \neq \perp, \text{TE_UI}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE}}{\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \overbrace{\text{T_Named}(x)}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Named}(x)}^{\text{new_ty}}}$$

9.5 TypingRule.TInt

9.5.1 Prose

One of the following applies:

- All of the following apply (NOT_WELL_CONSTRAINED):
 - * `ty` is an integer type that is not well-constrained;
 - * `new_ty` is the unconstrained integer type.
- All of the following apply (WELL_CONSTRAINED):
 - * `ty` is the well-constrained integer type constrained by constraints c_i , for $u = 1..k$;
 - * annotating each constraint c_i , for $i = 1..k$, yields `new_ci` $\parallel \# \text{TE}$;
 - * `new_constraints` is the list of annotated constraints `new_ci`, for $i = 1..k$;
 - * `new_ty` is the well-constrained integer type constrained by `new_constraints`.

9.5.2 Example

In the following examples, all the uses of integer types are valid:

```
type MyType of integer;
func foo (x: integer) => integer
begin
  return x;
end
```

```
func main () => integer
begin
  var x: integer;

  x = 4;
  x = foo (x as integer);

  let y: integer = x;

  assert x as integer == x;

  return 0;
end
```

```
type MyType of integer {1..12};

func foo (x: integer {1..12}) => integer {1..12}
begin
  return x;
end
```

```
func main () => integer
begin
  var x: integer {1..12};

  x = 4;
  x = foo (x as integer {1..12});

  let y: integer {1..12} = x;

  assert x as integer {1..11} == x;

  return 0;
end
```

```
func foo {N} (x: bits(N)) => integer
begin
  return N;
```

```

end

func bar (N: integer) => bits(N)
begin
  return Zeros(N);
end

func main() => integer
begin
  assert 3 == foo ('101');
  assert bar(3) == '000';

  return 0;
end

```

9.5.3 Formally

$$\begin{array}{c}
\text{NOT_WELL_CONSTRAINED} \\
\text{ty} \stackrel{\text{is}}{=} \text{T_Int}(c) \quad \text{ast_label}(c) \neq \text{WellConstrained} \\
\hline
\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \underbrace{\text{ty}}_{\text{new_ty}} \\
\\
\text{WELL_CONSTRAINED} \\
\text{constraints} \stackrel{\text{is}}{=} c_{1..k} \quad i = 1..k : \text{annotate_constraint}(c_i) \xrightarrow{\text{type}} \text{new_c}_i \text{ // \#TE} \\
\text{new_constraints} := \text{new_c}_{1..k} \\
\hline
\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \underbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}_{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\text{T_Int}(\text{WellConstrained}(\text{new_constraints}))}_{\text{new_ty}}
\end{array}$$

9.6 TypingRule.TBits

9.6.1 Prose

All of the following apply:

- `ty` is the bit-vector type with width given by the expression `e_width` and the bitfields given by `bitfields`, that is, `T_Bits(e_width, bitfields)`;
- annotating the **statically evaluable** integer expression `e_width` yields `e_width' // #TE`;
- annotating the bitfields `bitfields` yields `bitfields' // #TE`;
- `new_ty` is the bit-vector type with width given by the expression `e_width'` and the bitfields given by `bitfields'`, that is, `T_Bits(e_width', bitfields')`

9.6.2 Example

```

type MyType of bits(4);

func foo (x: bits(4)) => bits(4)
begin
  return NOT x;
end

func main () => integer
begin
  var x: bits(4);

  x = '1010';
  x = foo (x as bits(4));

  let y: bits(4) = x;

  assert x as bits(4) == x;

  return 0;
end

```

In the following example, all the uses of bitvector types are valid:

9.6.3 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_static_constrained_integer}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} \text{e_width}' \quad // \text{ \#TE} \\
 \text{annotate_bitfields}(\text{tenv}, \text{e_width}', \text{bitfields}) \xrightarrow{\text{type}} \text{bitfields}' \quad // \text{ \#TE}
 \end{array}
 }{
 \text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \text{T_Bits}(\text{e_width}, \text{bitfields})) \xrightarrow{\text{type}} \text{T_Bits}(\text{e_width}', \text{bitfields}')
 }$$

9.6.4 Comments

The width of a bitvector type $\text{T_Bits}(\text{e_width}, \text{bitfields})$, given by the expression e_width , must be non-negative.

9.7 TypingRule.TTuple

9.7.1 Prose

All of the following apply:

- ty is the tuple type with member types tys , that is, $\text{T_Tuple}(\text{tys})$;
- tys is the list ty_i , for $i = 1..k$;

- annotating each type ty_i in tenv , for $i = 1..k$, yields ty'_i *//* #TE;
- new_ty is the tuple type with member types ty' , for $i = 1..k$.

In the following example, all the uses of tuple types are valid:

```
type MyType of (integer, boolean);

func foo (x: (integer, boolean)) => (integer, boolean)
begin
  let (z, y): (integer, boolean) = x;
  return (z + 1, FALSE --> y);
end

func main () => integer
begin
  var x: (integer, boolean);

  x = (3, TRUE);
  x = foo (x as (integer, boolean));

  let y: (integer, boolean) = x;

  let (x0, x1) = x as (integer, boolean);
  assert x0 == 4 && x1 == TRUE;

  return 0;
end
```

9.7.2 Example

9.7.3 Formally

$$\frac{k \geq 2 \quad \text{tys} \stackrel{\text{is}}{=} \text{ty}_{1..k} \quad i = 1..k : \text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}_i) \xrightarrow{\text{type}} \text{ty}'_i \quad \text{//} \quad \text{\#TE}}{\text{annotate_type}(\underbrace{\text{tys}}^{\text{decl}}, \text{tenv}, \text{T_Tuple}(\text{tys})) \xrightarrow{\text{type}} \text{T_Tuple}(\text{tys}')}$$

9.8 TypingRule.TArray

9.8.1 Prose

All of the following apply:

- ty is the array type with element type t ;
- Annotating the type t in tenv yields t' *//* #TE;
- One of the following applies:
 - * All of the following apply (EXPR_IS_ENUM):

- the array index is **e** and determining whether **e** corresponds to an enumeration in **tenv** via `get_variable_enum` yields the enumeration variable name **s** of size **i**, that is, $\langle \mathbf{s}, \mathbf{i} \rangle \#_{TE}$;
 - **new_ty** is the array type indexed by an enumeration type named **s** of length **i** and of elements of type **t'**, that is, $T_Array(ArrayLength_Enum(\mathbf{s}, \mathbf{i}), \mathbf{t}')$.
- * All of the following apply (`EXPR_NOT_ENUM`):
- the array index is **e** and determining whether **e** corresponds to an enumeration in **tenv** via `get_variable_enum` yields **None** (meaning it does not correspond to an enumeration) $\#_{TE}$;
 - annotating the statically evaluable integer expression **e** yields $\mathbf{e}' \#_{TE}$;
 - **new_ty** the array type indexed by integer bounded by the expression **e'** and of elements of type **t'**, that is, $T_Array(ArrayLength_Expr(\mathbf{e}'), \mathbf{t}')$.
- * All of the following apply (`INDEX_ENUM`):
- the array index is an enumeration type named **s** and a list of **i** labels, that is, $ArrayLength_Enum(\mathbf{s}, \mathbf{i})$;
 - let **ty** be the named type defined for **s**, that is, $T_Named(\mathbf{s})$;
 - determining the **underlying type** of **ty** yields $\mathbf{t_s_anon} \#_{TE}$;
 - checking whether **t_s_anon** is an enumeration type yields $TRUE \#_{TE}$;
 - **t** is an enumeration with labels **li**;
 - checking whether **li** has the same length as **i** yields $TRUE \#_{TE}$;
 - **new_ty** is the array type indexed by an enumeration type named **s** of length **i** and of elements of type **t'**, that is, $T_Array(ArrayLength_Enum(\mathbf{s}, \mathbf{i}), \mathbf{t}')$.

9.8.2 Example

In the following example, all the uses of array types are valid:

```
type MyType of array [4] of integer;

func foo (x: array [4] of integer) => array [4] of integer
begin
  var y = x;
  y[3] = 2;
  return y;
end

func main () => integer
begin
  var x: array [4] of integer;

  x[1] = 1;
  print(x);
  x = foo (x as array [4] of integer);
```

```

let y: array [4] of integer = x;

return 0;
end

```

9.8.3 Formally

EXPR_IS_ENUM

$$\begin{array}{c}
\text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} t' \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \langle s, i \rangle \quad // \quad \#TE \\
\hline
\text{annotate_type}(\underbrace{\text{decl}}_{\text{---}}, \text{tenv}, \text{array } [e] \text{ of } t) \xrightarrow{\text{type}} \text{array } [e\#i] \text{ of } t'
\end{array}$$

$\text{ty} \quad \text{new_ty}$
 $\text{T_Array}(\text{ArrayLength_Expr}) \quad \text{T_Array}(\text{ArrayLength_Enum})$

EXPR_NOT_ENUM

$$\begin{array}{c}
\text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} t' \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\
\text{annotate_static_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e' \quad // \quad \#TE \\
\hline
\text{annotate_type}(\underbrace{\text{decl}}_{\text{---}}, \text{tenv}, \text{array } [e] \text{ of } t) \xrightarrow{\text{type}} \text{array } [e'] \text{ of } t'
\end{array}$$

$\text{ty} \quad \text{new_ty}$
 $\text{T_Array}(\text{ArrayLength_Expr}) \quad \text{T_Array}(\text{ArrayLength_Expr})$

INDEX_ENUM

$$\begin{array}{c}
\text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} t' \quad // \quad \#TE \\
\text{***** common prefix *****} \\
t_s := T_Named(s) \quad \text{make_anonymous}(\text{tenv}, t_s) \xrightarrow{\text{type}} t_s_anon \quad // \quad \#TE \\
\text{check}(\text{ast_label}(t_s_anon) = T_Enum, TE_EET) \rightarrow \text{TRUE} \quad // \quad \#TE \\
t_s_anon \stackrel{\text{is}}{=} T_Enum(li) \\
\text{check}(\text{equal_length}(li, i), \text{TypeConflict}) \rightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_type}(\underbrace{\text{decl}}_{\text{---}}, \text{tenv}, \text{array } [s\#i] \text{ of } t) \xrightarrow{\text{type}} \text{array } [s\#i] \text{ of } t'
\end{array}$$

$\text{ty} \quad \text{new_ty}$
 $\text{T_Array}(\text{ArrayLength_Enum}) \quad \text{T_Array}(\text{ArrayLength_Enum})$

9.9 TypingRule.TEnumDecl

9.9.1 Prose

All of the following apply:

- ty is the enumeration type with enumeration literals li , that is, $T_Enum(li)$;

- `decl` is **TRUE**, indicating that `ty` should be considered in the context of a declaration;
- determining that `li` does not contain duplicates yields **TRUE**//**#TE**;
- determining that none of the labels in `li` is declared in the global environment yields **TRUE**//**#TE**;
- `new_ty` is the enumeration type `ty`.

9.9.2 Example

The following example declares a valid enumeration type:

```
type MyEnum of enumeration { A, B, C };
```

9.9.3 Formally

$$\frac{\begin{array}{c} \text{check_no_duplicates}(\text{li}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{li} \in \text{li} : \text{check_var_not_in_gen}(\text{tenv}, \text{li}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \end{array}}{\text{annotate_type}(\text{TRUE}, \text{tenv}, \text{T_Enum}(\text{li})) \xrightarrow{\text{type}} \text{T_Enum}(\text{li})}$$

9.10 TypingRule.TRecordExceptionDecl

9.10.1 Prose

All of the following apply:

- `ty` is a **structured type** with AST label `L`;
- the list of fields of `ty` is `fields`;
- `decl` is **TRUE**, indicating that `ty` should be considered in the context of a declaration;
- `fields` is a list of pairs where the first element is an identifier and the second is a type — (x_i, t_i) , for $i = 1..k$;
- checking that the list of field identifiers $x_{1..k}$ does not contain duplicates yields **TRUE**//**#TE**;
- annotating each field type t_i , for $i = 1..k$, yields an annotated type t'_i //**#TE**;
- `fields'` is the list with (x_i, t'_i) , for $i = 1..k$;
- `new_ty` is the AST node with AST label `L` (either record type or exception type, corresponding to the type `ty`) and fields `fields'`.

9.10.2 Example

In the following example, all the uses of record or exception types are valid:

```
type MyRecord of record { a: integer, b: boolean };
type MyException of exception { a: integer, b: boolean };

func main () => integer
begin return 0; end
```

9.10.3 Formally

$$\frac{
 \begin{array}{c}
 L \in \{\text{T_Record}, \text{T_Exception}\} \\
 \text{fields} \stackrel{\text{is}}{=} [i = 1..k : (x_i, t_i)] \quad \text{check_no_duplicates}(x_{1..k}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 i = 1..k : \text{annotate_type}(\text{FALSE}, \text{tenv}, t_i) \xrightarrow{\text{type}} t'_i \text{ // } \#TE \\
 \text{fields}' := [i = 1..k : (x_i, t'_i)]
 \end{array}
 }{
 \text{annotate_type}(\text{TRUE}, \text{tenv}, L(\text{fields})) \xrightarrow{\text{type}} L(\text{fields}')
 }$$

9.11 TypingRule.TNonDecl

9.11.1 Prose

All of the following apply:

- `ty` is a **structured type** or an enumeration type;
- `decl` is **FALSE**, indicating that `ty` should be considered to be outside the context of a declaration of `ty`;
- a type error is returned, indicating that the use of anonymous form of enumerations, record, and exceptions types is not allowed here.

9.11.2 Example

In the following example, the use of a record type outside of a declaration is erroneous:

```
func (x: record { a: integer, b: boolean }) => integer
begin return 0; end
```

9.11.3 Formally

$$\frac{
 \text{ast_label}(\text{ty}) \in \{\text{T_Enum}, \text{T_Record}, \text{T_Exception}\}
 }{
 \text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_IAF})
 }$$

9.12 TypingRule.AnnotateConstraint

The function

$$\text{annotate_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates an integer constraint c in the static environment tenv yielding the annotated integer constraint new_c . Otherwise, the result is a type error.

9.12.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is the exact integer constraint for the expression e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *annotate_static_constrained_integer* to e in tenv yields $e' \text{ // } \#TE$;
 - * define new_c as the exact integer constraint for e' , that is, $\text{Constraint_Exact}(e')$.
- All of the following apply (RANGE):
 - * c is the range integer constraint for expressions $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *annotate_static_constrained_integer* to $e1$ in tenv yields $e1' \text{ // } \#TE$;
 - * applying *annotate_static_constrained_integer* to $e2$ in tenv yields $e2' \text{ // } \#TE$;
 - * define new_c as the range integer constraint for expressions $e1'$ and $e2'$, that is, $\text{Constraint_Range}(e1', e2')$.

9.12.2 Formally

EXACT

$$\frac{\text{annotate_static_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e' \text{ // } \#TE}{\text{annotate_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}}$$

RANGE

$$\frac{\begin{array}{l} \text{annotate_static_constrained_integer}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \text{ // } \#TE \\ \text{annotate_static_constrained_integer}(\text{tenv}, e2) \xrightarrow{\text{type}} e2' \text{ // } \#TE \end{array}}{\text{annotate_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}}$$

9.13 TypingRule.GetVariableEnum

The function

$$\text{get_variable_enum}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \langle (\overbrace{\text{identifier}}^{\text{x}}, \overbrace{\text{N}}^n) \rangle$$

tests whether the expression e represents a variable of an enumeration type. If so, the result is x — the name of the variable and the number of labels defined for the enumeration type. Otherwise, the result is `None`.

9.13.1 Prose

One of the following applies:

- All of the following apply (`NOT_EVAR`):
 - * e is not a variable expression;
 - * the result is `None`.
- All of the following apply (`NO_DECLARED_TYPE`):
 - * e is a variable expression for x , that is, `E_Var(x)`;
 - * x is not associated with a type in the global environment of `tenv`;
 - * the result is `None`.
- All of the following apply (`DECLARED_ENUM`):
 - * e is a variable expression for x , that is, `E_Var(x)`;
 - * x is associated with a type t in the global environment of `tenv`;
 - * obtaining the `underlying type` of t in `tenv` yields an enumeration type with labels `li` *//* `#TE`;
 - * the result is the pair consisting of x and the length of `li`.
- All of the following apply (`DECLARED_NOT_ENUM`):
 - * e is a variable expression for x , that is, `E_Var(x)`;
 - * x is associated with a type t in the global environment of `tenv`;
 - * obtaining the `underlying type` of t in `tenv` yields a type that is not an enumeration type;
 - * the result is `None`.

9.13.2 Formally

$$\begin{array}{c}
\text{NOT_EVAR} \\
\hline
\text{ast_label}(\mathbf{e}) \neq \mathbf{E_Var} \\
\hline
\text{get_variable_enum}(\text{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \mathbf{None}
\\[10pt]
\text{NO_DECLARED_TYPE} \\
\hline
G^{\text{tenv}}.\text{declared_types}(\mathbf{x}) \xrightarrow{\text{type}} \mathbf{None} \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{\mathbf{E_Var}(\mathbf{x})}^{\mathbf{e}}) \xrightarrow{\text{type}} \mathbf{None}
\\[10pt]
\text{DECLARED_ENUM} \\
\hline
G^{\text{tenv}}.\text{declared_types}(\mathbf{x}) \xrightarrow{\text{type}} \langle \mathbf{t} \rangle \quad \text{make_anonymous}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{T_Enum}(\mathbf{li}) \quad // \quad \mathbf{\#TE} \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{\mathbf{E_Var}(\mathbf{x})}^{\mathbf{e}}) \xrightarrow{\text{type}} \langle \langle \mathbf{x}, |\mathbf{li}| \rangle \rangle
\\[10pt]
\text{DECLARED_NOT_ENUM} \\
\hline
G^{\text{tenv}}.\text{declared_types}(\mathbf{x}) \xrightarrow{\text{type}} \langle \mathbf{t} \rangle \\
\text{make_anonymous}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{t1} \quad \text{ast_label}(\mathbf{t1}) \neq \mathbf{T_Enum} \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{\mathbf{E_Var}(\mathbf{x})}^{\mathbf{e}}) \xrightarrow{\text{type}} \mathbf{None}
\end{array}$$

9.14 TypingRule.AnnotateStaticInteger

The function

$$\text{annotate_static_integer}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{expr}}^{\mathbf{e}}) \longrightarrow \overbrace{\mathbf{expr}}^{\mathbf{e}''} \cup \overbrace{\mathbf{T_TypeError}}^{\mathbf{\#TE}}$$

annotates a **statically evaluable** integer expression \mathbf{e} in the static environment tenv and returns the annotated expression \mathbf{e}'' . Otherwise, the result is a type error.

9.14.1 Prose

All of the following apply:

- annotating the expression \mathbf{e} in tenv yields $(\mathbf{t}, \mathbf{e}') // \mathbf{\#TE}$;
- determining whether \mathbf{t} has the structure of an integer yields $\mathbf{TRUE} // \mathbf{\#TE}$;
- determining whether \mathbf{e}' is **statically evaluable** in tenv yields $\mathbf{TRUE} // \mathbf{\#TE}$;
- applying *normalize* to \mathbf{e}' in tenv yields \mathbf{e}'' .

9.14.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \quad // \quad \#TE \\
\text{check_structure_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{normalize}(\text{tenv}, e') \xrightarrow{\text{type}} e'' \\
\hline
\text{annotate_static_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e''
\end{array}$$

Chapter 10

Typing of Bitfields

We define rules for annotating a single bitfield and a list of bitfields:

- `TypingRule.TBitField` (see Section 10.1);
- `TypingRule.TBitFields` (see Section 10.2);

10.1 `TypingRule.TBitField`

The function

$$\text{annotate_bitfield}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{Z}}^{\text{width}}, \overbrace{\text{bitfield}}^{\text{field}}) \longrightarrow \overbrace{\text{bitfield}}^{\text{new_field}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a bitfield — `field` — with an integer — `width` — indicating the number of bits in the bitvector type that contains `field`, in an environment `tenv`, resulting in an annotated bitfield — `new_field` — or a type error, if one is detected.

10.1.1 Prose

- `field` is a bitfield with list of slices `slices`;
- annotating the slices `slices` yields `slices1//\#TE`;
- One of the following applies:
 - * All of the following apply (SIMPLE):
 - checking whether the range of positions in `slices1` fits inside `0..width-1` yields `TRUE//\#TE`;
 - `new_field` is a bitfield named `name` with list of slices `slices1`, that is, `BitField.Simple(name, slices1)`.
 - * All of the following apply (NESTED):

- converting the `slices1` into a list of positions with `width` and static environment `tenv` yields `positions` `//#TE`;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` `//#TE`;
 - let `width'` be the length of the list `positions`;
 - annotating the bitfields `bitfields'` with `width'` in static environment `tenv` yields `bitfields''` `//#TE`;
 - `new_fields` is the nested bitfield with `slices1` and bitfields `bitfields''`, that is, `BitField_Nested(slices1, bitfields'')`.
- * All of the following apply (TYPE):
- Annotating the type `t` yields `t'` `//#TE`;
 - checking whether the range of positions in `slices1` fit inside `0..width` yields `TRUE` `//#TE`;
 - converting the list of slices `slices1` into a list of positions in `tenv` yields `positions` `//#TE`;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` `//#TE`;
 - let `width'` be the length of the list `positions`;
 - checking whether the `t` and the bitvector with `width'` bits have the same width yields `TRUE` `//#TE`
 - `new_field` is the typed-bitfield with name `name`, list of slices `slices1` and type `t'`, that is, `BitField_Type(name, slices1, t')`.

10.1.2 Example

In the following example, all the uses of bitvector types with bitfields are valid:

```
type MyType of bits(4) { [3:2] A, [1] B };

func foo (x: bits(4) { [3:2] A, [1] B }) => bits(4) { [3:2] A, [1] B }
begin
  return x;
end

func main () => integer
begin
  var x: bits(4) { [3:2] A, [1] B };

  x = '1010';
  x = foo (x as bits(4) { [3:2] A, [1] B });

  let y: bits(4) { [3:2] A, [1] B } = x;

  assert x as bits(4) { [3:2] A, [1] B } == x;

  return 0;
end
```


10.1.3 Formally

SIMPLE

$$\begin{array}{c}
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \frac{\text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE}{\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Simple}(\text{name}, \text{slices})) \xrightarrow{\text{type}} \text{BitField_Simple}(\text{name}, \text{slices1})}
 \end{array}$$

NESTED

$$\begin{array}{c}
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \begin{array}{c}
 \text{disjoint_slices_to_positions}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \quad // \quad \#TE \\
 \text{check_positions_in_width}(\text{tenv}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{width}' := |\text{positions}| \\
 \text{annotate_bitfields}(\text{tenv}, \text{width}', \text{bitfields}') \xrightarrow{\text{type}} \text{bitfields}' \quad // \quad \#TE
 \end{array} \\
 \hline
 \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Nested}(\text{name}, \text{slices}, \text{bitfields}')) \xrightarrow{\text{type}} \text{BitField_Nested}(\text{slices1}, \text{bitfields}')
 \end{array}$$

TYPE

$$\begin{array}{c}
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \begin{array}{c}
 \text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}' \quad // \quad \#TE \\
 \text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{disjoint_slices_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \quad // \quad \#TE \\
 \text{check_positions_in_width}(\text{tenv}, \text{slices1}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{width}' := |\text{positions}| \\
 \text{check_bits_equal_width}(\text{T_Bits}(\text{width}', []), \text{t}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE
 \end{array} \\
 \hline
 \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Type}(\text{name}, \text{slices}, \text{t})) \xrightarrow{\text{type}} \text{BitField_Type}(\text{name}, \text{slices1}, \text{t}')
 \end{array}$$

10.2 TypingRule.TBitFields

The function

$$\text{annotate_bitfields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e_width}}, \overbrace{\text{bitfields}}^{\text{fields}}) \longrightarrow \overbrace{\text{bitfields} \cup \text{TTypeError}}^{\text{new_fields}} \quad \#TE$$

annotates a list of bitfields — **fields** — with an expression denoting the overall number of bits in the containing bitvector type — **e_width**, in an environment **tenv**, resulting in an annotated list of bitfields — **new_fields** or a type error, if one is detected.

10.2.1 Prose

All of the following apply:

- checking that the list of bitfield names in `bitfields` does not contain duplicates yields `TRUE` *//* `#TE`;
- symbolically simplifying `e_width` in `tenv` via `reduce_constants` yields the literal integer for `width` *//* `#TE`;
- annotating each bitfield `field` in `fields` with width `width` in `tenv` yields the corresponding annotated bitfield `new_field` *//* `#TE`;
- `new_fields` is the list of annotated bitfields.

10.2.2 Formally

$$\begin{array}{c}
 \text{names} := [\text{field} \in \text{fields} : \text{bitfield_get_name}(\text{field})] \\
 \text{check_no_duplicates}(\text{names}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{reduce_constants}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} \text{L_Int}(\text{width}) \text{ // } \#TE \\
 \text{field} \in \text{fields} : \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} \text{new_field} \text{ // } \#TE \\
 \text{new_fields} := [\text{field} \in \text{fields} : \text{new_field}] \\
 \hline
 \text{annotate_bitfields}(\text{tenv}, \text{e_width}, \text{fields}) \xrightarrow{\text{type}} \text{new_fields}
 \end{array}$$

Chapter 11

Typing of Expressions

The function

$$\text{annotate_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow (\overbrace{\text{ty}}^{\text{t}} \times \overbrace{\text{expr}}^{\text{new_e}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

specifies how to annotate an expression e in an environment tenv . Formally, the result of annotating the expression e in tenv is either the pair $(t, \text{new_e})$, where t is a type and new_e is an annotated expression, or a type error, and one of the following applies:

- `TypingRule.ELit` (see Section 11.1);
- `TypingRule.ELocalVar` (see Section 11.2)
- `TypingRule.EGlobalVarConstant` (see Section 11.3)
- `TypingRule.EGlobalVarConstantNoVal` (see Section 11.4)
- `TypingRule.EGlobalVar` (see Section 11.5)
- `TypingRule.EUndefIdent` (see Section 11.6)
- `TypingRule.Binop` (see Section 11.7)
- `TypingRule.Unop` (see Section 11.8)
- `TypingRule.ECond` (see Section 11.9)
- `TypingRule.ESlice` (see Section 11.10)
- `TypingRule.ESetter` (see Section 11.11)
- `TypingRule.ECall` (see Section 11.12)
- `TypingRule.EGetArray` (see Section 11.13)
- `TypingRule.ESliceOrEGetArrayError` (see Section 11.14)

- `TypingRule.ERecord` (see Section 11.15)
- `TypingRule.EGetRecordField` (see Section 11.16)
- `TypingRule.EGetBadRecordField` (see Section 11.17)
- `TypingRule.EGetBadBitField` (see Section 11.18)
- `TypingRule.EGetBitField` (see Section 11.19)
- `TypingRule.EGetBitFieldNested` (see Section 11.20)
- `TypingRule.EGetBitFieldTyped` (see Section 11.21)
- `TypingRule.EGetTupleItem` (see Section 11.22)
- `TypingRule.EGetBadField` (see Section 11.23)
- `TypingRule.EConcat` (see Section 11.24)
- `TypingRule.ETuple` (see Section 11.25)
- `TypingRule.EUnknown` (see Section 11.26)
- `TypingRule.EPattern` (see Section 11.27)
- `TypingRule.ATC` (see Section 11.28)

The annotation rewrites the input expression in the following cases, making the annotation of statements simpler:

- Variables with constant values are substituted by their constant values.
- Slicing expressions that correspond to calling a getter are replaced with respective call expressions.
- Slicing expressions that correspond to `array access` expressions are replaced by `array access` expressions.

We also define the following helper rules:

- `TypingRule.Lit` (Section 11.29)
- `TypingRule.ExpressionList` (Section 11.30)
- `TypingRule.ReduceSlicesToCall` (Section 11.31)
- `TypingRule.StaticConstrainedInteger` (Section 11.32)
- `TypingRule.CheckATC` (Section 11.33)

11.1 TypingRule.ELit

11.1.1 Prose

All of the following apply:

- e is the literal expression v ;
- t is the type of the literal v ;
- new_e is e .

11.1.2 Formally

$$\frac{\text{annotate_literal}(v) \xrightarrow{\text{type}} t}{\text{annotate_expr}(\text{tenv}, E.\text{Literal}(v)) \xrightarrow{\text{type}} (t, E.\text{Literal}(v))}$$

11.2 TypingRule.ELocalVar

11.2.1 Prose

One of the following applies:

- All of the following apply (CONSTANT):
 - * e denotes a local variable x ;
 - * x is bound to a local constant v of type t in the local environment given by tenv ;
 - * new_e is the `Literal` v .
- All of the following apply (NON_CONSTANT):
 - * e denotes a local variable x ;
 - * x is not bound to a constant in the local environment given by tenv ;
 - * x has type t in the local environment given by tenv ;
 - * new_e is e .

11.2.2 Formally

$$\frac{\begin{array}{l} \text{CONSTANT} \\ L^{\text{tenv}}.\text{constant_values}(x) = v \quad L^{\text{tenv}}.\text{local_storage_types}(x) = (t, \text{LDK_Constant}) \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{E.\text{Var}(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{E.\text{Literal}(v)}^{\text{new_e}})}$$

$$\frac{\begin{array}{l} \text{NON_CONSTANT} \\ L^{\text{tenv}}.\text{constant_values}(x) = \perp \\ L^{\text{tenv}}.\text{local_storage_types}(x) = (t, k) \quad k \in \{\text{LDK_Var}, \text{LDK_Let}\} \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{E.\text{Var}(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{E.\text{Var}(x)}^{\text{new_e}})}$$

11.3 TypingRule.EGlobalVarConstantVal

11.3.1 Prose

All of the following apply:

- e denotes a global variable x ;
- x is bound to a constant v of type ty in the global environment given by $tenv$;
- t is ty ;
- new_e is the Literal for v .

11.3.2 Formally

$$\frac{G^{tenv}.global_storage_types(x) = (ty, GDK_Constant) \quad G^{tenv}.constant_values(x) = v}{\text{annotate_expr}(tenv, E_Var(x)) \xrightarrow{\text{type}} (ty, E_Literal(v))}$$

11.4 TypingRule.EGlobalVarConstantNoVal

Our type system does not currently address assignments of non-constant expressions (for example, function calls) to global constant variables. This section can be seen as a placeholder until the right details are filled in.

11.4.1 Prose

All of the following apply:

- e denotes a global variable x ;
- x is not bound to constant in the global environment given by $tenv$;
- t is ty ;
- new_e is e .

11.4.2 Formally

$$\frac{G^{tenv}.global_storage_types(x) = (ty, GDK_Constant) \quad G^{tenv}.constant_values(x) = \perp}{\text{annotate_expr}(tenv, E_Var(x)) \xrightarrow{\text{type}} (ty, E_Var(x))}$$

11.5 TypingRule.EGlobalVar

11.5.1 Prose

All of the following apply:

- e denotes a global variable x ;
- x is not bound to a global constant;
- x has type ty in the global environment given by $tenv$;
- t is ty ;
- new_e is e .

11.5.2 Formally

$$\frac{G^{tenv}.constant_values(x) = \perp \quad G^{tenv}.global_storage_types(x) = (ty, k) \quad k \neq GDK_Constant}{annotate_expr(tenv, E_Var(x)) \xrightarrow{type} (ty, E_Var(x))}$$

11.6 TypingRule.EUndefIdent

11.6.1 Prose

All of the following apply:

- e is a variable x ;
- x is not bound to a type in $tenv$;
- the result is a type error indicating that x is an undefined identifier.

11.6.2 Formally

$$\frac{G^{tenv}.global_storage_types(x) = \perp \quad L^{tenv}.global_storage_types(x) = \perp}{annotate_expr(tenv, \overbrace{E_Var(x)}^e) \xrightarrow{type} TypeError(TE_UI)}$$

11.7 TypingRule.Binop

11.7.1 Prose

All of the following apply:

- e denotes a binary operation op over two expressions $e1$ and $e2$, that is, $E.Binop(op, e1, e2)$;

- the result of annotating $e1$ in $tenv$ is $(t1, e1') \#TE$;
- the result of annotating $e2$ in $tenv$ is $(t2, e2') \#TE$;
- the result of checking compatibility of op with $t1$ and $t2$ as per Section 8.9 is $t \#TE$;
- new_env denotes op over $e1'$ and $e2'$.

11.7.2 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t1, e1') \#TE \\
 \text{annotate_expr}(tenv, e2) \xrightarrow{\text{type}} (t2, e2') \#TE \\
 \text{check_binop}(tenv, op, t1, t2) \xrightarrow{\text{type}} t \#TE
 \end{array}
 }{
 \text{annotate_expr}(tenv, \overbrace{E_Binop(op, e1, e2)}^e) \xrightarrow{\text{type}} (t, \overbrace{E_Binop(op, e1', e2')}^{new_e})
 }$$

11.8 TypingRule.Unop

11.8.1 Prose

All of the following apply:

- e denotes a unary operation op over an expression e' , that is $E_Unop(op, e')$;
- annotating e' in $tenv$ yields $(t'', e'') \#TE$;
- checking compatibility of op with t'' as per Section 8.8 yields $t \#TE$;
- new_e denotes op over e'' , that is, $E_Unop(op, e'')$.

11.8.2 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(tenv, e') \xrightarrow{\text{type}} (t'', e'') \#TE \\
 \text{check_unop}(tenv, op, t'') \xrightarrow{\text{type}} t \#TE
 \end{array}
 }{
 \text{annotate_expr}(tenv, E_Unop(op, e')) \xrightarrow{\text{type}} (t, E_Unop(op, e''))
 }$$

11.9 TypingRule.ECond

11.9.1 Prose

All of the following apply:

- e denotes a conditional expression with condition e_cond with two options e_true and e_false ;
- annotating e_cond in $tenv$ results in $(t_cond, e_cond') \#TE$;

- annotating `e_true` in `tenv` results in (t_true, e_true') $\text{\textit{//}} \#TE$;
- annotating `e_false` in `tenv` results in (t_false, e_false') ;
- obtaining the lowest common ancestor of `t_true` and `t_false` results in t $\text{\textit{//}} \#TE$;
- `new_e` is the condition `e_cond'` with two options `e_true'` and `e_false'`, that is, `E_Cond(e_cond', e_true', e_false')`.

11.9.2 Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e_cond) \xrightarrow{\text{type}} (t_cond, e_cond') \text{ \textit{//} \#TE} \\
 \text{annotate_expr}(\text{tenv}, e_true) \xrightarrow{\text{type}} (t_true, e_true') \text{ \textit{//} \#TE} \\
 \text{annotate_expr}(\text{tenv}, e_false) \xrightarrow{\text{type}} (t_false, e_false') \text{ \textit{//} \#TE} \\
 \hline
 \text{lowest_common_ancestor}(t_true, t_false) \xrightarrow{\text{type}} t \text{ \textit{//} \#TE} \\
 \hline
 \text{annotate_expr}(\text{E_Cond}(e_cond, e_true, e_false)) \xrightarrow{\text{type}} \\
 (t, \text{E_Cond}(e_cond', e_true', e_false'))
 \end{array}$$

11.10 TypingRule.ESlice

11.10.1 Prose

All of the following apply:

- `e` denotes the slicing of expression `e'` by the slices `slices`, that is, `E_Slice(e', slices)`;
- determining whether `e'` together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` $\text{\textit{//}} \#TE$;
- annotating the expression `e'` in `tenv` yields (t_e', e'') $\text{\textit{//}} \#TE$;
- obtaining the `structure` of `t_e'` in `tenv` yields `struct_t_e'` $\text{\textit{//}} \#TE$;
- `struct_t_e'` is either a bitvector or an integer;
- obtaining the width of `slices` in `tenv` via `slices_width` yields `w` $\text{\textit{//}} \#TE$;
- `slices'` is the result of annotating `slices` in `tenv`;
- `t` is the bitvector type of width `w`, that is, `T_Bits(w, [])`;
- `new_e` is the slicing of expression `e''` by the slices `slices'`, that is, `E_Slice(e'', slices')`.

11.10.2 Formally

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\
\text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_e', e'') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} \text{struct_t_e'} \quad // \quad \#TE \\
\text{ast_label}(\text{struct_t_e'}) \in \{T_Int, T_Bits\} \quad \text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} w \quad // \quad \#TE \\
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices'} \quad // \quad \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{T_Bits(w, [])}^t, \overbrace{E_Slice(e'', \text{slices}')}^{\text{new_e}})
\end{array}$$

11.10.3 Comments

The width of `slices` might be a symbolic expression if one of the widths references a `let` identifier with a non-compile-time-constant initializer expression.

11.11 TypingRule.ESetter

11.11.1 Prose

All of the following apply:

- `e` denotes the slicing of expression `e'` by the slices `slices`, that is, `E.Slice(e', slices)`;
- determining whether `e'` together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a positive answer — $\langle\langle \text{name}, \text{args} \rangle\rangle \quad // \quad \#TE$;
- applying `annotate_call` to annotate the call with $(\text{tenv}, \text{name}, \text{args}, \text{ST_Setter})$ yields $(\text{name1}, \text{args1}, \text{eqs}, \langle \text{ty} \rangle) \quad // \quad \#TE$;
- `t` is `ty`;
- `new_e` is the call expression `E.Call(name1, args1, eqs)`.

11.11.2 Formally

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \langle\langle \text{name}, \text{args} \rangle\rangle \quad // \quad \#TE \\
\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Setter}) \xrightarrow{\text{type}} (\text{name1}, \text{args1}, \text{eqs}, \langle \text{ty} \rangle) \quad // \quad \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{\langle \text{ty} \rangle}^t, \overbrace{E_Call(\text{name1}, \text{args1}, \text{eqs})}^{\text{new_e}})
\end{array}$$

11.12 TypingRule.ECall

11.12.1 Prose

All of the following apply:

- e denotes a call to a subprogram named `name` with arguments `args`, that is, `E.Call(name, args)`;
- applying `annotate_call` to `name`, `args`, and `ST_Function` in `tenv` annotates the call of that subprogram in `tenv` as a function (annotating calls is defined in Chapter 19) and yields $(name', args', eqs', \langle t \rangle) \#TE$. Notice that passing `ST_Function` to `annotate_call` checks that `name` is not a procedure and that a value is indeed returned;
- `new_e` is the call to the subprogram named `name'` with arguments `args'` and parameters `eqs'`, that is, `E.Call(name', args', eqs')`.

11.12.2 Formally

$$\frac{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Function}) \xrightarrow{\text{type}} (\text{name}', \text{args}', \text{eqs}', \langle t \rangle) \#TE}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E.Call}(\text{name}, \text{args})}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E.Call}(\text{name}', \text{args}', \text{eqs}')}^{\text{new_e}})}$$

11.13 TypingRule.EGetArray

Definition 29 (Array Access) We refer to a right-hand-side expression of the form $b[i]$, where b, i are subexpressions, as an *array access* expression. We refer to b and i as the *base* and the *index* subexpressions, respectively.

In the untyped AST, an *array access* expression is represented by `E.Slice(base, [index])`. In the typed AST, it is represented by `E.GetArray(base, index)`

11.13.1 Prose

All of the following apply:

- e denotes the slicing of expression e' by the slices `slices`;
- determining whether e' together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` $\#TE$;
- (t_e', e'') is the result of annotating the expression e' in `tenv`;
- t_e' has the structure of an array with index `size` and element type `ty'`;
- One of the following applies:
 - * All of the following apply (OKAY):

- `slices` is a list containing a single slice for an expression `e_index`, that is, `[Slice_Single(e_index)]`;
 - annotating the expression `e_index` in `tenv` yields (t_index', e_index') *// #TE*;
 - determining the type of the array index for `size` in `tenv` via *type_of_array_length* yields `wanted_t_index`;
 - determining whether `t_index'` *type-satisfies* `wanted_t_index` yields *TRUE* *// #TE*;
 - `t` is `ty'`;
 - `new_e` is the *array access* expression for `e''` and index `e_index'`, that is, `E_GetArray(e'', e_index')`.
- * All of the following apply (ERROR):
- `slices` is not a list containing a single slice for an expression `e_index`;
 - the result is a type error indicating that an array must be accessed with a slice corresponding to a single index expression.

11.13.2 Formally

OKAY

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e'') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} T_Array(\text{size}, ty') \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{slices} = [\text{Slice_Single}(e_index)] \\
\text{annotate_expr}(\text{tenv}, e_index) \xrightarrow{\text{type}} (t_index', e_index') \quad // \quad \#TE \\
\text{type_of_array_length}(\text{tenv}, \text{size}) \xrightarrow{\text{type}} \text{wanted_t_index} \\
\text{checked_typesat}(\text{tenv}, t_index', \text{wanted_t_index}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{ty'}^t, \overbrace{E_GetArray(e'', e_index')}^{\text{new_e}})
\end{array}$$

ERROR

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e'') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} T_Array(\text{size}, ty') \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{slices} \neq [\text{Slice_Single}(_)] \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{IllegalArraySlice})
\end{array}$$

11.14 TypingRule.ESliceOrEGetArrayError

11.14.1 Prose

All of the following apply:

- `e` denotes the slicing of expression `e'` by the slices `slices`;
- determining whether `e'` together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` *//* `#TE`;
- `(t_e', e'')` is the result of annotating the expression `e'` in `tenv`;
- `t_e'` has the structure `t'`;
- `t'` is neither an integer type, a bitvector type, or an array type;
- the result is an error indicating that the type of `e'` is inappropriate for slicing.

11.14.2 Formally

$$\frac{
 \begin{array}{l}
 \text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \text{ // } \#TE \\
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_e', e'') \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} t' \quad \text{ast_label}(t') \notin \{T_Int, T_Bits, T_Array\}
 \end{array}
 }{
 \text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{IllegalSliceType})
 }$$

11.15 TypingRule.ERecord

11.15.1 Prose

All of the following apply:

- `e` denotes the record construction expression (which is also used for creating exceptions) of type `ty` with fields `fields`, that is, `E_Record(ty, fields)`;
- obtaining the *underlying type* of `ty` in `tenv` yields `ty_anon` *//* `#TE`;
- checking that `ty_anon` is a *structured type* yields `TRUE` *//* `#TE`;
- `ty_anon` is a *structured type* with a list of `field` elements (consisting of a field name and a field type);
- obtaining the list of field names from `fields` yields the list of identifiers `initialized_fields`;
- obtaining the list of field names from `field_types` yields the list of identifiers `names`;
- checking whether the set of identifiers in `names` is equal to the set of identifiers in `initialized_fields` yields `TRUE` *//* `#TE`;
- checking that the list `initialized_fields` does not contain duplicates yields `TRUE` *//* `#TE`;

- applying *annotate_field_init* to annotate each field element (name, e') of fields in tenv yields $(\text{name}, e_{\text{name}}) \text{ // } \#TE$;
- define fields' as the list containing $(\text{name}, e_{\text{name}})$ for each field element (name, e') of fields;
- t is ty ;
- new_e is the record expression with type ty and field initializers fields' , that is, $E_Record(\text{ty}, \text{fields}')$;

11.15.2 Formally

$$\begin{array}{c}
\text{check}(\text{ast_label}(\text{ty}) = T_Named, \text{NamedTypeExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty_anon} \text{ // } \#TE \\
\text{check}(\text{ast_label}(\text{ty_anon}) \in \{T_Record, T_Exception\}, TE_EST) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{ty_anon} \stackrel{\text{is}}{=} L(\text{field_types}) \quad \text{initialized_fields} := \{\text{name} \mid (\text{name}, _) \in \text{fields}\} \\
\quad \text{names} := \text{field_names}(\text{field_types}) \\
\text{check}(\{\text{names}\} = \{\text{initialized_fields}\}, TE_MFI) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_no_duplicates}(\text{initialized_fields}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
(\text{name}, e') \in \text{fields} : \text{annotate_field_init}(\text{tenv}, (\text{name}, e'), \text{field_types}) \xrightarrow{\text{type}} \\
\quad (\text{name}, e_{\text{name}}) \text{ // } \#TE \\
\text{fields}' := [(\text{name}, e') \in \text{fields} : (\text{name}, e_{\text{name}})] \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Record(\text{ty}, \text{fields})}^e) \xrightarrow{\text{type}} (\overbrace{\text{ty}}^t, \overbrace{E_Record(\text{ty}, \text{fields}')}^{\text{new_e}})
\end{array}$$

11.16 TypingRule.EGetRecordField

11.16.1 Prose

All of the following apply:

- e denotes the access of field field_name in the value represented by the expression $e1$, that is, $E_GetField(e1, \text{field_name})$;
- annotating the expression $e1$ in tenv yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the *underlying type* of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a *structured type* with fields fields ;
- the field field_name is associated with the type t in fields ;
- new_e is the access of field field_name on the record or exception object $e2$, that is, $E_GetField(e2, \text{field_name})$.

11.16.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
t_e2 \stackrel{\text{is}}{=} L(\text{fields}) \\
\hline
L \in \{T_Record, T_Exception\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \langle t \rangle \\
\text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} (t, E_GetField(e2, \text{field_name}))
\end{array}$$

11.17 TypingRule.EGetBadRecordField

11.17.1 Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField(e1, field_name)`;
- annotating the expression $e1$ in tenv yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the [underlying type](#) of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a [structured type](#) with fields `fields`;
- the field `field_name` is not associated with any type in `fields`
- the result is a type error indicating the missing field.

11.17.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
t_e2 \stackrel{\text{is}}{=} L(\text{fields}) \\
\hline
L \in \{T_Record, T_Exception\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
\text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})
\end{array}$$

11.18 TypingRule.EGetBadBitField

11.18.1 Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField(e1, field_name)`;
- annotating the expression $e1$ in tenv yields $(t_e1, e2) \text{ // } \#TE$;

- obtaining the **underlying type** of t_e1 yields t_e2 *//* **#TE**;
- t_e2 is a bitvector type with bit fields **bitfields**;
- the field **field_name** is not found in **bitfields**
- the result is a type error indicating the missing field.

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \quad \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})
\end{array}$$

11.19 TypingRule.EGetBitField

11.19.1 Prose

All of the following apply:

- e denotes the access of field **field_name** in the value represented by the expression $e1$, that is, $E_GetField(e1, \text{field_name})$;
- annotating the expression $e1$ in tenv yields $(t_e1, e2)$ *//* **#TE**;
- obtaining the **underlying type** of t_e1 yields t_e2 *//* **#TE**;
- t_e2 is a bitvector type with bit fields **bitfields**;
- **field_name** is declared in **bitfields** with a slice list **slices**, that is, $\text{BitField_Simple}(_, \text{slices})$;
- $e3$ denotes the slicing of the expression $e2$ by the slices **slices**, that is, $E_Slice(e2, \text{slices})$;
- annotating $e3$ in tenv yields $(t, \text{new_e})$ *//* **#TE**.

11.19.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t, \text{new_e}) \text{ // } \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
\end{array}$$

11.20 TypingRule.EGetBitFieldNested

11.20.1 Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField($e1$, field_name)`;
- annotating the expression $e1$ in $tenv$ yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the [underlying type](#) of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a bitvector type with bit fields `bitfields`;
- `field_name` is declared in `bitfields` with a slice list `slices` and nested bitfields `bitfields'`, that is, `BitField_Nested(_, slices, bitfields')`;
- $e3$ denotes the slicing of the expression $e2$ by the slices `slices`, that is, `E_Slice($e2$, slices)`;
- annotating $e3$ in $tenv$ yields $(t_e4, new_e) \text{ // } \#TE$;
- t_e4 is a bitvector type with length expression `width`, that is, `T_Bits(width, _)`;
- t is a bitvector type with length expression `width` and bitfields `bitfields'`.

11.20.2 Formally

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(tenv, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \quad \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
 e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(tenv, e3) \xrightarrow{\text{type}} (t_e4, new_e) \text{ // } \#TE \\
 t_e4 \stackrel{\text{is}}{=} T_Bits(\text{width}, _) \quad t := T_Bits(\text{width}, \text{bitfields}') \\
 \hline
 \text{annotate_expr}(tenv, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, new_e)
 \end{array}$$

11.21 TypingRule.EGetBitFieldTypeed

11.21.1 Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField($e1$, field_name)`;
- annotating the expression $e1$ in $tenv$ yields $(t_e1, e2) \text{ // } \#TE$;

- obtaining the **underlying type** of t_e1 yields t_e2 $\text{\textit{\#TE}}$;
- t_e2 is a bitvector type with bit fields **bitfields**;
- **field_name** is declared in **bitfields** with a slice list **slices** and typed-bitfield with type t that is, $\text{BitField_Type}(_, \text{slices}, t)$;
- $e3$ denotes the slicing of the expression $e2$ by the slices **slices**, that is, $\text{E_Slice}(e2, \text{slices})$;
- annotating $e3$ in tenv yields $(t_e4, \text{new_e})$ $\text{\textit{\#TE}}$;
- determining whether t_e4 **type-satisfies** t yields TRUE $\text{\textit{\#TE}}$.

11.21.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \quad \text{\textit{\#TE}} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \quad \text{\textit{\#TE}} \\
t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} (\text{BitField_Type}(_, \text{slices}, t)) \\
e3 := \text{E_Slice}(e2, \text{slices}) \quad \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t_e4, \text{new_e}) \quad \text{\textit{\#TE}} \\
\text{checked_typesat}(\text{tenv}, t_e4, t) \xrightarrow{\text{type}} \text{TRUE} \quad \text{\textit{\#TE}} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_GetField}(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
\end{array}$$

11.22 TypingRule.EGetTupleItem

11.22.1 Prose

All of the following apply:

- e denotes the access of field **field_name** in the value represented by the expression $e1$, that is, $\text{E_GetField}(e1, \text{field_name})$;
- annotating the expression $e1$ in tenv yields $(t_e1, e2)$ $\text{\textit{\#TE}}$;
- obtaining the **underlying type** of t_e1 yields t_e2 $\text{\textit{\#TE}}$;
- t_e2 is tuple type with list of types **tys**, that is, $T_Tuple(\text{tys})$;
- **field_name** is an identifier with the prefix **item** and the constant **index**;
- determining whether **index** is between 0 and the number of types in **tys**, inclusive, yields TRUE $\text{\textit{\#TE}}$;
- t is the type at position **index** of **tys**;
- **new_e** is the expression for obtaining the item at index **index** from the expression $e2$, that is, $\text{E_GetItem}(e2, \text{index})$.

11.22.2 Formally

$$\frac{
\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
t_e2 \stackrel{\text{is}}{=} T_Tuple(\text{tys}) \quad \text{field_name} \stackrel{\text{is}}{=} \text{"item<index>"} \\
\text{check}(0 \leq \text{index} \leq |\text{tys}|, \text{IndexOutOfRange}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
t := \text{tys}[\text{index}] \quad \text{new_e} := E_GetItem(e2, \text{index})
\end{array}
}{
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
}$$

11.23 TypingRule.EGetBadField

11.23.1 Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E_GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields $(t_e1, e2) \text{ // \#TE}$;
- obtaining the **underlying type** of `t_e1` yields $t_e2 \text{ // \#TE}$;
- `t_e2` is neither one of the following types: `record`, `exception`, `bitvector`, or `tuple`;
- the result is an error indicating that the type of `e1` is inappropriate for accessing the field `field_name`.

11.23.2 Formally

$$\frac{
\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
\text{ast_label}(t_e2) \notin \{T_Record, T_Exception, T_Bits, T_Tuple\}
\end{array}
}{
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{ConflictingTypes})
}$$

11.24 TypingRule.EConcat

11.24.1 Prose

All of the following apply:

- e denotes the concatenation of a non-empty list of expressions `li`, that is, `E_Concat(li)`;
- annotating each expression `le[i]` in `tenv`, for $i = 1..k$, yields $(t_i, e_i) \text{ // \#TE}$;

- `e_s` is the list of expressions `ei`, for $i = 1..k$;
- obtaining the bitvector width of `ti` in `tenv` (which also checks that `ti` is a bitvector type), for $i = 1..k$, yields `wi` *#TE*;
- to obtain the (symbolic) width of the resulting bitvector, first define `width_sum1` to be `w1`;
- then define `width_sumi`, for $i = 2..k$, to be obtained by reducing the expression that sums `width_sumi-1` with the width `wi`;
- `t` is the bitvector of length `width_sumk` and the empty bitfield list, that is, `T_Bits(width_sumk, [])`;
- `new_e` is the concatenation expression for `e_s`, that is, `E_Concat(e_s)`.

11.24.2 Formally

$$\begin{array}{c}
 i = 1..k : \text{annotate_expr}(\text{tenv}, \text{li}[i]) \xrightarrow{\text{type}} (t_i, e_i) \text{ // } \#TE \\
 t_s := [i = 1..k : t_i] \\
 e_s := [i = 1..k : e_i] \quad i = 1..k : \text{get_bitvector_width}(\text{tenv}, t_i) \xrightarrow{\text{type}} w_i \text{ // } \#TE \\
 \text{width_sum}_1 := w_1 \\
 i = 2..k : \text{normalize}(\text{tenv}, \text{E_Binop}(\text{PLUS}, \text{width_sum}_{i-1}, w_i)) \xrightarrow{\text{type}} \text{width_sum}_i \\
 \hline
 \text{annotate_expr}(\text{tenv}, \text{E_Concat}(\text{li})) \xrightarrow{\text{type}} (\text{T_Bits}(\text{width_sum}_k, []), \text{E_Concat}(e_s))
 \end{array}$$

11.24.3 Comments

The sum of the widths of the bitvector types `ts` might be a symbolic expression that is unresolvable to an integer. For example:

```

func foo{N}(x: bits(N)) => bit
begin
  return x[0];
end

config LIMIT1: integer = 2;
config LIMIT2: integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = 7;

func bar() => integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
begin
  var ret: integer = 0;
  while ret < LIMIT1 do
    ret = ret + ret * 2;
  end
  return ret as integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
end

```

```

func main() => integer
begin
  let N = bar();
  let M = LIMIT2;
  let x = Zeros(N);
  let y = Zeros(M);
  let z = foo([x, y]);
  return 0;
end

```

11.25 TypingRule.ETuple

11.25.1 Prose

All of the following apply:

- e denotes a tuple expression with list of expressions li , that is, $E_Tuple(li)$;
- annotating each expression $le[i]$ in $tenv$, for $i = 1..k$, yields $(t_i, e_i) \#TE$;
- t is the tuple type with list of types t_i , for $i = 1..k$;
- new_e is tuple expression over list of expressions e_i , for $i = 1..k$.

11.25.2 Formally

$$\frac{i = 1..k : \text{annotate_expr}(tenv, le[i]) \xrightarrow{\text{type}} (t_i, e_i) \#TE}{\text{annotate_expr}(tenv, E_Tuple(li)) \xrightarrow{\text{type}} (T_Tuple(t_{1..k}), E_Tuple(e_{1..k}))}$$

11.26 TypingRule.EUnknown

11.26.1 Prose

All of the following apply:

- e denotes an expression UNKNOWN of type ty , that is, $E_Unknown(ty)$;
- annotating the type ty in $tenv$ yields $ty1 \#TE$;
- obtaining the [structure](#) of $ty1$ in $tenv$ yields $ty2 \#TE$;
- t is $ty1$;
- new_e is an expression UNKNOWN of type $ty2$, that is, $E_Unknown(ty2)$.

11.26.2 Formally

$$\frac{\begin{array}{c} \text{annotate_type}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \quad \# \text{TE} \\ \text{get_structure}(\text{tenv}, \text{ty1}) \xrightarrow{\text{type}} \text{ty2} \quad \# \text{TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \text{E_Unknown}(\text{ty})) \xrightarrow{\text{type}} (\text{ty1}, \text{E_Unknown}(\text{ty2}))}$$

11.27 TypingRule.EPattern

11.27.1 Prose

All of the following apply:

- e denotes a pattern expression to test whether $e1$ matches the pattern pat , that is, $\text{E_Pattern}(e1, \text{pat})$;
- annotating the expression $e1$ in tenv yields $(t_e2, e2) \# \text{TE}$;
- applying *annotate_pattern* to t_e2 and pat in tenv yields $\text{pat}' \# \text{TE}$;
- t is T_Bool ;
- new_e denotes whether the expression $e2$ matches pat' , that is, $\text{E_Pattern}(e2, \text{pat}')$.

11.27.2 Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e2, e2) \quad \# \text{TE} \\ \text{annotate_pattern}(\text{tenv}, t_e2, \text{pat}) \xrightarrow{\text{type}} \text{pat}' \quad \# \text{TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Pattern}(e1, \text{pat})}^e) \xrightarrow{\text{type}} (\overbrace{\text{T_Bool}}^t, \overbrace{\text{E_Pattern}(e2, \text{pat}')}^{\text{new_e}})}$$

11.28 TypingRule.ATC

11.28.1 Prose

All of the following apply:

- e denotes an asserting type conversion with expression e' and type ty , that is $\text{E_ATC}(e', \text{ty})$;
- annotating the expression e' in tenv yields $(t, e'') \# \text{TE}$;
- obtaining the *structure* of t in tenv yields $t_struct \# \text{TE}$;
- annotating the type ty in tenv yields $\text{ty}' \# \text{TE}$;
- obtaining the *structure* of ty' in tenv yields $\text{ty_struct} \# \text{TE}$;

- applying *check_atc* to *t_struct* and *ty_struct* in *tenv* to check whether the type assertion will always fail yields *TRUE*//*#TE*;
- checking whether *t* *subtype-satisfies* *ty'* in *tenv* yields *b*//*#TE*;
- *new_e* is *E_ATC*(*ty'*, *e''*) if *b* is *TRUE* and *e''* otherwise;
- *t* is *ty'*.

11.28.2 Formally

$$\begin{array}{c}
 \text{TYPE_EQUAL} \\
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t, e'') \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ // } \#TE \\
 \text{annotate_type}(\text{tenv}, ty) \xrightarrow{\text{type}} ty' \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, ty') \xrightarrow{\text{type}} ty_struct \text{ // } \#TE \\
 \text{check_atc}(\text{tenv}, t_struct, ty_struct) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{subtype_satisfies}(\text{tenv}, t, ty') \xrightarrow{\text{type}} b \text{ // } \#TE \\
 \text{new_e} := \text{choice}(b, E_ATC(ty', e''), e'') \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_ATC(e', ty)}^e) \xrightarrow{\text{type}} (\overbrace{ty'}^t, \text{new_e})
 \end{array}$$

11.29 TypingRule.Lit

Annotating literals is done via the helper function

$$\text{annotate_literal}(\overbrace{\text{literal}}^l) \longrightarrow \overbrace{ty}^t$$

which we use in this chapter for TypingRule.ELit as well as in subsequent chapters.

11.29.1 Prose

The result of annotating a literal *l* is *t* and one of the following applies:

- (*int*): *l* is an integer literal *n* and *t* is the well-constrained integer type, constraining its set to the single value *n*;
- (*bool*): *l* is a Boolean literal and *t* is the Boolean type;
- (*real*): *l* is a real literal and *t* is the real type;
- (*string*): *l* is a string literal and *t* is the string type;
- (*bitvector*): *l* is a bitvector literal of length *n* and *t* is the bitvector type of fixed width *n*.

11.29.2 Example

In the following example, we show several literals and their corresponding types in comments:

```
func main () => integer
begin
  var n1 = 5; // type: integer{5}
  var n2 = 1_000__000; // type integer{1000000}
  var n4 = 0xa_b_c_d_e_f__A__B__C__D__E__F__0___1234567890;
    // type integer{53170898287292728730499578000}
  var btrue = TRUE; // type: boolean
  var bfalse = FALSE; // type: boolean
  var rzero = 1234567890.0123456789; // type: real
  var s1 = "hello\\world \n\t \"here I am \"; // type: string
  var s2 = ""; // type: string
  var bv1 = '11 01'; // type: bits(4)
  var bv2 = ''; // type: bits(0)
  return 0;
end
```

11.29.3 Formally

INT

$$\text{annotate_literal}(\text{L_Int}(n)) \xrightarrow{\text{type}} \text{T_Int}(\langle [\text{Constraint_Exact}(\overset{\text{E.Literal(L.Int)}{\boxed{n}})}] \rangle)$$

BOOL

$$\text{annotate_literal}(\text{L_Bool}(_)) \xrightarrow{\text{type}} \text{T_Bool}$$

REAL

$$\text{annotate_literal}(\text{L_Real}(_)) \xrightarrow{\text{type}} \text{T_Real}$$

STRING

$$\text{annotate_literal}(\text{L_String}(_)) \xrightarrow{\text{type}} \text{T_String}$$

BITVECTOR

$$\frac{n := |\text{bits}|}{\text{annotate_literal}(\text{L_Bitvector}(\text{bits})) \xrightarrow{\text{type}} \text{T_Bits}(\overset{\text{E.Literal(L.Int)}{\boxed{n}}}{}, [])}$$

11.30 TypingRule.ExpressionList

The function

$$\text{annotate_exprs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}^*}^{\text{exprs}}) \longrightarrow \overbrace{(\text{ty} \times \text{expr})^*}^{\text{typed_exprs}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of expressions **exprs** from left to right, yielding a list of pairs **typed_exprs**, each consisting of a type and expression. Otherwise, the result is a type error.

11.30.1 Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **exprs** is empty;
 - * **typed_exprs** is empty.
- All of the following apply (**NON_EMPTY**):
 - * **exprs** has **e** as its **head** expression and **exprs1** as its **tail**;
 - * annotating **e** in **tenv** yields the pair **typed_expr** consisting of a type and an expression **// #TE**;
 - * annotating the expression list **exprs1** in **tenv** yields **typed_exprs // #TE**;
 - * **typed_exprs** is the list with **typed_expr** as its **head** and **typed_exprs** as its **tail**.

11.30.2 Formally

$$\begin{array}{c} \text{EMPTY} \\ \text{annotate_exprs}(\text{tenv}, \overbrace{[]}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{typed_exprs}} \\ \\ \text{NON_EMPTY} \\ \frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{typed_expr} \text{ // } \#TE \\ \text{annotate_exprs}(\text{tenv}, \text{exprs1}) \xrightarrow{\text{type}} \text{typed_exprs1} \text{ // } \#TE \end{array}}{\text{annotate_exprs}(\text{tenv}, \overbrace{[e] + \text{exprs1}}^{\text{exprs}}) \xrightarrow{\text{type}} \overbrace{[\text{typed_expr}] + \text{typed_exprs1}}^{\text{typed_exprs}}} \end{array}$$

11.31 TypingRule.ReduceSlicesToCall

The function

$$\text{reduce_slices_to_call}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \langle \overbrace{(\text{identifier} \times \text{expr}^*)}^{\text{name} \times \text{args}} \rangle \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the expression e together with the list of slices $slices$ constitute a call to a subprogram in $tenv$. If so, it returns a pair consisting of the name of the called subprogram — $name$ — and the list of actual arguments — $args$. Otherwise, it returns $None$. Otherwise, the result is a type error.

11.31.1 Prose

One of the following applies:

- All of the following apply (YES):
 - * e is a variable expression for x , that is, $E_Var(x)$;
 - * determining whether x is a subprogram name with $slices$ as its actual arguments via `should_slices_reduce_to_call` yields a list of actual argument expressions `args`^{*#TE*};
 - * the result is $\langle(x, args)\rangle$.
- All of the following apply (NO):
 - * e is a variable expression for x , that is, $E_Var(x)$;
 - * determining whether x is a subprogram name with $slices$ as its actual arguments via `should_slices_reduce_to_call` yields $None$;
 - * the result is $None$.
- All of the following apply (NON_VAR):
 - * e is not a variable expression;
 - * the result is $None$.

11.31.2 Formally

YES

$$\frac{\text{should_slices_reduce_to_call}(tenv, x, slices) \xrightarrow{\text{type}} \langle args \rangle}{\text{reduce_slices_to_call}(tenv, E_Var(x), slices) \xrightarrow{\text{type}} \langle(x, args)\rangle}$$

NO

$$\frac{\text{should_slices_reduce_to_call}(tenv, x, slices) \xrightarrow{\text{type}} None}{\text{reduce_slices_to_call}(tenv, E_Var(x), slices) \xrightarrow{\text{type}} None}$$

NON_VAR

$$\frac{\text{ast_label}(e) \neq E_Var}{\text{reduce_slices_to_call}(tenv, e, slices) \xrightarrow{\text{type}} None}$$

11.32 TypingRule.StaticConstrainedInteger

The function

$$\text{annotate_static_constrained_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{e''} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a [statically evaluable](#) integer expression e of a constrained integer type in the static environment tenv and returns the annotated expression e'' . Otherwise, the result is a type error.

11.32.1 Prose

All of the following apply:

- annotating the expression e in tenv yields $(t, e') \#TE$;
- determining whether t is a statically [constrained integer](#) in tenv yields $\text{TRUE} \#TE$;
- determining whether e' is [statically evaluable](#) in tenv yields $\text{TRUE} \#TE$;
- applying [normalize](#) to e' in tenv yields e'' .

11.32.2 Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \#TE \\ \text{check_constrained_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE} \#TE \\ \text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \#TE \\ \text{normalize}(\text{tenv}, e') \xrightarrow{\text{type}} e'' \end{array}}{\text{annotate_static_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e''}$$

11.33 TypingRule.CheckATC

The function

$$\text{check_atc}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{t1}, \overbrace{\text{ty}}^{t2}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the types $t1$ and $t2$, which are assumed to not be named types, are compatible for a typing assertion in the static environment tenv , yielding TRUE . Otherwise, the result is a type error.

11.33.1 Prose

One of the following applies:

- All of the following apply (EQUAL):

- * determining whether $\mathbf{t1}$ is *type-equivalent* to $\mathbf{t2}$ in \mathbf{tenv} yields $\mathbf{TRUE} \# \mathbf{TE}$;
- * the result is \mathbf{TRUE} .
- All of the following apply ($\mathbf{DIFFERENT_LABELS_ERROR}$):
 - * determining whether $\mathbf{t1}$ is *type-equivalent* to $\mathbf{t2}$ in \mathbf{tenv} yields \mathbf{FALSE} ;
 - * the AST labels of $\mathbf{t1}$ and $\mathbf{t2}$ are different;
 - * the result is a type error indicating that the type assertion will always fail.
- All of the following apply ($\mathbf{INT_BITS}$):
 - * determining whether $\mathbf{t1}$ is *type-equivalent* to $\mathbf{t2}$ in \mathbf{tenv} yields \mathbf{FALSE} ;
 - * the AST labels of $\mathbf{t1}$ and $\mathbf{t2}$ are the same;
 - * the AST label of $\mathbf{t1}$ is either $\mathbf{T_Int}$ or $\mathbf{T_Bits}$;
 - * the result is \mathbf{TRUE} .
- All of the following apply (\mathbf{TUPLE}):
 - * determining whether $\mathbf{t1}$ is *type-equivalent* to $\mathbf{t2}$ in \mathbf{tenv} yields \mathbf{FALSE} ;
 - * $\mathbf{t1}$ is a tuple type with list of tuples $\mathbf{l1}$, that is, $\mathbf{T_Tuple(l1)}$;
 - * $\mathbf{t1}$ is a tuple type with list of tuples $\mathbf{l2}$, that is, $\mathbf{T_Tuple(l2)}$;
 - * checking whether $\mathbf{l1}$ and $\mathbf{l2}$ have the same length yields $\mathbf{TRUE} \# \mathbf{TypeError(TE_TAF)}$;
 - * applying *check_atc* to $\mathbf{l1[i]}$ and $\mathbf{l2[i]}$ in \mathbf{tenv} for every $\mathbf{i} \in \mathbf{indices(l1)}$ yields $\mathbf{TRUE} \# \mathbf{TE}$;
 - * the result is \mathbf{TRUE} ;
- All of the following apply ($\mathbf{OTHER_ERROR}$):
 - * determining whether $\mathbf{t1}$ is *type-equivalent* to $\mathbf{t2}$ in \mathbf{tenv} yields \mathbf{FALSE} ;
 - * the AST labels of $\mathbf{t1}$ and $\mathbf{t2}$ are the same;
 - * the AST label of $\mathbf{t1}$ is neither $\mathbf{T_Int}$, nor $\mathbf{T_Bits}$, nor $\mathbf{T_Tuple}$;
 - * the result is a type error indicating that the type assertion will always fail.

11.33.2 Formally

$$\begin{array}{c}
 \text{EQUAL} \\
 \hline
 \frac{\text{type_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{TRUE} \# \mathbf{TE}}{\text{check_atc}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{TRUE}} \\
 \\
 \text{DIFFERENT_LABELS_ERROR} \\
 \hline
 \frac{\begin{array}{c} \text{type_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{FALSE} \\ \text{ast_label}(\mathbf{t1}) \neq \text{ast_label}(\mathbf{t2}) \end{array}}{\text{check_atc}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{TypeError(TE_TAF)}}
 \end{array}$$

INT_BITS	
$type_equal(tenv, t1, t2) \xrightarrow{type} FALSE$	
$ast_label(t1) = ast_label(t2) \quad ast_label(t1) \in \{T_Int, T_Bits\}$	
$check_atc(tenv, t1, t2) \xrightarrow{type} TRUE$	
TUPLE	
$type_equal(tenv, t1, t2) \xrightarrow{type} FALSE$	
$t1 = T_Tuple(l1) \quad t2 = T_Tuple(l2) \quad check(l1 = l2 , TE_TAF) \xrightarrow{type} TRUE \quad \#TE$	
$i \in indices(l1) : check_atc(l1[i], l2[i]) \xrightarrow{type} TRUE \quad \#TE$	
$check_atc(tenv, t1, t2) \xrightarrow{type} TRUE$	
OTHER_ERROR	
$type_equal(tenv, t1, t2) \xrightarrow{type} FALSE$	
$ast_label(t1) = ast_label(t2) \quad ast_label(t1) \notin \{T_Int, T_Bits, T_Tuple\}$	
$check_atc(tenv, t1, t2) \xrightarrow{type} TRUE$	

11.34 TypingRule.SlicesWidth

The function

$$slices_width(\overbrace{SE}^{tenv}, \overbrace{slice^*}^{slices}) \longrightarrow \overbrace{expr}^{width} \cup \overbrace{TTypeError}^{\#TE}$$

returns an expression `slices` that represents the width of all slices given by `slices` in the static environment `tenv`.

11.34.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is the empty list;
 - * `width` is the literal integer expression for 0.
- All of the following apply (NON_EMPTY):
 - * `slices` is the list with `head s` and `tail slices1`;
 - * applying `slice_width` to `s` yields `e1`;
 - * applying `slices_width` to `slices1` yields `e2`;
 - * symbolically simplifying the binary expression summing `e1` with `e2` yields `width//#TE`.

11.34.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{width} \\
 \text{E.Literal(L.Int)} \\
 \text{0} \\
 \text{slices} \\
 \text{[]} \\
 \text{type} \\
 \text{slices_width}(\text{tenv}, \text{[]}) \xrightarrow{\text{type}} \text{0} \\
 \\
 \text{NON_EMPTY} \\
 \text{slice_width}(s) \xrightarrow{\text{type}} e1 \\
 \text{slices1} \xrightarrow{\text{type}} e2 \quad \text{normalize}(\overbrace{e1 \text{ PLUS } e2}^{\text{E.Binop}}) \xrightarrow{\text{type}} \text{width} \text{ // \#TE} \\
 \text{slices} \\
 \text{[s] + slices1} \\
 \text{type} \\
 \text{slices_width}(\text{tenv}, \text{[s] + slices1}) \xrightarrow{\text{type}} \text{width}
 \end{array}$$

11.35 TypingRule.SliceWidth

The function

$$\text{slice_width}(\overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\text{expr}}^{\text{width}}$$

returns an expression `slices` that represents the width of the slices given by `slice`.

11.35.1 Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `slice` is a single slice, that is, `Slice_Single(_)`;
 - * `width` is the literal integer expression for 1;
- All of the following apply (STAR, length):
 - * `slice` is either a slice of the form `_:*e` or `_:+e`;
 - * `width` is `e`;
- All of the following apply (RANGE):
 - * `slice` is a slice of the form `e1..e2`;
 - * `width` is the expression for `1 + (e1 - e2)`.

11.35.2 Formally

SINGLE

$$\text{slice_width}(\overbrace{\text{Slice_Single}(_)^{\text{slice}}}^{\text{type}}) \longrightarrow \overbrace{1^{\text{E_Literal}(\text{L_Int})}}^{\text{width}}$$

STAR

$$\text{slice_width}(\overbrace{\text{Slice_Star}(_, e)^{\text{slice}}}^{\text{type}}) \longrightarrow \overbrace{e^{\text{width}}}$$

LENGTH

$$\text{slice_width}(\overbrace{\text{Slice_Length}(_, e)^{\text{slices}}}^{\text{type}}) \longrightarrow \overbrace{e^{\text{width}}}$$

RANGE

$$\text{slice_width}(\overbrace{\text{Slice_Range}(e1, e2)^{\text{slices}}}^{\text{type}}) \longrightarrow \overbrace{\overbrace{1^{\text{E_Literal}(\text{L_Int})}}^{\text{width}} \text{ PLUS } \overbrace{(e1 \text{ MINUS } e2)^{\text{E_Binop}}}^{\text{E_Binop}}}$$

Chapter 12

Typing of Left-Hand-Side Expressions

The function

$$\text{annotate_lexpr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{lexpr}}^{\text{le}}, \overbrace{\text{ty}}^{\text{t.e}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{new_le}} \cup \text{TTypeError}$$

annotates a left-hand side expression `le` in an environment `tenv`, assuming `t.e` to be the type of the corresponding right-hand-side expression, resulting in an annotated expression `new_le`. Otherwise, the result is a type error. One of the following applies:

- `TypingRule.LEDiscard` (see Section 12.1),
- `TypingRule.LEVar` (see Section 12.2),
- `TypingRule.LEDestructuring` (see Section 12.3),
- `TypingRule.LESlice` (see Section 12.4),
- `TypingRule.LESetArray` (see Section 12.5),
- `TypingRule.LESetStructuredField` (see Section 12.6),
- `TypingRule.LESetBadBitField` (see Section ??),
- `TypingRule.LESetBitField` (see Section 12.7),
- `TypingRule.LESetBitFieldNested` (see Section ??),
- `TypingRule.LESetBitFieldTyped` (see Section ??),
- `TypingRule.LESetBadField` (see Section 12.8),
- `TypingRule.LEConcat` (see Section 12.9).

We also make use of the helper tuple `TypingRule.LEBits` (see Section 12.10).

Some of the rules require viewing left-hand-side expressions as their corresponding right-hand side expressions. The correspondence is defined in the ASL Syntax Reference [1, Chapter 5] and given by the function `rexpr : lexpr → expr`.

12.1 TypingRule.LEDiscard

12.1.1 Prose

All of the following apply:

- `le` denotes an expression that can be discarded, that is, `LE_Discard`;
- `new_le` is `le`.

12.1.2 Formally

$$\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Discard}}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{LE_Discard}}^{\text{new_le}}$$

12.2 TypingRule.LEVar

12.2.1 Prose

All of the following apply:

- `le` denotes a left-hand-side variable expression for `x`, that is, `LE_Var(x)`;
- One of the following applies (LOCAL):
 - * `x` is declared in `tenv` as a local storage element with type `ty` and local declaration keyword `k`;
 - * checking that `k` corresponds to a mutable variable, that is, `LDK_Var`, yields `TRUE//TE.AIM`;
 - * determining whether `ty` `type-satisfies` `t_e` in `tenv` yields `TRUE//#TE`;
 - * `new_le` is `le`.
- One of the following applies (GLOBAL):
 - * `x` is declared in `tenv` as a global storage element with type `ty` and global declaration keyword `k`;
 - * checking that `k` corresponds to a mutable variable, that is, `GDK_Var`, yields `TRUE//TE.AIM`;
 - * determining whether `ty` `type-satisfies` `t_e` in `tenv` yields `TRUE//#TE`;
 - * `new_le` is `le`.

- One of the following applies (ERROR_UNDEFINED):
 - * x is not declared in tenv as a local storage element nor as a global storage element;
 - * the result is a type error **TE_UI**.

12.2.2 Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 L^{\text{tenv}}.\text{local_storage_types}(\text{id}) = (\text{ty}, k) \quad \text{check}(k = \text{LDK_Var}, \text{TE_AIM}) \longrightarrow \text{TRUE} \parallel \# \text{TE} \\
 \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{le}}^{\text{new_le}} \\
 \\
 \text{GLOBAL} \\
 L^{\text{tenv}}.\text{global_storage_types}(\text{id}) = (\text{ty}, k) \\
 \text{check}(k = \text{GDK_Var}, \text{AssignToImmutable}) \longrightarrow \text{TRUE} \parallel \# \text{TE} \\
 \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{le}}^{\text{new_le}} \\
 \\
 \text{ERROR_UNDEFINED} \\
 L^{\text{tenv}}.\text{local_storage_types}(\text{id}) = \perp \quad L^{\text{tenv}}.\text{global_storage_types}(\text{id}) = \perp \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})
 \end{array}$$

12.3 TypingRule.LEDestructuring

12.3.1 Prose

All of the following apply:

- le denotes a tuple of left-hand-side expressions les , that is, $\text{LE_Destructuring}(\text{les})$;
- les is a list $e_{1..k}$;
- checking whether t_e is a tuple type yields $\text{TRUE} \parallel \text{TE_ETT}$;
- t_e is a tuple type over the list of types tys , that is, $\text{T_Tuple}(\text{tys})$;
- determining whether les and sub_tys have the same length yields $\text{TRUE} \parallel \text{TE_LMM}$;
- sub_tys is the list of types $\text{t}_{1..k}$;
- annotating the left-hand-side expression e_i with the type t_i , for $i = 1..k$, yields $e'_i \parallel \# \text{TE}$;
- the list of expressions les' is e'_i , for $i = 1..k$;
- new_le is the list of left-hand-side expressions les' , that is, $\text{LE_Destructuring}(\text{les}')$.

12.3.2 Formally

$$\begin{array}{c}
\text{les} \stackrel{\text{is}}{=} [e_{1..k}] \quad \text{check}(\text{ast.label}(\text{t_e}) = \text{T_Tuple}, \text{TE_ETT}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
\text{t_e} \stackrel{\text{is}}{=} \text{T_Tuple}(\text{tys}) \\
\text{equal.length}(\text{les}, \text{tys}) \xrightarrow{\text{type}} \text{b} \quad \text{check}(\text{b}, \text{TE_LMM}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
\text{tys} \stackrel{\text{is}}{=} [\text{t}_{1..k}] \quad i = 1..k : \text{annotate.lexpr}(\text{tenv}, e_i, \text{t}_i) \xrightarrow{\text{type}} e'_i \text{ // } \# \text{TE} \\
\text{les}' \stackrel{\text{is}}{=} [i = 1..k : e'_i] \\
\hline
\text{annotate.lexpr}(\text{tenv}, \overbrace{\text{LE_Deconstructing}(\text{les})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{LE_Deconstructing}(\text{les}')}^{\text{new_le}}
\end{array}$$

12.4 TypingRule.LESlice

12.4.1 Prose

All of the following apply:

- `le` denotes the slicing of a left-hand-side expression `le1` by the slices `slices`, that is, `LE_Slice(le1, slices)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields $(\text{t_le1}, _) \text{ // } \# \text{TE}$;
- `t_le1` is a bitvector type;
- annotating the left-hand-side expression `le1` in `tenv` yields `le2 // #TE`;
- obtaining the width of the slices `slices` in `tenv` and simplifying them yields `width`;
- `t` is the bitvector type of width `width` and empty list of bitfields;
- checking whether `t_e` *type-satisfies* `t` yields `TRUE // #TE`;
- annotating `slices` in `tenv` yields `slices2 // #TE`;
- checking that the slices `slices2` are all disjoint yields `TRUE // #TE`;
- `new_le` is the slicing of `le2` by `slices2`, that is, `LE_Slice(le2, slices2)`.

12.4.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{struct_t_le1} \text{ // \#TE} \\
\text{ast_label}(\text{struct_t_le1}) = \text{T_Bits} \quad \text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{width}' \quad \text{normalize}(\text{tenv}, \text{width}') \xrightarrow{\text{type}} \text{width} \\
\text{t} := \text{T_Bits}(\text{width}, [_]) \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices2} \text{ // \#TE} \\
\text{check_disjoint_slices}(\text{tenv}, \text{slices2}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{new_le} := \text{LE_Slice}(\text{le2}, \text{slices2}) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Slice}(\text{le1}, \text{slices})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

12.5 TypingRule.LESetArray

12.5.1 Prose

All of the following apply:

- `le` denotes the slicing of a left-hand-side expression `le1` by the slices `slices`, that is, `LE.Slice(le1, slices)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _)` *//* *#TE*;
- obtaining the *structure* of `t_le1` in `tenv` yields an array type of size `size` and element type `t`, that is, `T.Array(size, t)` *//* *#TE*;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields `le2` *//* *#TE*;
- determining that `t_e` *type-satisfies* `t` in `tenv` yields `TRUE` *//* *#TE*;
- determining whether `slices` is a single slice with index expression `e_index` yields `TRUE` *//* *#TE*;
- annotating the index expression `e_index` in `tenv` yields `(t_index', e_index')` *//* *#TE*;
- determining the array length type of `size` in `tenv` (via *type_of_array_length*) yields `wanted_t_index`;
- determining whether `t_index'` *type-satisfies* `wanted_t_index` in `tenv` yields `TRUE` *//* *#TE*;
- `new_le` is an access to array `le2` at index `e_index'`, that is, `LE.SetArray(le2, e_index')`.

12.5.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Array}(\text{size}, \text{t}) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{check}(|\text{slices}| = 1, \text{ArraySliceShouldBeSingleIndex}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\quad \text{slices} \stackrel{\text{is}}{=} [\text{s}] \\
\text{check}(\text{ast_label}(\text{s}) = \text{Slice_Single}, \text{ArraySliceShouldBeSingleIndex}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\quad \text{s} \stackrel{\text{is}}{=} \text{Slice_Single}(\text{e_index}) \\
\text{annotate_expr}(\text{tenv}, \text{e_index}) \xrightarrow{\text{type}} (\text{t_index}', \text{e_index}') \text{ // \#TE} \\
\text{type_of_array_length}(\text{tenv}, \text{size}) \xrightarrow{\text{type}} \text{wanted_t_index} \\
\text{checked_typesat}(\text{tenv}, \text{t_index}', \text{wanted_t_index}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{new_le} := \text{LE_SetArray}(\text{le2}, \text{e_index}') \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Slice}(\text{le1}, \text{slices})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

12.6 TypingRule.LESetStructuredField

12.6.1 Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields $(\text{t_le1}, _) \text{ // \#TE}$;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields `le2 // \#TE`;
- obtaining the `structure` of `t_le1` in `tenv` yields `aststructured type` with fields `fields // \#TE`;
- checking that there exists a type associated with the field `field` in `fields` `TRUE // \#TE`;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` `type-satisfies` `t` yields `TRUE // \#TE`;
- `new_le` is the access to the field `field` in `le2`, that is, `LE_SetField(le2, field)`.

12.6.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} L(\text{fields}) \text{ // \#TE} \\
L \in \{\text{T_Exception}, \text{T_Record}\} \quad \text{assoc_opt}(\text{fields}, \text{field}) \xrightarrow{\text{type}} \text{ty_opt} \\
\text{check}(\text{ty_opt} \neq \text{None}, \text{TE_MF}) \rightarrow \text{TRUE} \text{ // \#TE} \\
\text{ty_opt} \stackrel{\text{is}}{=} \langle \text{t} \rangle \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{new_le} := \text{LE.SetField}(\text{le2}, \text{field}) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE.SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

12.7 TypingRule.LESetBitField

12.7.1 Prose

12.7.2 Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE.SetField(le1, field)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _)` // #TE;
- annotating the left-hand-side expression `le1` in `tenv` yields `le2` // #TE;
- obtaining the `structure` of `t_le1` in `tenv` yields a bitvector type with with bitfields `bitfields` // #TE;
- One of the following applies:
 - * All of the following applies (ERROR_MISSING_FIELD):
 - applying `find_bitfield_opt` to `bitfields` and `field` yields `None`, meaning the field is not declared in `t_le1`;
 - the result is a type error `TE_MF`.
 - * All of the following applies (FIELD_SIMPLE):
 - applying `find_bitfield_opt` to `bitfields` and `field` yields a bitfield with corresponding slices `slices`, that is, `BitField.Simple(_, slices)`;
 - `w` is the width of `slices`;
 - `t` is defined as the bitvector type of width `w` and empty list of bitfields, that is, `T_Bits(w, [])`;
 - checking whethert_e `type-satisfies` `t` in `tenv` yields `TRUE` // #TE;

- `le2` is defined as the slicing of `le1` by `slices`, that is,
`LE.Slice(le1, slices)`;
 - annotating the left-hand-side expression `le2` in `tenv` yields `new_le` *//* `#TE`.
- * All of the following applies (`FIELD_NESTED`):
- applying *`find_bitfield_opt`* to `bitfields` and `field` yields a nested bitfield with corresponding slices `slices` and list of bitfields `bitfields'`, that is,
`BitField.Nested(_, slices, bitfields')`;
 - `w` is the width of `slices`;
 - `t` is defined as the bitvector type of width `w` and list of bitfields `bitfields'`, that is, `T.Bits(w, bitfields')`;
 - checking whether `t.e` *type-satisfies* `t` in `tenv` yields `TRUE` *//* `#TE`;
 - `le3` is defined as the slicing of `le1` by `slices`, that is,
`LE.Slice(le1, slices)`;
 - annotating the left-hand-side expression `le3` in `tenv` yields `new_le` *//* `#TE`.
- * All of the following applies (`FIELD_TYPED`):
- applying *`find_bitfield_opt`* to `bitfields` and `field` yields a typed bitfield with corresponding slices `slices` and a type `t`, that is,
`BitField.Type(_, slices, t)`;
 - `w` is the width of `slices`;
 - `t'` is defined as the bitvector type of width `w` and an empty list of bitfields, that is, `T.Bits(w, [])`;
 - checking whether `t'` *type-satisfies* `t` in `tenv` yields `TRUE` *//* `#TE`;
 - checking whether `t.e` *type-satisfies* `t` in `tenv` yields `TRUE` *//* `#TE`;
 - `le2` is defined as the slicing of `le1` by `slices`, that is,
`LE.Slice(le1, slices)`;
 - annotating the left-hand-side expression `le2` in `tenv` yields `new_le` *//* `#TE`.

12.7.3 Formally

ERROR_MISSING_FIELD

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, \text{repr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // } \#TE \\
 \text{***** common prefix *****} \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE.SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t.e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_MF})
 \end{array}$$

FIELD_SIMPLE

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \quad \text{t} := \text{T_Bits}(\text{w}, [\]) \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

FIELD_NESTED

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \quad \text{t} := \text{T_Bits}(\text{w}, \text{bitfields}') \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{MissingField})
\end{array}$$

FIELD_TYPED

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Type}(_, \text{slices}, \text{t}) \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \\
\text{t}' := \text{T_Bits}(\text{w}, [\]) \quad \text{checked_typesat}(\text{tenv}, \text{t}', \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{MissingField})
\end{array}$$

12.8 TypingRule.LESetBadField

12.8.1 Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE_SetField(le1, field)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _) // #TE`;
- annotating the left-hand-side expression `le1` in `tenv` yields `le2 // #TE`;
- obtaining the `structure` of `t_le1` in `tenv` yields a type `t // #TE`;
- `t` is neither a `structured type` nor a bitvector type;
- the result is an error indicating that the type of `le` conflicts with the requirements of a field access expression.

12.8.2 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{t} \text{ // } \#TE \\
 \text{ast_label}(\text{t}) \notin \{\text{T_Exception}, \text{T_Record}, \text{T_Bits}\}
 \end{array}
 }{
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
 }$$

12.9 TypingRule.LEConcat

12.9.1 Prose

All of the following apply:

- `le` denotes the concatenation of left-hand-side expressions `les`, that is, `LE_Concat(les, _)`;
- annotating the right-hand-side expression corresponding to `le` in `tenv` yields `(t_e_eq, _) // #TE`;
- checking whether the bitwidth of `t_e_eq` equals the bitwidth of `t_e` in `tenv` yields `TRUE // #TE`;
- `les` is the list of left-hand-side expressions `lei`, for $i = 1..k$;

- annotating each left-hand-side expression le_i as a bitvector-typed expression (via *annotate_lebits*) yields the annotated left-hand-side expression $le1_i$ and corresponding bitwidth $width_i$, for $i = 1..k$;
- $les1$ is defined as the list $le1_{1..k}$;
- $width_s$ is defined as the list $width_{1..k}$;
- new_le is the concatenation of left-hand-side expressions $les1$ with corresponding list of widths $width_s$.

12.9.2 Formally

$$\begin{array}{c}
 le \stackrel{is}{=} LE_Concat(les, _) \quad \text{annotate_expr}(\text{tenv}, \text{repr}(le)) \xrightarrow{\text{type}} (t_e_eq, _) \text{ // } \#TE \\
 \quad \quad \quad \text{check_bits_equal_width}(\text{tenv}, t_e_eq, t_e) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 les \stackrel{is}{=} le_{1..k} \quad i = 1..k : \text{annotate_lebits}(\text{tenv}, le_i) \xrightarrow{\text{type}} (le1_i, width_i) \text{ // } \#TE \\
 \quad \quad \quad les1 := le1_{1..k} \quad width_s := width_{1..k} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, le, t_e) \xrightarrow{\text{type}} \overbrace{LE_Concat(les1, width_s)}^{new_le}
 \end{array}$$

12.10 TypingRule.LEBits

The helper function

$$\text{annotate_lebits}(\overbrace{SE}^{\text{tenv}}, \overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{le1}} \times \overbrace{N}^{\text{width}}$$

annotates a left-hand-side expression le , which is checked to be of bitvector type with width $width$, resulting in the annotated expression and $width$, or a type error, if one is detected.

All of the following apply:

- annotating the right-hand-side expression corresponding to le in $tenv$ yields $(t_e1, _) // \#TE$;
- obtaining the *structure* of t_e1 in $tenv$ yields $t_e1_struct // \#TE$;
- checking whether t_e1_struct is a bitvector type yields $\text{TRUE} // \#TE$;
- t_e1_struct is a bitvector type with width e_width ;
- applying *reduce_constants* to e_width yields the literal $1 // \#TE$;
- checking whether 1 is an integer literal yields $\text{TRUE} // \#TE$;
- 1 is the integer literal for the integer $width$;

- t_e2 is defined as the bitvector type of width given by $width$ and an empty list of bitfields, that is, $T_Bits(\overbrace{\text{width}}^{E_Literal(L_Int)}, [])$;
- annotating the left-hand-side expression t_e2 in $tenv$ yields $le1 \#TE$.

12.10.1 Formally

$$\begin{array}{c}
\text{annotate_expr}(tenv, \text{repr}(le1)) \xrightarrow{\text{type}} (t_e1, _) \parallel \#TE \\
\text{get_structure}(tenv, t_e1) \xrightarrow{\text{type}} t_e1_struct \parallel \#TE \\
\text{check}(\text{ast_label}(t_e1_struct) = T_Bits, \text{BitvectorTypeExpected}) \longrightarrow \text{TRUE} \parallel \#TE \\
t_e1_struct \stackrel{\text{is}}{=} T_Bits(e_width, _) \quad \text{reduce_constants}(tenv, e_width) \xrightarrow{\text{type}} 1 \\
\text{check}(\text{ast_label}(1) = L_Int, \text{IntegerLiteralExpected}) \longrightarrow \text{TRUE} \parallel \#TE \\
1 \stackrel{\text{is}}{=} L_Int(width) \\
\text{t_e2} := T_Bits(\overbrace{\text{width}}^{E_Literal(L_Int)}, []) \quad \text{annotate_lexpr}(tenv, t_e2) \xrightarrow{\text{type}} le1 \parallel \#TE \\
\hline
\text{annotate_lebits}(tenv, le) \xrightarrow{\text{type}} (le1, width)
\end{array}$$

Chapter 13

Typing of Slices

The function

$$\text{annotate_slice}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\mathbf{s}}) \longrightarrow \overbrace{\text{slice}}^{\mathbf{s}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a single slice \mathbf{s} in the static environment tenv , resulting in an annotated slice \mathbf{s}' . Otherwise, the result is a type error.

One of the following applies:

- `TypingRule.SliceSingle` (see Section 13.1),
- `TypingRule.SliceLength` (see Section 13.2),
- `TypingRule.SliceRange` (see Section 13.3),
- `TypingRule.SliceStar` (see Section 13.4).

We also define a rule for typing of a list of slices: `TypingRule.Slices` (see Section 13.5).

The function

$$\text{annotate_slices}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\text{slice}^*}^{\text{slices}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of slices slices in the static environment tenv , resulting in an annotated list of slices slices' . Otherwise, the result is a type error.

The relevant rule is given by:

- `TypingRule.Slices` (see Section 13.5)

13.1 TypingRule.SliceSingle

13.1.1 Prose

All of the following apply:

- \mathbf{s} is a slice at index \mathbf{i} , that is `Slice.Single(i)`;
- annotating the slice at offset \mathbf{i} of length 1 yields $\mathbf{s}' \text{ // } \#TE$.

13.1.2 Formally

$$\frac{\text{annotate_slice}(\text{Slice_Length}(i, \text{E_Literal}(1))) \xrightarrow{\text{type}} s' \quad // \quad \#TE}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Single}(i)}^s) \xrightarrow{\text{type}} s'}$$

13.2 TypingRule.SliceLength

13.2.1 Prose

All of the following apply:

- s is a slice of length length and offset offset , that is, $\text{Slice_Length}(\text{offset}, \text{length})$;
- annotating the expression offset in tenv yields $(t_offset, \text{offset}') // \#TE$;
- annotating the *statically evaluable constrained integer* expression length in tenv yields $\text{length}' // \#TE$;
- determining whether t_offset has the *structure of an integer* yields $\text{TRUE} // \#TE$;
- s' is the slice at offset offset' and length length' , that is, $\text{Slice_Length}(\text{offset}', \text{length}')$.

13.2.2 Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{offset}) \xrightarrow{\text{type}} (t_offset, \text{offset}') \quad // \quad \#TE \\ \text{annotate_static_constrained_integer}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} \text{length}' \quad // \quad \#TE \\ \text{check_structure_integer}(\text{tenv}, t_offset) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \end{array}}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Length}(\text{offset}, \text{length})}^s) \xrightarrow{\text{type}} \overbrace{\text{Slice_Length}(\text{offset}', \text{length}')}^{s'}}$$

13.3 TypingRule.SliceRange

13.3.1 Prose

All of the following apply:

- s is a slice for the range (j, i) , that is $\text{Slice_Range}(j, i)$;
- pre_length is $i - (j - i + 1)$;
- annotating the slice at offset i of length pre_length yields $s' // \#TE$.

13.3.2 Formally

$$\frac{
\begin{array}{l}
\text{binop_literals}(\text{MINUS}, i, i) \xrightarrow{\text{type}} \text{pre_length}' \\
\text{binop_literals}(\text{PLUS}, \text{pre_length}', \text{E_Literal}(1)) \xrightarrow{\text{type}} \text{pre_length} \\
\text{annotate_slice}(\text{Slice_Length}(i, \text{pre_length})) \xrightarrow{\text{type}} s' \quad // \quad \#TE
\end{array}
}{
\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Range}(j, i)}^s) \xrightarrow{\text{type}} s'
}$$

13.4 TypingRule.SliceStar

13.4.1 Prose

All of the following apply:

- s is a slice $[\text{factor} * : \text{pre_length}]$, that is, $\text{Slice_Star}(\text{factor}, \text{pre_length})$;
- pre_offset is $\text{factor} * \text{pre_length}$;
- annotating the slice at offset pre_offset of length pre_length yields $s' // \#TE$.

13.4.2 Formally

$$\frac{
\begin{array}{l}
\text{binop_literals}(\text{MUL}, \text{factor}, \text{pre_length}) \xrightarrow{\text{type}} \text{pre_offset} \\
\text{annotate_slice}(\text{Slice_Length}(\text{pre_offset}, \text{pre_length})) \xrightarrow{\text{type}} s' \quad // \quad \#TE
\end{array}
}{
\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Star}(\text{factor}, \text{pre_length})}^s) \xrightarrow{\text{type}} s'
}$$

13.5 TypingRule.Slices

13.5.1 Prose

All of the following apply:

- annotating the slice $\text{slices}[i]$ in tenv , for each $i \in \text{indices}(\text{slices})$, yields the slice $s_i // \#TE$;
- define slices' as the list of slices s_i , for each $i \in \text{indices}(\text{slices})$.

13.5.2 Formally

$$\frac{
\begin{array}{l}
i \in \text{indices}(\text{slices}) : \text{annotate_slice}(\text{tenv}, \text{slices}[i]) \xrightarrow{\text{type}} s_i \quad // \quad \#TE \\
\text{slices}' := [i \in \text{indices}(\text{slices}) : s_i]
\end{array}
}{
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices}'
}$$

Chapter 14

Typing of Patterns

The function

$$\text{annotate_pattern}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{pattern}}^{\text{p}}) \longrightarrow \overbrace{\text{pattern}}^{\text{new_p}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a pattern p in a static environment tenv given a type t , resulting in a pattern new_p or a type error, if one is detected, and one of the following applies:

- `TypingRule.PAll` (see Section 14.1),
- `TypingRule.PAny` (see Section 14.2),
- `TypingRule.PGeq` (see Section 14.3),
- `TypingRule.PLeq` (see Section 14.4),
- `TypingRule.PNot` (see Section 14.5),
- `TypingRule.PRange` (see Section 14.6),
- `TypingRule.PSingle` (see Section 14.7),
- `TypingRule.PMask` (see Section 14.8),
- `TypingRule.PTuple` (see Section 14.9).

14.1 TypingRule.PAll

14.1.1 Prose

All of the following apply:

- p is the pattern matching everything, that is, `Pattern.All`;
- new_p is p .

14.1.2 Formally

$$\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_All}}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_All}}^{\text{new_p}}$$

14.2 TypingRule.PAny

14.2.1 Prose

All of the following apply:

- p is the pattern which matches anything in a list li , that is, $\text{Pattern_Any}(li)$;
- annotating each pattern in li yields the list of annotated pattern $\text{new_li} \text{ // } \#TE$;
- new_p is the pattern which matches anything in new_li , that is, $\text{Pattern_Any}(\text{new_li})$.

14.2.2 Formally

$$\frac{l \in li : \text{annotate_pattern}(\text{tenv}, t, l) \xrightarrow{\text{type}} l' \text{ // } \#TE \quad \text{new_li} := [l \in li : l']}{\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Any}(li)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Any}(\text{new_li})}^{\text{new_p}}}$$

14.3 TypingRule.PGeq

14.3.1 Prose

All of the following apply:

- p is the pattern which matches anything greater than or equal to an expression e , that is, $\text{Pattern_Geq}(e)$;
- annotating the expression e in tenv yields $(t_e, e') \text{ // } \#TE$;
- determining whether e' is a **statically evaluable** expression yields $\text{TRUE} \text{ // } \#TE$;
- obtaining the **structure** of t in tenv yields $t_struct \text{ // } \#TE$;
- obtaining the **structure** of t_e in tenv yields $t_e_struct \text{ // } \#TE$;
- b is true if and only if t_struct and t_e_struct are both integer types or both real types;
- if b is **FALSE** a type error is returned (indicating that the types of t and t_e are inappropriate for the **GEQ** operator), which short-circuits the entire rule;
- new_p is the pattern which matches anything greater than or equal to e' .

14.3.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \text{ // \#TE} \\
\text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ // \#TE} \\
b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
\quad \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
\text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \underbrace{\text{Pattern_Geq}(e)}_p) \xrightarrow{\text{type}} \underbrace{\text{Pattern_Geq}(e')}_{\text{new_p}}
\end{array}$$

14.4 TypingRule.PLeq

14.4.1 Prose

All of the following apply:

- p is the pattern which matches anything less than or equal to an expression e , that is, $\text{Pattern_Leq}(e)$;
- annotating the expression e in tenv yields $(t_e, e') \text{ // \#TE}$;
- determining whether e' is a **statically evaluable** expression yields $\text{TRUE} \text{ // \#TE}$;
- obtaining the **structure** of t in tenv yields $t_struct \text{ // \#TE}$;
- obtaining the **structure** of t_e in tenv yields $t_e_struct \text{ // \#TE}$;
- b is true if and only if t_struct and t_e_struct are both integer types or both real types;
- if b is **FALSE** a type error is returned (indicating that the types of t and t_e are inappropriate for the LEQ operator), which short-circuits the entire rule;
- new_p is the pattern which matches anything less than or equal to e' .

14.4.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \text{ // } \#TE \\
\text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ // } \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ // } \#TE \\
b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
\text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
\text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Leq}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Leq}(e')}^{\text{new_p}}
\end{array}$$

14.5 TypingRule.PNot

14.5.1 Prose

All of the following apply:

- p is the pattern which matches the negation of a pattern q , that is, $\text{Pattern_Not}(q)$;
- annotating q in tenv yields $\text{new_q} \text{ // } \#TE$;
- new_p is pattern which matches the negation of new_q , that is, $\text{Pattern_Not}(\text{new_q})$.

14.5.2 Formally

$$\begin{array}{c}
\text{annotate_pattern}(\text{tenv}, q) \xrightarrow{\text{type}} \text{new_q} \text{ // } \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Not}(q)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Not}(\text{new_q})}^{\text{new_p}}
\end{array}$$

14.6 TypingRule.PRange

14.6.1 Prose

All of the following apply:

- p is the pattern which matches anything within the range given by expressions $e1$ and $e2$, that is, $\text{Pattern_Range}(e1, e2)$;
- annotating the expression $e1$ in tenv yields $(t_e1, e1') \text{ // } \#TE$;
- annotating the expression $e2$ in tenv yields $(t_e2, e2') \text{ // } \#TE$;
- determining whether both $e1'$ and $e2'$ are compile-time constant expressions yields $\text{TRUE} \text{ // } \#TE$;

- obtaining the `structure` for `t`, `t_e1`, and `t_e2` yields `t_struct`, `t_e1_struct`, and `t_e2_struct`, respectively `// #TE`;
- a check the AST labels of `t_struct`, `t_e1_struct`, and `t_e2_struct` are all the same and are either `T_Int` or `T_Real` yields `TRUE`. Otherwise, the result is a type error, which short-circuits the entire rule. The type error indicates that the types of `e1`, `e2` and the type `t` must be either of integer type or of real type.
- `new_p` is a range pattern with bounds `e1'` and `e2'`, that is, `Pattern_Range(e1', e2')`.

14.6.2 Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e1') \quad // \quad \#TE \\
\text{annotate_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t_e2, e2') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e1_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e2) \xrightarrow{\text{type}} t_e2_struct \quad // \quad \#TE \\
b := \text{ast_label}(t_struct) = \text{ast_label}(t_e1_struct) = \text{ast_label}(t_e2_struct) \wedge \\
\text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
\text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Range}(e1, e2)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Range}(e1', e2')}^{\text{new_p}}
\end{array}$$

14.7 TypingRule.PSingle

14.7.1 Prose

All of the following apply:

- `p` is the pattern that matches the expression `e`, that is, `Pattern_Single(e)`;
- annotating the expression `e` in `tenv` yields `(t_e, e')` `// #TE`;
- obtaining the `structure` of `t` yields `t_struct` `// #TE`;
- obtaining the `structure` of `t_e` yields `t_e_struct` `// #TE`;
- One of the following holds:
 - * All of the following apply (`T_BOOL`, `T_REAL`, `T_INT`):
 - the AST label of `t_struct` is one of `T_Bool`, `T_Real`, or `T_Int`;
 - checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE` `// #TE`;
 - * All of the following apply (`T_BITS`):
 - the AST label of `t_struct` is `T_Bits`;

- checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE//#TE`;
- determining whether the bitwidths of `t_struct` and `t_e_struct` are equal yields `TRUE//#TE`;
- * All of the following apply (`T_ENUM`):
 - the AST label of `t_struct` is `T_Enum`;
 - checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE//#TE`;
 - determining whether the lists of enumeration literals of `t_struct` and `t_e_struct` are equal yields `TRUE//#TE`;
- * All of the following apply (`ERROR`):
 - determining whether the labels of `t_struct` and `t_e_struct` are the same yields `TRUE//#TE`;
 - the label of `t_struct` is not one of `T_Bool`, `T_Real`, `T_Int`, `T_Bits`, or `T_Enum`;
 - the result is a type error indicating that the types `t` and `t_e` are inappropriate for this pattern.
- `new_p` is the pattern that matches the expression `e'`, that is, `Pattern_Single(e')`.

14.7.2 Formally

`T_BOOL`, `T_REAL`, `T_INT`

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \text{ast_label}(t_struct) \in \{T_Bool, T_Real, T_Int\} \\
 \text{check}(\text{ast_label}(t_struct) = \text{ast_label}(t_e_struct), TE_OTB) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \underbrace{\text{Pattern_Single}(e)}_p) \xrightarrow{\text{type}} \underbrace{\text{Pattern_Single}(e')}_{\text{new_p}}
 \end{array}$$

`T_BITS`

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \text{ast_label}(t_struct) = T_Bits \\
 \text{check}(\text{ast_label}(t_struct) = \text{ast_label}(t_e_struct), TE_OTB) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
 \text{bitwidth_equal}(\text{tenv}, t_struct, t_e_struct) \xrightarrow{\text{type}} b \\
 \text{check}(b, \text{BitvectorsDifferentWidths}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \underbrace{\text{Pattern_Single}(e)}_p) \xrightarrow{\text{type}} \underbrace{\text{Pattern_Single}(e')}_{\text{new_p}}
 \end{array}$$

T_ENUM

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) = T_Enum \\
check(ast_label(t_struct) = ast_label(t_e_struct), TE_OTB) \longrightarrow TRUE \quad // \quad \#TE \\
t_struct \stackrel{\text{is}}{=} T_Enum(li1) \quad t_e_struct \stackrel{\text{is}}{=} T_Enum(li2) \\
check(li1 = li2, EnumDifferentLabels) \longrightarrow TRUE \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}
\end{array}$$

ERROR

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
check(ast_label(t_struct) = ast_label(t_e_struct), TE_OTB) \longrightarrow TRUE \quad // \quad \#TE \\
ast_label(t_struct) \notin \{T_Bool, T_Real, T_Int, T_Bits, T_Enum\} \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
\end{array}$$

14.8 TypingRule.PMask

14.8.1 Prose

All of the following apply:

- p is the pattern which matches a mask m , that is, $\text{Pattern_Mask}(m)$;
- determining whether t has the structure of a bitvector type yields $TRUE // \#TE$;
- n is the length of mask m ;
- determining whether t *type-satisfies* the bitvector type of length n (that is, $T_Bits(n, [])$), yields $TRUE // \#TE$;
- new_p is p .

14.8.2 Formally

$$\begin{array}{c}
check_structure(\text{tenv}, t, T_Bits) \xrightarrow{\text{type}} TRUE \quad // \quad \#TE \\
n := |m| \quad checked_typesat(\text{tenv}, t, T_Bits(n, [])) \xrightarrow{\text{type}} TRUE \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Mask}(m)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Mask}(m)}^{\text{new_p}}
\end{array}$$

14.9 TypingRule.PTuple

14.9.1 Prose

All of the following apply:

- p is the pattern which matches a tuple li , that is, $\text{Pattern_Tuple}(li)$;
- obtaining the `structure` of t yields $t_struct \# \# \text{TE}$;
- determining whether t_struct is a tuple type yields $\text{TRUE} \# \# \text{TE}$;
- t_struct is a tuple type with list of tuple t_s ;
- determining whether t_s is a list of the same size as li yields $\text{TRUE} \# \# \text{TE}$;
- annotating each pattern in li with the corresponding type in t_s at each position i yields a pattern $li'[i] \# \# \text{TE}$;
- new_li is the list of annotated patterns $li'[i]$ at the same positions those of li ;
- new_p is the pattern which matches the tuple new_li , that is, $\text{Pattern_Tuple}(new_li)$.

14.9.2 Formally

$$\begin{array}{c}
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \# \# \text{TE} \\
 \text{check}(\text{ast_label}(t_struct) = \text{T_Tuple}, \text{TypeConflict}) \longrightarrow \text{TRUE} \# \# \text{TE} \\
 t_struct \stackrel{\text{is}}{=} \text{T_Tuple}(t_s) \\
 \text{check}(\text{equal_length}(li, t_s), \text{InvalidArity}) \longrightarrow \text{TRUE} \# \# \text{TE} \\
 i \in \text{indices}(li) : \text{annotate_pattern}(\text{tenv}, t_s[i], li[i]) \xrightarrow{\text{type}} li'[i] \# \# \text{TE} \\
 new_li := i \in \text{indices}(li) : li'[i] \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Tuple}(li)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Tuple}(new_li)}^{new_p}
 \end{array}$$

Chapter 15

Typing of Local Declarations

The function

$$\text{annotate_local_decl_item}(\overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}, \overbrace{\text{local_decl_item}}^{\text{ldi}}) \longrightarrow \\ (\overbrace{\text{SE}}^{\text{new_tenv}}, \overbrace{\text{local_decl_item}}^{\text{new_ldi}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a local declaration item `ldi` with a local declaration keyword `ldk`, given a type `ty`, in a static environment `tenv` results in `(new_env, new_ldi)` where `new_env` is the modified static environment and `new_ldi` is the annotated local declaration item. Otherwise, the result is a type error.

One of the following applies:

- `TypingRule.LDDiscard` (see Section 15.1),
- `TypingRule.LDVar` (see Section 15.2),
- `TypingRule.LDTyped` (see Section 15.3),
- `TypingRule.LDTuple` (see Section 15.4).

We also define the following helper rule: `TypingRule.CheckCanBeInitializedWith` Section 15.5.

15.1 TypingRule.LDDiscard

15.1.1 Prose

All of the following apply:

- `ldi` is a local declaration which can be discarded, that is, `LDI_Discard(None)`;
- `new_env` is `tenv`;
- `new_ldi` is `ldi`.

15.1.2 Example

```
func main () => integer
begin

  let - = 42;
  let - = "abc";
  let - = '101010';

  return 0;
end
```

15.1.3 Formally

$$\text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{LDI_Discard}(\text{None}), \text{ldk}) \xrightarrow{\text{type}} (\text{tenv}, \text{ldi})$$

15.2 TypingRule.LDVar

15.2.1 Prose

All of the following apply:

- `ldi` denotes a variable `x`, that is, `LDI_Var(x)`;
- determining whether `x` is not declared in `tenv` yields `TRUE // #TE`;
- `new_env` is `tenv` modified so that `x` is locally declared to have type `ty`;
- `new_ldi` is the declaration of variable `x`.

15.2.2 Example

```
func main () => integer
begin
  let x = 3;
  assert x == 3;

  return 0;
end
```

15.2.3 Formally

$$\frac{\begin{array}{c} \text{check_var_not_in_env}(\text{tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{add_local}(\text{tenv}, x, \text{ty}, \text{ldk}) \xrightarrow{\text{type}} \text{new_tenv} \end{array}}{\text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{LDI_Var}(x), \text{ldk}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{LDI_Var}(x))}$$

15.3 TypingRule.LDTyped

15.3.1 Prose

All of the following apply:

- `ldi` denotes a local declaration item `ldi'` with local declaration keyword `ldk` and a type `t`, that is `LDI_Typed(ldi', t)`;
- annotating the type `t` in `tenv` yields `t' // #TE`;
- determining whether `t'` can be initialized with `ty` in `tenv` yields `TRUE // #TE`;
- annotating the local declaration item `ldi'` with the local declaration keyword `ldk`, given the type `t`, in the environment `tenv`, yields `(new_tenv, new_ldi')`;
- `new_ldi` is the local declaration denoting `new_ldi'` and the type `t'`, that is, `LDI_Typed(new_ldi', t')`.

15.3.2 Example

```
type MyT of integer;

func foo (t: MyT) => integer
begin
  return t as integer;
end

func main () => integer
begin
  let x: MyT = 42;
  var z: MyT;

  assert foo (x) == 42;
  assert foo (z) == 0;

  return 0;
end
```

15.3.3 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t' \text{ // } \#TE \\
 \text{check_can_be_initialized_with}(\text{tenv}, t', ty) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_local_decl_item}(\text{tenv}, t', \text{ldi}', \text{ldk}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi}') \text{ // } \#TE
 \end{array}
 }{
 \text{annotate_local_decl_item}(\text{tenv}, ty, \text{LDI_Typed}(\text{ldi}', t), \text{ldk}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{LDI_Typed}(\text{new_ldi}', t'))
 }$$

15.4 TypingRule.LDTuple

15.4.1 Prose

All of the following apply:

- `ldi` denotes a tuple of local declaration items `ldi1..k`, that is, `LDTuple(ldi1..k)`;
- determining the `structure` of `ty` in `tenv` yields `t' // #TE`;
- determining whether `t'` is a tuple type yields `TRUE // #TE`;
- determining whether `t'` the number of elements of `t'` is `k` yields `TRUE // #TE`;
- annotating the local declaration items in `ldi`s from right to left with their corresponding (that is, with the same index) types `t1..k` in `tenv`, propagating static environments from one annotation to the next, yields the local declaration items `ldi' 1..k // #TE`;
- `new_tenv` is the static environment yielded by annotating `ldi1`;
- `new_ldi` is a tuple of local declaration items with `ldi' 1..k`, that is, `LDTuple(ldi' 1..k)`.

15.4.2 Example

```
type MyT of (integer, integer {0..4}, boolean);
```

```
func main() => integer
begin
  let (x, -, y) = (5, 3, TRUE);

  assert x == 5 && y;
  return 0;
end
```

15.4.3 Formally

$$\begin{array}{c}
 \text{get_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t}' \text{ // } \#TE \\
 \text{check}(\text{ast_label}(\text{t}') = \text{T_Tuple}, \text{TupleTypeExpected}) \longrightarrow \text{TRUE // } \#TE \\
 \text{t}' \stackrel{\text{is}}{=} \text{T_Tuple}([t_{1..n}]) \\
 \text{check}(k = n, \text{InvalidArity}) \longrightarrow \text{TRUE // } \#TE \\
 \text{new_tenv}_k = \text{tenv} \\
 i = k..1 : \text{annotate_local_decl_item}(\text{new_tenv}_i, t_i, \text{ldi}_i, \text{ldk}) \xrightarrow{\text{type}} \\
 (\text{new_tenv}_{i-1}, \text{ldi}'_i) \text{ // } \#TE \\
 \text{new_tenv} = \text{new_tenv}_0 \\
 \hline
 \text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{LDTuple}(\text{ldi}_{1..k}), \text{ldk}) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{LDTuple}(\text{ldi}'_{1..k}))
 \end{array}$$

15.5 TypingRule.CheckCanBeInitializedWith

The function

$$check_can_be_initialized_with(\overbrace{SE}^{tenv}, \overbrace{ty}^s, \overbrace{ty}^t) \xrightarrow{type} \{TRUE\} \cup \overbrace{TTypeError}^{\#TE}$$

checks whether an expression of type s can be used to initialize a storage element of type t in the static environment $tenv$. If the answer is **FALSE**, the result is a type error.

15.5.1 Prose

One of the following applies:

- All of the following apply (OKAY):
 - * testing whether t **type-satisfies** s in $tenv$ yields **TRUE**;
 - * the result is **TRUE**.
- All of the following apply (ERROR):
 - * testing whether t **type-satisfies** s in $tenv$ yields **FALSE**;
 - * the result is a type error indicating that an expression of type s cannot be used to initialize a storage element of type t .

15.5.2 Formally

OKAY

$$\frac{type - satisfies(tenv, t, s) \xrightarrow{type} TRUE}{check_can_be_initialized_with(tenv, s, t) \xrightarrow{type} TRUE}$$

ERROR

$$\frac{type - satisfies(tenv, t, s) \xrightarrow{type} FALSE}{check_can_be_initialized_with(tenv, s, t) \xrightarrow{type} TypeError(CannotBeInitializedWith)}$$

Chapter 16

Typing of Statements

The function

$$\text{annotate_stmt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \longrightarrow (\overbrace{\text{stmt}}^{\text{new_s}}, \overbrace{\text{SE}}^{\text{new_env}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a statement `s` in an environment `tenv`, resulting in the annotated statement `new_s` and a modified environment `new_env`. Otherwise, the result is a type error.

One of the following applies:

- `TypingRule.SPass` (see Section 16.1),
- `TypingRule.SAssign` (see Section 16.2),
- `TypingRule.SReturnNone` (see Section 16.3),
- `TypingRule.SReturnOne` (see Section 16.4),
- `TypingRule.SReturnSome` (see Section 16.5),
- `TypingRule.SSeq` (see Section 16.6),
- `TypingRule.SCall` (see Section 16.7),
- `TypingRule.SCond` (see Section 16.8),
- `TypingRule.SCase` (see Section 16.9),
- `TypingRule.SAssert` (see Section 16.10),
- `TypingRule.SWhile` (see Section 16.11),
- `TypingRule.SRepeat` (see Section 16.12),
- `TypingRule.SFor` (see Section 16.13),
- `TypingRule.SThrowNone` (see Section 16.14),

- `TypingRule.SThrowSome` (see Section 16.15),
- `TypingRule.STry` (see Section 16.16).
- `TypingRule.SDeclSome` (see Section 16.17),
- `TypingRule.SDeclNone` (see Section 16.18).

We also define the following helper functions:

- `TypingRule.CaseAlt` (see Section 16.19),
- `TypingRule.SForConstraints` (see Section 16.20)
- `TypingRule.AnnotateLoopLimit` (see Section 16.21)
- `TypingRule.DeclareLocalConstant` (see Section 16.22)
- `TypingRule.AnnotateLocalDelItemUnit` (Section 16.23)

16.1 TypingRule.SPass

16.1.1 Prose

All of the following apply:

- `s` is a pass statement, that is, `S_Pass`;
- `new_s` is `s`;
- `new_env` is `tenv`.

16.1.2 Formally

$$\text{annotate_stmt}(\text{tenv}, \text{S_Pass}) \xrightarrow{\text{type}} (\text{S_Pass}, \text{tenv})$$

16.2 TypingRule.SAssign

16.2.1 Prose

All of the following apply:

- `s` is an assignment `le = re`, that is, `S_Assign(le, re)`;
- One of the following applies:
 - * All of the following apply (SETTER):
 - reducing `(tenv, le, re)` to a setter call via `setter_should_reduce_to_call_s` yields the statement `new_s` (indicating that the assignment corresponds to setter) *// #TE*;

- `new_env` is `tenv`.
- * All of the following apply (`NON_SETTER`):
 - reducing `(tenv, le, re)` to a setter call via
`setter_should_reduce_to_call_s` yields `None` (indicating the assignment
 does not correspond to a setter);
 - annotating the right-hand-side expression `re` in `tenv` yields `(t_re, re1) // #TE`;
 - annotating the left-hand-side expression `le` with the type `t_re` in `tenv`
 yields `le1 // #TE`;
 - `new_s` is the assignment `le1 = re1`, that is, `S_Assign(le1, re1)`;
 - `new_env` is `tenv`.

16.2.2 Formally

SETTER

$$\frac{\text{setter_should_reduce_to_call_s}(\text{tenv}, \text{le}, \text{re}) \xrightarrow{\text{type}} \langle \text{new_s} \rangle \quad // \quad \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Assign}(\text{le}, \text{re})}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

NON_SETTER

$$\frac{\begin{array}{l} \text{setter_should_reduce_to_call_s}(\text{tenv}, \text{le}, \text{re}) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\ \text{annotate_expr}(\text{tenv}, \text{re}) \xrightarrow{\text{type}} (\text{t_re}, \text{re1}) \quad // \quad \#TE \\ \text{annotate_lexpr}(\text{tenv}, \text{le}, \text{t_re}) \xrightarrow{\text{type}} \text{le1} \quad // \quad \#TE \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Assign}(\text{le}, \text{re})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Assign}(\text{le1}, \text{re1})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

16.3 TypingRule.SReturnNone

16.3.1 Prose

All of the following apply:

- `s` is a `return` statement with no expression, that is, `S_Return(None)`;
- the enclosing subprogram does not have a `return` type (it is either a setter or a procedure);
- `new_s` is a `return` statement with no expression, that is, `S_Return(None)`;
- `new_env` is `tenv`.

16.3.2 Formally

$$\frac{L^{\text{tenv}}.\text{return_type} = \text{None}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Return}(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Return}(\text{None})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

16.4 TypingRule.SReturnOne

16.4.1 Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (RETURN_EXPR_NO_RETURN_TYPE):
 - **s** is a **return** statement with some expression;
 - the enclosing subprogram does not have a return type;
 - * All of the following apply (RETURN_TYPE_NO_RETURN_EXPR):
 - **s** is a **return** statement with no expression;
 - the enclosing subprogram has a returned type;
- the result is an error indicating the mismatch between the declared (existence of the) return type and the (existence of the) return expression.

16.4.2 Formally

$$\begin{array}{c}
 \text{RETURN_EXPR_NO_RETURN_TYPE} \\
 L^{\text{tenv}}.\text{return_type} = \text{None} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Return}(\langle _ \rangle)}^s) \xrightarrow{\text{type}} \text{TypeError}(\text{InvalidReturnStmt}) \\
 \\
 \text{RETURN_TYPE_NO_RETURN_EXPR} \\
 L^{\text{tenv}}.\text{return_type} = \langle _ \rangle \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Return}(\text{None})}^s) \xrightarrow{\text{type}} \text{TypeError}(\text{InvalidReturnStmt})
 \end{array}$$

16.5 TypingRule.SReturnSome

16.5.1 Prose

All of the following apply:

- **s** is a **return** statement with an expression **e**, that is, **S_Return**(⟨**e**′⟩);
- the enclosing subprogram has a return type **t**;
- annotating the right-hand-side expression **e** in **tenv** yields (**t_e**′, **e**′) *#TE*;
- checking whether **t_e**′ *type-satisfies* **t** in **tenv** yields **TRUE** *#TE*;
- **new_s** is a **return** statement with value **e**′, that is, **S_Return**(⟨**e**′⟩);
- **new_env** is **tenv**.

16.5.2 Formally

$$\frac{
 \begin{array}{c}
 L^{\text{tenv}}.\text{return_type} = \langle t \rangle \quad \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e') \quad // \text{ \#TE} \\
 \text{checked_typesat}(\text{tenv}, t_e', t) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE}
 \end{array}
 }{
 \text{annotate_stmt}(\text{tenv}, \underbrace{S_Return(\langle e \rangle)}_s) \xrightarrow{\text{type}} (\underbrace{S_Return(\langle e' \rangle)}_{\text{new_s}}, \underbrace{\text{tenv}}_{\text{new_tenv}})
 }$$

16.6 TypingRule.SSeq

16.6.1 Prose

All of the following apply:

- **s** is the AST node for the sequence of statements **s1** and **s2**, that is, **S_Seq(s1, s2)**;
- annotating **s1** in **tenv** yields **(new_s1, tenv1)** // **\#TE**;
- annotating **s2** in **tenv1** yields **(new_s2, new_tenv)** // **\#TE**;
- **new_s** is the AST node for the sequence of statements **new_s1** and **new_s2**, that is, **S_Seq(new_s1, new_s2)**.

16.6.2 Formally

$$\frac{
 \begin{array}{c}
 \text{annotate_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} (\text{new_s1}, \text{tenv1}) \quad // \text{ \#TE} \\
 \text{annotate_stmt}(\text{tenv1}, s2) \xrightarrow{\text{type}} (\text{new_s2}, \text{new_tenv}) \quad // \text{ \#TE}
 \end{array}
 }{
 \text{annotate_stmt}(\text{tenv}, \underbrace{S_Seq(s1, s2)}_s) \xrightarrow{\text{type}} (\underbrace{S_Seq(\text{new_s1}, \text{new_s2})}_{\text{new_s}}, \text{new_tenv})
 }$$

16.7 TypingRule.SCall

16.7.1 Prose

All of the following apply:

- **s** is a call to a subprogram named **name** with arguments **args**;
- annotating the call to **name** with arguments **args**, as a procedure (that is, with **ST_Procedure**), as per Chapter 19 (which makes sure that the call does not have a return type), yields **(new_name, new_args, eqs, None)** // **\#TE**;
- **new_s** is the call to a subprogram named **new_name** with arguments **new_args** and parameter assignments **new_eqs**;
- **new_tenv** is **tenv**.

16.7.2 Formally

$$\frac{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Procedure}) \xrightarrow{\text{type}} (\text{new_name}, \text{new_args}, \text{eqs}, \text{None}) \text{ // } \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Call}(\text{name}, \text{args})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{(\text{S_Call}(\text{new_name}, \text{new_args}, \text{eqs}), \text{tenv})}^{\text{new_s}}}$$

16.7.3 Comments

Notice that the input statement, which belongs to the untyped AST, has two children nodes — **name** and **args**, whereas the output statement, which belongs to the typed AST has the additional node **new_eqs**, which associates expressions with parameters.

16.8 TypingRule.SCond

16.8.1 Prose

All of the following apply:

- **s** is a condition **e** with the statements **s1** and **s2**, that is, **S_Cond(e, s1, s2)**;
- annotating the right-hand-side expression **e** in **tenv** yields **(t_cond, e_cond) // #TE**;
- checking that **t_cond** **type-satisfies** **T_Bool** yields **TRUE // #TE**;
- annotating the statement **s1** in **tenv** yields **s1' // #TE**;
- annotating the statement **s2** in **tenv** yields **s2' // #TE**;
- **new_s** is the condition **e_cond** with the statements **s1'** and **s2'**, that is, **S_Cond(e_cond, s1', s2')**;
- **new_env** is **tenv**.

16.8.2 Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{t_cond}, \text{e_cond}) \text{ // } \#TE \\ \text{checked_typesat}(\text{tenv}, \text{t_cond}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{annotate_block}(\text{tenv}, \text{s1}) \xrightarrow{\text{type}} \text{s1}' \text{ // } \#TE \\ \text{annotate_block}(\text{tenv}, \text{s2}) \xrightarrow{\text{type}} \text{s2}' \text{ // } \#TE \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Cond}(\text{e}, \text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{(\text{S_Cond}(\text{e_cond}, \text{s1}', \text{s2}'))}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

16.9 TypingRule.SCase

16.9.1 Prose

All of the following apply:

- s is a case statement with expression e and case clauses $cases$, that is, $S_Case(e1, cases1)$;
- annotating the right-hand-side expression e in $tenv$ yields $(t_e, e1) // \#TE$;
- annotating each case clause as per Section 16.19 in $cases$ yields the annotated list of clauses $cases1 // \#TE$;
- new_s is a case statement with expression $e1$ and case clauses $cases1$;
- new_env is $tenv1$.

16.9.2 Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(tenv, e) \xrightarrow{\text{type}} (t_e, e1) // \#TE \\ i \in \text{indices}(cases) : \text{annotate_case}(tenv, cases[i]) \xrightarrow{\text{type}} case_i // \#TE \\ cases1 := [i \in \text{indices}(cases) : case_i] \end{array}}{\text{annotate_stmt}(tenv, \underbrace{S_Case(e, cases)}_s) \xrightarrow{\text{type}} (\underbrace{S_Case(e1, cases1)}_{new_s}, \underbrace{tenv}_{new_tenv})}$$

16.10 TypingRule.SAssert

16.10.1 Prose

All of the following apply:

- s is an assert statement with expression e , that is, $S_Assert(e)$;
- annotating the right-hand-side expression e in $tenv$ yields $(t_e', e') // \#TE$;
- checking that t_e' *type-satisfies* T_Bool in $tenv$ yields $TRUE // \#TE$;
- new_s is an assert statement with expression e' , that is, $S_Assert(e')$;
- new_env is $tenv$.

16.10.2 Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(tenv, e) \xrightarrow{\text{type}} (t_e', e') // \#TE \\ \text{checked_typesat}(tenv, t_e', T_Bool) \xrightarrow{\text{type}} TRUE // \#TE \end{array}}{\text{annotate_stmt}(tenv, \underbrace{S_Assert(e)}_s) \xrightarrow{\text{type}} (\underbrace{S_Assert(e')}_{new_s}, \underbrace{tenv}_{new_tenv})}$$

16.11 TypingRule.SWhile

16.11.1 Prose

All of the following apply:

- s is a **while** statement with expression $e1$, optional limit expression $limit1$, and statement block $s1$, that is, $S_While(e1, s1)$;
- annotating the right-hand-side expression $e1$ in $tenv$ yields $(t, e2) // \#TE$;
- annotating the optional limit expression $limit1$ via *annotate_loop_limit* in $tenv$ yields $limit2 // \#TE$;
- checking that t *type-satisfies* T_Bool in $tenv$ yields $TRUE // \#TE$;
- new_s is a **while** statement with expression $e2$, optional limit expression $limit2$, and statement block $s2$, that is, $S_While(e2, s2)$;
- new_env is $tenv$.

16.11.2 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t, e2) // \#TE \\
 \text{annotate_loop_limit}(tenv, limit1) \xrightarrow{\text{type}} limit2 // \#TE \\
 \text{checked_typesat}(tenv, t, T_Bool) \xrightarrow{\text{type}} TRUE // \#TE \\
 \text{annotate_block}(tenv, s1) \xrightarrow{\text{type}} s2 // \#TE
 \end{array}
 }{
 \text{annotate_stmt}(tenv, \overbrace{S_While(e1, limit1, s1)}^s) \xrightarrow{\text{type}} (\overbrace{S_While(e2, limit2, s2)}^{new_s}, \overbrace{tenv}^{new_tenv})
 }$$

16.12 TypingRule.SRepeat

16.12.1 Prose

All of the following apply:

- s is a **repeat** statement with statement block $s1$, optional limit expression $limit1$, and expression $e1$, that is, $S_Repeat(s1, e1, limit1)$;
- annotating $s1$ as a block statement in $tenv$ yields $s2 // \#TE$;
- annotating the optional limit expression $limit1$ via *annotate_loop_limit* in $tenv$ yields $limit2 // \#TE$;
- annotating the right-hand-side expression $e1$ in $tenv$ yields $(t, e2) // \#TE$;
- checking that t *type-satisfies* T_Bool in $tenv$ yields $TRUE // \#TE$;

- `new_s` is a `repeat` statement with statement block `s2`, optional limit expression `limit2`, and condition expression `e2` and , that is, `S.Repeat(s2,e2,limit2)`;
- `new_env` is `tenv`.

16.12.2 Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_block}(\text{tenv}, s1) \xrightarrow{\text{type}} s2 \text{ // \#TE} \\
 \text{annotate_loop_limit}(\text{tenv}, \text{limit1}) \xrightarrow{\text{type}} \text{limit2} \text{ // \#TE} \\
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t, e2) \text{ // \#TE} \\
 \text{checked_typesat}(\text{tenv}, t, \text{T.Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE}
 \end{array}
 }{
 \text{annotate_stmt}(\text{tenv}, \underbrace{\text{S.Repeat}(s1, e1, \text{limit1})}_s) \xrightarrow{\text{type}} (\underbrace{\text{S.Repeat}(s2, e2, \text{limit2})}_{\text{new_s}}, \underbrace{\text{tenv}}_{\text{new_tenv}})
 }$$

16.13 TypingRule.SFor

16.13.1 Prose

All of the following apply:

- `s` is a `for` statement with index `index_name`, start expression `start_e`, direction `dir`, end expression `end_e`, body statement (block) `body`, and optional limit expression `limit`, that is, `S.For`

$$\left\{ \begin{array}{ll}
 \text{index_name} & : \text{index_name} \\
 \text{start_e} & : \text{start_e} \\
 \text{for_direction} & : \text{direction} \\
 \text{end_e} & : \text{end_e} \\
 \text{body} & : \text{body} \\
 \text{limit} & : \text{limit}
 \end{array} \right\};$$
- annotating the right-hand-side expression `start_e` in `tenv` yields `(start_t, start_e')//\#TE`;
- annotating the right-hand-side expression `end_e` in `tenv` yields `(end_t, end_e')//\#TE`;
- annotating the optional loop limit expression `limit` via `annotate_loop_limit` in `tenv` yields `limit'//\#TE`;
- obtaining the `underlying type` of `start_t` in `tenv` yields `start_struct//\#TE`;
- obtaining the `underlying type` of `end_t` in `tenv` yields `end_struct//\#TE`;
- applying `for_constraints` to `start_struct`, `end_struct`, `start_e'`, `end_e'`, and `dir` in `tenv`, to obtain the constraints on the loop index `index_name`, yields `cs//\#TE`;
- `ty` is the integer type with constraints `cs`;
- checking that `index_name` is not already declared in `tenv` yields `TRUE//\#TE`;

- adding `index_name` as a local immutable variable with type `ty` to `tenv` yields `tenv'`;
- annotating `body` as a block statement in `tenv'` yields `body' // #TE`;
- `new_s` is the `for` statement with index `index_name`, start expression `start_e'`, direction `dir`, end expression `end_e'`, body statement (block) `body'`, and optional limit expression `limit`;
- `new_tenv` is `tenv` (notice that this means `index_name` is only declared for annotating `body'` but then goes out of scope).

16.13.2 Formally

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{start_e}) \xrightarrow{\text{type}} (\text{start_t}, \text{start_e}') \quad // \text{ \#TE} \\
\text{annotate_expr}(\text{tenv}, \text{end_e}) \xrightarrow{\text{type}} (\text{end_t}, \text{end_e}') \quad // \text{ \#TE} \\
\text{annotate_loop_limit}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} \text{limit}' \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{start_t}) \xrightarrow{\text{type}} \text{start_struct} \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{end_t}) \xrightarrow{\text{type}} \text{end_struct} \quad // \text{ \#TE} \\
\text{for_constraints}(\text{tenv}, \text{start_struct}, \text{end_struct}, \text{start_e}', \text{end_e}', \text{dir}) \xrightarrow{\text{type}} \text{cs} \quad // \text{ \#TE} \\
\text{ty} := \text{T_Int}(\text{cs}) \quad \text{check_var_not_in_env}(\text{tenv}, \text{index_name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\text{add_local}(\text{tenv}, \text{ty}, \text{index_name}, \text{LDK_Let}) \xrightarrow{\text{type}} \text{tenv}' \\
\text{annotate_block}(\text{tenv}', \text{body}) \xrightarrow{\text{type}} \text{body}' \quad // \text{ \#TE} \\
\hline
\text{annotate_stmt} \left(\text{tenv}, \text{S_For} \left\{ \begin{array}{l} \overbrace{\text{index_name} : \text{index_name}}^{\text{s}} \\ \text{start_e} : \text{start_e} \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e} \\ \text{body} : \text{body} \\ \text{limit} : \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \\
\left(\text{S_For} \left\{ \begin{array}{l} \overbrace{\text{index_name} : \text{index_name}}^{\text{new_s}} \\ \text{start_e} : \text{start_e}' \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e}' \\ \text{body} : \text{body}' \\ \text{limit} : \text{limit}' \end{array} \right\}, \underbrace{\text{new_tenv}}_{\text{tenv}} \right)
\end{array}$$

16.14 TypingRule.SThrowNone

16.14.1 Prose

All of the following apply:

- `s` is a throw statement with no expression, that is, `S_Throw(None)`;
- `new_s` is `s`;
- `new_env` is `tenv`.

16.14.2 Formally

$$\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(\text{None})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})$$

16.15 TypingRule.SThrowSome

16.15.1 Prose

All of the following apply:

- `s` is a throw statement with expression `e`, that is, `S_Throw((e))`;
- annotating the right-hand-side expression `e` in `tenv` yields $(\text{t_e}, e') \# \text{\#TE}$;
- checking that `t_e` has the structure of an exception type yields $\text{TRUE} \# \text{\#TE}$;
- `new_s` is a throw statement with expression `e'` and type `t_e`, that is, `S_Throw(((e', t_e)))`;
- `new_env` is `tenv`.

16.15.2 Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\text{t_e}, e') \# \text{\#TE} \\ \text{check_structure}(\text{tenv}, \text{t_e}, \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \# \text{\#TE} \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}((e))}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(((e', \text{t_e})))}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

16.16 TypingRule.STry

16.16.1 Prose

All of the following apply:

- s is a try statement with statement s' , list of catchers catchers and an optional **otherwise** block;
- annotating the statement s' as a block statement yields $s' \text{ // } \#TE$;
- annotating each catcher $\text{catchers}[i]$, for each i in $\text{indices}(\text{catchers})$ in tenv yields $c_i \text{ // } \#TE$;
- $\text{catchers}'$ is the list of annotated catchers c_i for each $i \in \text{indices}(\text{catchers})$;
- One of the following applies:
 - * All of the following apply (NO_OTHERWISE):
 - there is no **otherwise** statement;
 - new_s is a try statement with statement s' , list catchers $\text{catchers}'$ and no **otherwise** statement, that is $S_Try(s', \text{catchers}', \text{None})$;
 - * All of the following apply (OTHERWISE):
 - there is an **otherwise** statement otherwise ;
 - annotating the statement otherwise as a block statement in tenv yields $\text{otherwise}' \text{ // } \#TE$;
 - new_s is a try statement with statement s' , list catchers $\text{catchers}'$ and otherwise statement $\text{otherwise}'$, that is $S_Try(s', \text{catchers}', \langle \text{otherwise}' \rangle)$;
- new_env is tenv .

16.16.2 Formally

NO_OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} s' \text{ // } \#TE \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} c_i \text{ // } \#TE \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{***** common prefix *****} \\
 \text{new_s} := S_Try(s', \text{catchers}', \text{None}) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{S_Try(s', \text{catchers}', \text{None})}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})
 \end{array}$$

OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} s' \text{ // } \#TE \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} c_i \text{ // } \#TE \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{***** common prefix *****} \\
 \text{annotate_block}(\text{tenv}, \text{otherwise}) \xrightarrow{\text{type}} \text{otherwise}' \text{ // } \#TE \\
 \text{new_s} := S_Try(s', \text{catchers}', \text{otherwise}') \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{S_Try(s', \text{catchers}', \langle \text{otherwise}' \rangle)}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})
 \end{array}$$

16.17 TypingRule.SDeclSome

16.17.1 Prose

All of the following apply:

- s is a declaration with local declaration keyword ldk , local identifiers ldi , and an expression e , that is, $S_Decl(ldk, ldi, \langle e \rangle)$;
- annotating the right-hand-side expression e in $tenv$ yields $(t_e, e') // \#TE$;
- One of the following applies:
 - * All of the following apply (CONSTANT):
 - ldk indicates a local constant declaration, that is, $LDK_Constant$;
 - symbolically simplifying e in $tenv$ yields the literal $v // \#TE$;
 - declaring a local constant of type t_e , literal v and identifier ldi in $tenv$ yields (new_tenv, ldi') ;
 - new_s is a declaration with ldk , ldi' and an expression e' .
 - * All of the following apply (NON-CONSTANT):
 - ldk indicates that this is not a local constant declaration, that is, $ldk \neq LDK_Constant$;
 - declaring the local identifiers ldi of type t_e with local declaration keyword ldk in $tenv$ yields (new_tenv, ldi') ;
 - new_s is a declaration with ldk , ldi' and an expression e' .

16.17.2 Formally

CONSTANT

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e) \xrightarrow{\text{type}} (t_e, e') // \#TE \\
 ldk = LDK_Constant \quad \text{reduce_constants}(tenv, e) \xrightarrow{\text{type}} v // \#TE \\
 \text{declare_local_constant}(tenv, v, ldi) \xrightarrow{\text{type}} new_tenv \\
 new_s := S_Decl(LDK_Constant, ldi', \langle e' \rangle) \\
 \hline
 \text{annotate_stmt}(tenv, \overbrace{S_Decl(ldk, ldi, \langle e \rangle)}^s) \xrightarrow{\text{type}} (new_s, new_tenv)
 \end{array}$$

NON-CONSTANT

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e) \xrightarrow{\text{type}} (t_e, e') // \#TE \\
 ldk \neq LDK_Constant \\
 \text{annotate_local_decl_item}(tenv, t_e, ldk, ldi) \xrightarrow{\text{type}} (new_tenv, ldi') \\
 new_s := S_Decl(ldk, ldi', \langle e' \rangle) \\
 \hline
 \text{annotate_stmt}(tenv, \overbrace{S_Decl(ldk, ldi, \langle e \rangle)}^s) \xrightarrow{\text{type}} (new_s, new_tenv)
 \end{array}$$

16.18 TypingRule.SDeclNone

16.18.1 Prose

All of the following apply:

- s is a local declaration statement with a variable keyword and local identifiers ldi , and no initial expression, that is, $S_Decl(LDK_Var, ldi, \text{None})$ (local declarations of `let` variables and constants require an initializing expression, otherwise they are rejected by an ASL parser);
- annotating the uninitialised local declarations ldi in $tenv$ yields (new_tenv, ldi') ;
- new_s is a local declaration statement with variable keyword, local identifiers ldi' , and no initial expression, that is, $S_Decl(LDK_Var, ldi', \text{None})$.

16.18.2 Formally

$$\frac{\begin{array}{c} \text{annotate_local_decl_item_uninit}(tenv, ldi) \xrightarrow{\text{type}} (new_tenv, ldi') \quad // \quad \#TE \\ new_s := S_Decl(LDK_Var, ldi', \text{None}) \end{array}}{\text{annotate_stmt}(tenv, \overbrace{S_Decl(LDK_Var, ldi', \text{None})}^s) \xrightarrow{\text{type}} (new_s, new_tenv)}$$

16.19 TypingRule.CaseAlt

The helper function

$$\text{annotate_case}(\overbrace{SE}^{tenv}, \overbrace{case_alt}^{case}, \overbrace{ty}^{t_e}) \longrightarrow \overbrace{case_alt \cup TTypeError}^{case1} \quad \#TE$$

annotates the case clause `case` for an expression of type t_e in $tenv$, resulting in the annotated case clause `case1`. Otherwise, the result is a type error.

16.19.1 Prose

All of the following apply:

- `case` is a case clause with pattern $p0$, `optional where` expression $w0$, and `otherwise` statement $s0$, that is, $\{pattern : p0, where : w0, stmt : s0\}$;
- annotating the pattern $p0$ with type t_e in $tenv$ yields $p1 \quad // \quad \#TE$;
- annotating the statement $s0$ as a block statement in $tenv$ yields $s1 \quad // \quad \#TE$;
- One of the following applies:
 - * All of the following apply (`NO_WHERE_STMT`):
 - $w0$ is `None` (that is, no `where` expression);

- `case1` is $\{\text{pattern} : p1, \text{where} : \text{None}, \text{stmt} : s1\}$.
- * All of the following apply (`WHERE_STMT`):
 - `w0` is the singleton expression for `e_w0`, that is, $\langle e_w0 \rangle$;
 - annotating the expression `e_w0` in `tenv` yields $(twe, e_w1) \#TE$;
 - checking whether the structure of `twe` in `tenv` is that of the `boolean` type yields $TRUE \#TE$;
 - `case1` is $\{\text{pattern} : p1, \text{where} : \langle e_w1 \rangle, \text{stmt} : s1\}$.

16.19.2 Formally

NO_WHERE_STMT

$$\begin{array}{c}
 \text{case} = \{\text{pattern} : p0, \text{where} : \text{None}, \text{stmt} : s0\} \\
 \text{annotate_pattern}(\text{tenv}, t_e, p0) \xrightarrow{\text{type}} p1 \quad \#TE \\
 \text{annotate_block}(\text{tenv}, s0) \xrightarrow{\text{type}} s1 \quad \#TE \\
 \hline
 \text{annotate_case}(\text{tenv}, \text{case}, t_e) \xrightarrow{\text{type}} \overbrace{\{\text{pattern} : p1, \text{where} : \text{None}, \text{stmt} : s1\}}^{\text{case1}}
 \end{array}$$

WHERE_STMT

$$\begin{array}{c}
 \text{case} = \{\text{pattern} : p0, \text{where} : \langle e_w0 \rangle, \text{stmt} : s0\} \\
 \text{annotate_pattern}(\text{tenv}, t_e, p0) \xrightarrow{\text{type}} p1 \quad \#TE \\
 \text{annotate_block}(\text{tenv}, s0) \xrightarrow{\text{type}} s1 \quad \#TE \\
 \text{annotate_expr}(\text{tenv}, e_w0) \xrightarrow{\text{type}} (twe, e_w1) \quad \#TE \\
 \text{check_structure}(\text{tenv}, twe, T_Bool) \xrightarrow{\text{type}} TRUE \quad \#TE \\
 \hline
 \text{annotate_case}(\text{tenv}, \text{case}, t_e) \xrightarrow{\text{type}} \overbrace{\{\text{pattern} : p1, \text{where} : \langle e_w1 \rangle, \text{stmt} : s1\}}^{\text{case1}}
 \end{array}$$

16.20 TypingRule.SForConstraints

The function

$$\text{for_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{struct1}}, \overbrace{\text{ty}}^{\text{struct2}}, \overbrace{\text{expr}}^{e1'}, \overbrace{\text{expr}}^{e2'}, \overbrace{\text{dir}}^{\text{dir}}) \longrightarrow \overbrace{\text{int.constraints}}^{\text{vis}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

infers the integer constraints for a `for` loop index variable from the following:

- the [well-constrained version](#) of the type of the start expression — `struct1`
- the [well-constrained version](#) of the type of the end expression — `struct2`
- the annotated start expression — `e1'`
- the annotated end expression — `e2'`
- the loop direction — `dir`

The result is `vis`. Otherwise, the result is a type error.

16.20.1 Prose

One of the following applies:

- All of the following apply (NOT_INTEGERS):
 - * at least one of `struct1` and `struct2` is not an integer type;
 - * the result is a type error indicating that the start expression and end expression of `for` loops must have the `structure` of integer types.
- All of the following apply (UNCONSTRAINED):
 - * both of `struct1` and `struct2` are integer types;
 - * at least one of `struct1` and `struct2` is the unconstrained integer type;
 - * define `vis` as `Unconstrained`.
- All of the following apply (WELL_CONSTRAINED):
 - * both of `struct1` and `struct2` are integer types;
 - * neither `struct1` nor `struct2` is the unconstrained integer type;
 - * symbolically simplifying `e1'` in `tenv` yields `e1_n` *//* `#TE`;
 - * symbolically simplifying `e2'` in `tenv` yields `e2_n` *//* `#TE`;
 - * define `ics_up` as the single range constraint with expressions `e1_n` and `e2_n`;
 - * define `ics_down` as the single range constraint with expressions `e2_n` and `e1_n`;
 - * define `vis` as `ics_up` if `dir` is `Up` and `ics_down` otherwise.

16.20.2 Formally

NOT_INTEGERS

$$\frac{\text{ast_label}(\text{struct1}) \neq \text{T_Int} \vee \text{ast_label}(\text{struct2}) \neq \text{T_Int}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_LBI})}$$

UNCONSTRAINED

$$\frac{\begin{array}{l} \text{ast_label}(\text{struct1}) = \text{T_Int} \wedge \text{ast_label}(\text{struct2}) = \text{T_Int} \\ \text{struct1} = \text{unconstrained_integer} \vee \text{struct2} = \text{unconstrained_integer} \end{array}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \overbrace{\text{Unconstrained}}^{\text{vis}}}$$

WELL_CONSTRAINED

$$\frac{\begin{array}{l} \text{ast_label}(\text{struct1}) = \text{T_Int} \wedge \text{ast_label}(\text{struct2}) = \text{T_Int} \\ \text{struct1} \neq \text{unconstrained_integer} \wedge \text{struct2} \neq \text{unconstrained_integer} \\ \text{normalize}(\text{tenv}, \text{e1}') \xrightarrow{\text{type}} \text{e1_n} \text{ // } \text{\#TE} \\ \text{normalize}(\text{tenv}, \text{e2}') \xrightarrow{\text{type}} \text{e2_n} \text{ // } \text{\#TE} \\ \text{ics_up} := \text{WellConstrained}([\text{Constraint_Range}(\text{e1_n}, \text{e2_n})]) \\ \text{ics_down} := \text{WellConstrained}([\text{Constraint_Range}(\text{e2_n}, \text{e1_n})]) \\ \text{vis} := \text{choice}(\text{dir} = \text{Up}, \text{ics_up}, \text{ics_down}) \end{array}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \text{vis}}$$

16.21 TypingRule.AnnotateLoopLimit

The function

$$\text{annotate_loop_limit}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{expr} \rangle}^{\text{e}}) \longrightarrow \overbrace{\text{expr} \cup \text{TTypeError}}^{\text{e'} \quad \#TE}$$

annotates an optional expression e serving as the limit of a loop in tenv , yielding the optional loop expression e' . Otherwise, the result is a type error.

16.21.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * e is `None`;
 - * e' is `None`.
- All of the following apply (SOME):
 - * e is $\langle \text{limit} \rangle$;
 - * annotating `limit` in tenv yields $(\text{t}, \text{limit}') \#TE$;
 - * checking that t is a constrained integer in tenv via `check_constrained_integer` yields `TRUE` $\#TE$;
 - * e' is $\langle \text{limit}' \rangle$.

16.21.2 Formally

$$\begin{array}{c} \text{NONE} \\ \text{annotate_loop_limit}(\overbrace{\text{tenv}}^{\text{tenv}}, \overbrace{\text{None}}^{\text{limit}}) \xrightarrow{\text{type}} \overbrace{\text{None}}^{\text{limit}' \quad \#TE} \\ \\ \text{SOME} \\ \frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{t}, \text{limit}') \quad \#TE \\ \text{check_constrained_integer}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \end{array}}{\text{annotate_loop_limit}(\overbrace{\text{tenv}}^{\text{tenv}}, \overbrace{\langle \text{limit} \rangle}^{\text{limit}}) \xrightarrow{\text{type}} \overbrace{\langle \text{limit}' \rangle}^{\text{limit}' \quad \#TE}} \end{array}$$

16.22 TypingRule.DeclareLocalConstant

The function

$$\text{declare_local_constant}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{literal}}^{\text{v}}, \overbrace{\text{local_decl_item}}^{\text{ldi}}) \xrightarrow{\text{type}} \overbrace{\text{SE}}^{\text{new_tenv}}$$

adds the literal v with the local declaration item ldi as a constant to the local component of the static environment tenv , yielding the modified static environment new_tenv .

16.22.1 Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * `ldi` corresponds to a discarding declaration, that is, `LDI.Discard`;
 - * `new_tenv` is `tenv`.
- All of the following apply (VAR):
 - * `ldi` corresponds to a variable declaration for `x`, that is, `LDI.Var(x)`;
 - * define `new_tenv` and the environment `tenv` with its global component updated by binding `x` to `v` in its `constant_values` map.
- All of the following apply (TUPLE):
 - * `ldi` corresponds to a tuple declaration, that is, `LDI.Var(_)`;
 - * this case is not yet implemented.
- All of the following apply (TYPED):
 - * `ldi` corresponds to a typed declaration of the local declaration item `ldi'` and some type, that is, `LDI.Typed(ldi', _)`;
 - * applying `declare_local_constant` to `v` and `ldi'` in `tenv` yields `new_tenv`.

16.22.2 Formally

DISCARD

$$\text{declare_local_constant}(\text{tenv}, v, \overbrace{\text{LDI.Discard}}^{\text{ldi}}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new_tenv}}$$

VAR

$$\frac{\text{new_tenv} := (G^{\text{tenv}}, L^{\text{tenv}}.\text{constant_values}[x \mapsto v])}{\text{declare_local_constant}(\text{tenv}, v, \overbrace{\text{LDI.Var}(x)}^{\text{ldi}}) \xrightarrow{\text{type}} \text{new_tenv}}$$

TUPLE

$$\text{declare_local_constant}(\text{tenv}, v, \overbrace{\text{LDI.Tuple}(_)}^{\text{ldi}}) \xrightarrow{\text{type}} \text{not implemented yet}$$

TYPED

$$\frac{\text{declare_local_constant}(\text{tenv}, v, \text{ldi}') \xrightarrow{\text{type}} \text{new_tenv}}{\text{declare_local_constant}(\text{tenv}, v, \overbrace{\text{LDI.Typed}(\text{ldi}', _)}^{\text{ldi}}) \xrightarrow{\text{type}} \text{new_tenv}}$$

16.23 TypingRule.AnnotateLocalDeclItemUninit

The function

$$\text{annotate_local_decl_item_uninit}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{local_decl_item}}^{\text{ldi}}) \xrightarrow{\text{type}} (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{local_decl_item}}^{\text{new_ldi}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates the local declaration for a variable declaration without an initializing expressions in the static environment `tenv`, yielding a pair consisting of the annotated local declaration item `new_ldi` and the modified static environment `new_tenv`. Otherwise, the result is a type error.

16.23.1 Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * `ldi` corresponds to a discarding declaration, that is, `LDI_Discard`;
 - * `new_tenv` is `tenv` and `new_ldi` is `ldi`.
- All of the following apply (TYPED):
 - * `ldi` corresponds to a variable declaration via the local declaration item `ldi'` and type annotation `t`, that is, `LDI_Typed(ldi', t)`;
 - * annotating `t` in `tenv` yields `t' // \#TE`;
 - * annotating the local declaration item `ldi'` with the type `t'` and local declaration keyword `LDI_Var` yields `(new_tenv, new_ldi') // \#TE`;
 - * define `new_ldi` as the typed local declaration item with local declaration item `new_ldi'` and type `t'`.

16.23.2 Formally

DISCARD

$$\text{annotate_local_decl_item_uninit}(\text{tenv}, \overbrace{\text{LDI_Discard}}^{\text{ldi}}) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ldi}}^{\text{new_ldi}})$$

TYPED

$$\frac{\begin{array}{l} \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t' \text{ // } \#TE \\ \text{annotate_local_decl_item}(\text{tenv}, t', \text{LDK_Var}, \text{ldi}') \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi}') \text{ // } \#TE \\ \text{new_ldi} := \text{LDI_Typed}(\text{new_ldi}', t') \end{array}}{\text{annotate_local_decl_item_uninit}(\text{tenv}, \overbrace{\text{LDI_Typed}(\text{ldi}', t)}^{\text{ldi}}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi})}$$

ERROR

$$\frac{\textcolor{blue}{ast_label}(\text{ldi}) \in \{\text{LDI_Var}, \text{LDI_Tuple}\}}{\textcolor{blue}{annotate_local_decl_item_uninit}(\text{tenv}, \text{ldi}) \xrightarrow{\text{type}} \textcolor{blue}{TypeError}(\text{ExpectedTypedDeclaration})}$$

Chapter 17

Typing of Blocks

The function

$$\text{annotate_block}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_stmt}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a block statement `s` in static environment `tenv` and returns the annotated statement `new_stmt` or a type error, if one is detected.

17.1 TypingRule.Block

17.1.1 Prose

All of the following apply:

- annotating the statement `s` in `tenv` yields `(new_stmt, new_tenv)`^{`\#TE`};
- the modified environment `new_tenv` is dropped.

17.1.2 Example

```
func main () => integer
begin
  if TRUE then
    let i = 3;
    print (DecStr (i));
  end
  let i = "Some text";
  print (i);
  return 0;
end
```

17.1.3 Formally

$$\frac{\text{annotate_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new_stmt}, _) \text{ // } \#TE}{\text{annotate_block}(\text{tenv}, s) \xrightarrow{\text{type}} \text{new_stmt}}$$

17.1.4 Comments

A local identifier declared in a block statement (with `var`, `let`, or `constant`) is in scope from the point immediately after its declaration until the end of the immediately enclosing block. This means, we can discard the environment at the end of an enclosing block, which has the effect of dropping bindings of the identifiers declared inside the block.

Chapter 18

Typing of Catchers

The function

$$\text{annotate_catcher}(\overbrace{\text{SE}}^{\text{tenv}}, (\overbrace{(\text{identifier})}^{\text{name_opt}} \times \overbrace{\text{ty}}^{\text{ty}} \times \overbrace{\text{stmt}}^{\text{stmt}})) \longrightarrow \\ (\overbrace{(\text{identifier})}^{\text{name_opt}} \times \overbrace{\text{ty}'}^{\text{ty}'} \times \overbrace{\text{new_stmt}}^{\text{new_stmt}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a catcher given by the [optional](#) name of the matched exception — `name_opt` — the exception type — `ty` — and the statement to execute upon catching the exception — `stmt`. The result is the catcher with the same [optional](#) name — `name_opt`, an annotated type `ty'`, and annotated statement `new_stmt`. Otherwise, the result is a type error.

One of the following applies:

- `TypingRule.CatcherNone` (see [Section 18.1](#)),
- `TypingRule.CatcherSome` (see [Section 18.2](#)).

18.1 TypingRule.CatcherNone

18.1.1 Prose

All of the following apply:

- the catcher has no named identifier, that is, `(None, ty, stmt)`;
- annotating the type `ty` in `tenv` yields `ty'//\#TE`;
- determining whether `ty'` has the [structure](#) of an exception type yields `TRUE//\#TE`;
- annotating the block `stmt` in `tenv` yields `new_stmt`.

18.1.2 Formally

$$\begin{array}{c}
\text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \\
\text{check_structure}(\text{tenv}, \text{ty}', \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{annotate_block}(\text{tenv}, \text{stmt}) \xrightarrow{\text{type}} \text{new_stmt} \text{ // } \#TE \\
\hline
\text{annotate_catcher}(\text{tenv}, (\underbrace{\text{None}}_{\text{name_opt}}, \text{ty}, \text{stmt})) \xrightarrow{\text{type}} (\underbrace{\text{None}}_{\text{name_opt}}, \text{ty}', \text{new_stmt})
\end{array}$$

18.2 TypingRule.CatcherSome

18.2.1 Prose

All of the following apply:

- the catcher has a named identifier, that is, $(\langle \text{name} \rangle, \text{ty}, \text{stmt})$;
- annotating the type ty in tenv yields $\text{ty}' \text{ // } \#TE$;
- determining whether ty' has the `structure` of an exception type yields $\text{TRUE} \text{ // } \#TE$;
- the identifier `name` is not bound in tenv ;
- binding `name` in the local environment of tenv with the type ty' as an immutable variable (that is, with the local declaration keyword `LDK.Let`), yields the static environment tenv' ;
- annotating the block `stmt` in tenv' yields `new_stmt`.

18.2.2 Formally

$$\begin{array}{c}
\text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \\
\text{check_structure}(\text{tenv}, \text{ty}', \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{add_local}(\text{tenv}, \text{name}, \text{ty}', \text{LDK.Let}) \xrightarrow{\text{type}} \text{tenv}' \\
\text{annotate_block}(\text{tenv}', \text{stmt}) \xrightarrow{\text{type}} \text{new_stmt} \text{ // } \#TE \\
\hline
\text{annotate_catcher}(\text{tenv}, (\underbrace{\langle \text{name} \rangle}_{\text{name_opt}}, \text{ty}, \text{stmt})) \xrightarrow{\text{type}} (\underbrace{\langle \text{name} \rangle}_{\text{name_opt}}, \text{ty}', \text{new_stmt})
\end{array}$$

Chapter 19

Typing of Subprogram Calls

The function

$$\text{annotate_call}(\overbrace{\text{SIE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{expr}^*}^{\text{args}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}) \longrightarrow \overbrace{(\text{identifier}, \text{expr}^*)}^{\text{name1}, \text{args1}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\langle \text{ty} \rangle}^{\text{ret_ty_opt}})$$

annotates the call to subprogram **name** with arguments **args** and call type **call_type**, resulting in the following:

- **name1** — a string, which uniquely identifies **name** among the set of overloading subprograms declared with **name**;
- **args1** — the annotated argument expressions;
- **eqs** — the expressions providing values to the parameters;
- **ret_ty_opt** — the [optional](#) annotated return type.

Otherwise, the result is a type error.

The function is defined by the rule `TypingRule.AnnotateCall` (see Section [19.1](#)).

We also define helper functions via respective rules:

- `TypingRule.AnnotateCallArgTyped` (see Section [19.2](#))
- `TypingRule.CheckCalleeParams` (see Section [19.3](#))
- `TypingRule.RenameTyEqs` (see Section [19.4](#))
- `TypingRule.SubstExprNormalize` (see Section [19.5](#))
- `TypingRule.SubstExpr` (see Section [19.6](#))
- `TypingRule.SubstConstraint` (see Section [19.7](#))

- `TypingRule.CheckArgsTypeSat` (see Section 19.8)
- `TypingRule.AnnotateParameterDefining` (see Section 19.9)
- `TypingRule.AnnotateRetTy` (Section 19.10)
- `TypingRule.SubprogramForName` (see Section 19.11)
- `TypingRule.DeduceEqs` (see Section 19.12)
- `TypingRule.FilterCallCandidates` (see Section 19.13)
- `TypingRule.HasArgClash` (see Section 19.14)

19.1 `TypingRule.AnnotateCall`

19.1.1 Prose

All of the following apply:

- applying `annotate_exprs` to annotate the expression list `args` in `tenv` yields `caller_arg_typed` *#TE*;
- applying `annotate_call_arg_typed` to `name`, `caller_arg_typed`, `call_type` in `tenv` yields `(name1, args1, eqs, ret_ty)` *#TE*.

19.1.2 Formally

$$\frac{\begin{array}{l} \text{annotate_exprs}(\text{tenv}, \text{args}) \xrightarrow{\text{type}} \text{caller_arg_typed} \text{ } // \text{ } \#TE \\ \text{annotate_call_arg_typed}(\text{tenv}, \text{name}, \text{caller_arg_typed}, \text{call_type}) \xrightarrow{\text{type}} \\ \quad (\text{name1}, \text{args1}, \text{eqs}, \text{ret_ty}) \text{ } // \text{ } \#TE \end{array}}{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{call_type}) \xrightarrow{\text{type}} (\text{name1}, \text{args1}, \text{eqs}, \text{ret_ty})}$$

19.2 `TypingRule.AnnotateCallArgTyped`

The function

$$\text{annotate_call_arg_typed}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{(\text{ty} \times \text{expr})^*}^{\text{caller_args_typed}}, \overbrace{(\text{sub_program_type}, \text{ret_ty_opt})}^{\text{call_type}}) \longrightarrow \underbrace{(\text{identifier}, \text{expr}^*)}_{\text{name1, args1}}, \underbrace{(\text{identifier} \times \text{expr})^*}_{\text{eqs}}, \underbrace{\langle \text{ty} \rangle}_{\text{ret_ty_opt}} \cup \underbrace{\text{TTypeError}}_{\#TE}$$

is similar to `annotate_call`, except that the argument expressions are replaced by the annotated expressions. That is, pairs consisting of a type and an expression. Otherwise, the result is a type error.

19.2.1 Prose

All of the following apply:

- applying `unzip` to `caller_arg_typed` yields the corresponding lists of types and expressions `caller_arg_types` and `args1`;
- applying `subprogram_for_name` to match `name` and `caller_arg_types` in `tenv` yields `(eqs1, name1, callee) // #TE`;
- checking that `sub_program_type` of `callee` equals `call_type` yields `TRUE // #TE`;
- checking that the lengths of `callee.args` and `args1` are the same yields `TRUE // #TE`;
- applying `annotate_parameter_defining` to `callee.args`, `caller_args_typed`, and `callee_params` in `tenv` to annotate the implicit parameters yields `eqs3' // #TE`;
- define `eqs3` is the concatenation of `eqs3'` and `eqs1`;
- applying `check_args_typesat` to `callee_arg_types` to check that the actual arguments have correct types with respect to `caller_arg_types` in `tenv` yields `TRUE // #TE`;
- applying `check_callee_params` to `callee_params` to check they have correct types with respect to `eqs3` in `tenv` yields `TRUE // #TE`;
- applying `annotate_ret_ty` to `call_type` and `callee.return_type` to check that the two call types match and to substitute actual parameter arguments in the formal return type yields `ret_ty_opt // #TE`;
- define `eqs` as `eqs3`.

19.2.2 Formally

$$\begin{array}{c}
\text{unzip}(\text{caller_args_typed}) = (\text{caller_arg_types}, \text{args1}) \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \\
\quad (\text{eqs1}, \text{name1}, \text{callee}) \text{ // \#TE} \\
\text{check}(\text{callee.sub_program_type} = \text{call_type}, \text{TE_MRV}) \longrightarrow \text{TRUE // \#TE} \\
\text{equal_length}(\text{callee.args}, \text{args1}) \xrightarrow{\text{type}} \text{arity_match} \\
\text{check}(\text{arity_match}, \text{TE_CBA}) \longrightarrow \text{TRUE // \#TE} \\
\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{callee.args}, \\ \text{caller_args_typed}, \\ \text{callee.parameters} \end{array} \right) \xrightarrow{\text{type}} \text{eqs3}' \text{ // \#TE} \\
\text{eqs3} := \text{eqs3}' + \text{eqs1} \\
\text{check_args_typesat}(\text{tenv}, \text{callee.args}, \text{caller_arg_types}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE // \#TE} \\
\text{check_callee_params}(\text{tenv}, \text{callee.parameters}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE // \#TE} \\
\text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{callee.return.type}) \xrightarrow{\text{type}} \text{ret_ty_opt // \#TE} \\
\hline
\text{annotate_call_arg_typed}(\text{tenv}, \text{name}, \text{caller_args_typed}, \text{call_type}) \xrightarrow{\text{type}} \\
\quad (\text{name1}, \text{args1}, \overbrace{\text{eqs3}}^{\text{eqs}}, \text{ret_ty_opt})
\end{array}$$

19.3 TypingRule.CheckCalleeParams

The function

$$\text{check_callee_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{callee_params}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks that the parameters in `callee_params` are correct with respect to the parameter expressions `eqs3`. Otherwise, the result is a type error.

19.3.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `callee_params` is an empty list;
 - * the result is **TRUE**.
- All of the following apply:
 - * `callee_params` is a non-empty list with **head** `callee_param` and **tail** `callee_params1`;
 - * One of the following applies:
 - All of the following apply (NO_TYPE):

- ▷ `callee_param` does not have a type annotation, that is, `(_, None)`.
- All of the following apply (PARAMETERIZED):
 - ▷ `callee_param` is a parameter `s` with a type annotation of a `parameterized integer type` for the same parameter, that is, `(s, (T_Int(Parameterized(s))))`.
- All of the following apply (OTHER):
 - ▷ `callee_param` is a parameter `s` whose type annotation is `callee_param_t`, that is, `(s, (callee_param_t))`;
 - ▷ `callee_param_t` is not the `parameterized integer type` for the same parameter;
 - ▷ substituting the parameter expressions from `eqs3` in `callee_param_t` yields `callee_param_t_renamed``//#TE`;
 - ▷ applying `assoc_opt` to `eqs3` and `s` yields the expression `caller_param_e` (that is, the parameter `s` is associated with the expression `caller_param_e`);
 - ▷ annotating the expression `caller_param_e` in `tenv` yields `(caller_param_t, _)``//#TE`;
 - ▷ checking that `caller_param_t` `type-satisfies` `callee_param_t_renamed` in `tenv` yields `TRUE``//#TE`;
- * applying `check_callee_params` to `callee_params1` and `eqs3` in `tenv` yields `TRUE``//#TE`.

19.3.2 Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{check_callee_params}(\text{tenv}, \overbrace{[]}^{\text{callee_params}}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{NO_TYPE} \\
\text{callee_params} = [(_, \text{None})] + \text{callee_params1} \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{PARAMETERIZED} \\
\text{callee_params} = [(s, \langle T_Int(\text{Parameterized}(s)) \rangle)] + \text{callee_params1} \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{OTHER} \\
\text{callee_params} = [(s, \langle \text{callee_param_t} \rangle)] + \text{callee_params1} \\
\text{callee_param_t} \neq T_Int(\text{Parameterized}(s)) \\
\text{rename_ty_eqs}(\text{tenv}, \text{eqs3}, \text{callee_param_t}) \xrightarrow{\text{type}} \text{callee_param_t.renamed} \text{ // } \#TE \\
\text{assoc_opt}(\text{eqs3}, s) \xrightarrow{\text{type}} \langle \text{caller_param_e} \rangle \\
\text{annotate_expr}(\text{tenv}, \text{caller_param_e}) \xrightarrow{\text{type}} \text{callee_param_t} \text{ // } \#TE \\
\text{checked_typesat}(\text{tenv}, \text{callee_param_t}, \text{callee_param_t.renamed}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

19.4 TypingRule.RenameTyEqs

The function

$$\text{rename_ty_eqs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{new_ty}} \cup \overbrace{T_TypeError}^{\#TE}$$

transforms the type `ty` in the static environment `tenv`, by substituting parameter names with their corresponding expressions in `eqs`, yielding the type `new_ty`. Otherwise, the result is a type error.

19.4.1 Prose

One of the following applies:

- All of the following apply (`T_BITS`):
 - * `ty` is a bitvector type with width expression `e` and fields `fields`, that is, `T_Bits(e, fields)`;

- * applying *subst.expr_normalize* to `eqs` and `e` in `tenv` yields the expression `new_e`;
 - * define `new_ty` as a bitvector type with with expression `new_e` and fields `fields`.
- All of the following apply (`T_INT_WELLCONSTRAINED`):
 - * `ty` is a well-constrained integer type with constraints `constraints`;
 - * applying *subst.constraint* to each constraint `constraints[i]`, for `i` in *indices*(`constraints`), yields the constraint `new_ci`;
 - * define `new_constraints` as the list of constraints `new_ci`, for `i` in *indices*(`constraints`);
 - * define `new_ty` as the well-constrained integer type with constraints `new_constraints`.
 - All of the following apply (`T_INT_PARAMETERIZED`):
 - * `ty` is a *parameterized integer type* for the parameter `name`;
 - * applying *subst.expr_normalize* to `eqs` and the expression `E_Var(name)` yields `e`;
 - * define `new_ty` as the well-constrained integer type with the single constraint for `e`, that is, `T_Int(WellConstrained(Constraint.Exact(e)))`.
 - All of the following apply (`T_TUPLE`):
 - * `ty` is the tuple type over the list of tuples `tys`, that is, `T_Tuple(tys)`;
 - * applying *rename_ty_eqs* to `eqs` and the type `tys[i]`, for each `i` in *indices*(`tys`), yields the type `new_tyi`;
 - * define `new_tys` as the list of types `new_tyi`, for each `i` in *indices*(`tys`);
 - * define `new_ty` as the tuple type over `new_tys`, that is, `T_Tuple(new_tys)`.
 - All of the following apply (`OTHER`):
 - * `ty` is not one of the types in the previous cases, that is, `ty` is not a bitvector type, nor an integer type, nor a tuple type;
 - * `new_ty` is `ty`.

19.4.2 Formally

$$\begin{array}{c}
\text{T_BITS} \\
\hline
\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Bits}(e, \text{fields})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(\text{new_e}, \text{fields})}^{\text{new_ty}} \\
\\
\text{T_INT_WELLCONSTRAINED} \\
\text{i} \in \text{indices}(\text{constraints}) : \text{subst_constraint}(\text{tenv}, \text{constraints}[\text{i}]) \xrightarrow{\text{type}} \text{new_c}_i \\
\text{new_constraints} := [\text{i} \in \text{indices}(\text{constraints}) : \text{new_c}_i] \\
\text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{new_constraints})) \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty} \\
\\
\text{T_INT_PARAMETERIZED} \\
\text{subst_expr_normalize}(\text{eqs}, \text{E_Var}(\text{name})) \xrightarrow{\text{type}} e \\
\text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(e))) \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{name}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty} \\
\\
\text{T_TUPLE} \\
\text{i} \in \text{indices}(\text{tys}) : \text{rename_ty_eqs}(\text{eqs}, \text{tys}[\text{i}]) \xrightarrow{\text{type}} \text{new_ty}_i \\
\text{new_tys} := [\text{i} \in \text{indices}(\text{tys}) : \text{new_ty}_i] \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Tuple}(\text{tys})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{new_tys})}^{\text{new_ty}} \\
\\
\text{OTHER} \\
\text{ast_label}(\text{ty}) \notin \{\text{T_Bits}, \text{T_Int}, \text{T_Tuple}\} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}
\end{array}$$

19.5 TypingRule.SubstExprNormalize

The function

$$\text{subst_expr_normalize}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{new_e}}^{\text{expr}}$$

transforms the expression e in the static environment tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the expression new_e . Otherwise, the result is a type error.

19.5.1 Prose

All of the following apply:

- transforming e in the static environment tenv , by substituting the parameter expressions eqs , yields $e1$;
- symbolically simplifying $e1$ in tenv yields new_e .

19.5.2 Formally

$$\frac{\text{subst_expr}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}$$

19.6 TypingRule.SubstExpr

The function

$$\text{subst_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{substs}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{\text{new_e}}$$

transforms the expression e in the static environment tenv , by substituting parameter names with their corresponding expressions in substs , yielding the expression new_e . Otherwise, the result is a type error.

19.6.1 Prose

One of the following applies:

- All of the following apply (E_VAR_IN_SUBSTS):
 - * e is a variable expression for the identifier s , that is, $\text{E_Var}(s)$;
 - * applying *assoc_opt* to s and substs yields the expression new_e . That is, s is a parameter with an associated expression;
- All of the following apply ($\text{E_VAR_NOT_IN_SUBSTS}$):
 - * e is the variable expression for the identifier s , that is, $\text{E_Var}(s)$;
 - * applying *assoc_opt* to s and substs yields **None**. That is, s is not a parameter with an associated expression;
 - * define new_e is e .
- All of the following apply (E_UNOP):
 - * e is the unary operator expression for the operator op and expression e , that is, $\text{E_Unop}(\text{op}, e1)$;
 - * applying *subst_expr* to substs and $e1$ in tenv yields $e1'$;
 - * define new_e as the unary operator expression for the operator op and expression $e1'$, that is, $\text{E_Unop}(\text{op}, e1')$.
- All of the following apply (E_BINOP):

- * e is the binary operator expression for the operator op and expressions $e1$ and $e2$, that is, $E_Binop(op, e1, e2)$;
 - * applying *subst_expr* to $subst_s$ and $e1$ in $tenv$ yields $e1'$;
 - * applying *subst_expr* to $subst_s$ and $e2$ in $tenv$ yields $e2'$;
 - * define new_e as the unary operator expression for the operator op and expression $e1'$, that is, $E_Unop(op, e1')$.
- All of the following apply (E_COND):
 - * e is the conditional expression for expressions $e1$, $e2$, and $e3$, that is, $E_Cond(e1, e2, e3)$;
 - * applying *subst_expr* to $subst_s$ and $e1$ in $tenv$ yields $e1'$;
 - * applying *subst_expr* to $subst_s$ and $e2$ in $tenv$ yields $e2'$;
 - * applying *subst_expr* to $subst_s$ and $e3$ in $tenv$ yields $e3'$;
 - * define new_e as the conditional expression for expressions $e1'$, $e2'$, and $e3'$, that is, $E_Cond(e1', e2', e3')$.
 - All of the following apply (E_CONCAT):
 - * e is the concatenation of expressions e_s , that is, $E_Concat(e_s)$;
 - * applying *subst_expr* to $subst_s$ and every expression $e_s[i]$, for i in $indices(e_s)$ yields $new_e_s_i$;
 - * define es' as the list of expressions $new_e_s_i$, for i in $indices(e_s)$;
 - * define new_e as the concatenation of expressions es' , that is, $E_Concat(es')$.
 - All of the following apply (E_CALL):
 - * e is the call expression for subprogram x with arguments $args$ and parameter expressions $param_args$, that is, $E_Call(x, args, param_args)$;
 - * applying *subst_expr* to $subst_s$ and every argument expression $args[i]$, for i in $indices(args)$ yields e_i ;
 - * define $args'$ as e_i for each i in $indices(args)$;
 - * define new_e as the call expression for subprogram x with arguments $args'$ and parameter expressions $param_args$, that is, $E_Call(x, args', param_args)$.
 - All of the following apply ($E_GETARRAY$):
 - * e is the *array access* expression for base expression $e1$ and index expression $e2$, that is, $E_GetArray(e1, e2)$;
 - * applying *subst_expr* to $subst_s$ and $e1$ in $tenv$ yields $e1'$;
 - * applying *subst_expr* to $subst_s$ and $e2$ in $tenv$ yields $e2'$;
 - * define new_e as the *array access* expression for base expression $e1'$ and index expression $e2'$, that is, $E_GetArray(e1', e2')$.

- All of the following apply (E_GETFIELD):
 - * **e** is the field access expression for base expression **e** and field **x**, that is, `E_GetField(e1, x)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the field access expression for base expression **e** and field **x**, that is, `E_GetField(e1', x)`.
- All of the following apply (E_GETFIELDS):
 - * **e** is the access to fields **fields** with base expression **e1**, that is, `E_GetFields(e1, fields)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the access to fields **fields** with base expression **e1'**, that is, `E_GetFields(e1', fields)`.
- All of the following apply (E_GETITEM):
 - * **e** is the access to tuple item **i** of the tuple expression **e1**, that is, `E_GetItem(e1, i)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the access to tuple item **i** of the tuple expression **e1'**, that is, `E_GetItem(e1', i)`.
- All of the following apply (E_PATTERN):
 - * **e** is the pattern expression of expression **e1** and patterns **ps**, that is, `E_Pattern(e1, ps)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the pattern expression of expression **e1'** and patterns **ps**, that is, `E_Pattern(e1', ps)`.
- All of the following apply (E_RECORD):
 - * **e** is the record expression of record type **t** and list of fields **fields**;
 - * for every pair (**x**, **e1**) in **fields**, applying *subst.expr* to **substs** **e1** in **tenv** yields **e1'_x**;
 - * define **fields'** as the list of pairs (**x**, **e1'_x**) for every pair (**x**, **e1**) in **fields**;
 - * define **new_e** as the record expression of record type **t** and list of fields **fields'**.
- All of the following apply (E_SLICE):
 - * **e** is the slicing expression for sub-expression **e1** and list of slices **slices**, that is, `E_Slice(e1, slices)`;
 - * applying *subst.expr* to **e1** in **tenv** yields **e1'**;

- * define **new_e** as slicing expression for sub-expression **e1'** and list of slices **slices**, that is, **E.Slice(e1', slices)**.
- All of the following apply (**E_TUPLE**):
 - * **e** is the tuple expression of expressions **e_s**, that is, **E.Tuple(e_s)**;
 - * applying *subst_expr* to **substs** and every expression **e_s[i]** in **tenv**, for every **i** in **indices(e_s)** yields **new_e_i**;
 - * define **es'** as the list of expressions **new_e_i**, for every **i** in **indices(e_s)**;
 - * define **new_e** as the tuple expression of expressions **es'**, that is, **E.Tuple(es')**.
- All of the following apply (**E_ATC**):
 - * **e** is the type assertion of expression **e1** and type **t**, that is, **E.ATC(e1, t)**;
 - * applying *subst_expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the type assertion of expression **e1'** and type **t**, that is, **E.ATC(e1', t)**.
- All of the following apply (**OTHER**):
 - * **e** is either a literal expression or an unknown value expression;
 - * define **new_e** as **e**.

19.6.2 Formally

$$\begin{array}{c}
 \text{E_VAR_IN_SUBSTS} \\
 \frac{\text{assoc_opt}(\text{s}, \text{substs}) \xrightarrow{\text{type}} \langle \text{new_e} \rangle}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Var}(\text{s})}^{\text{e}}) \xrightarrow{\text{type}} \text{new_e}} \\
 \\
 \text{E_VAR_NOT_IN_SUBSTS} \\
 \frac{\text{assoc_opt}(\text{s}, \text{substs}) \xrightarrow{\text{type}} \text{None}}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Var}(\text{s})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{e}}^{\text{new_e}}} \\
 \\
 \text{E_UNOP} \\
 \frac{\text{subst_expr}(\text{tenv}, \text{substs}, \text{e1}) \xrightarrow{\text{type}} \text{e1}'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Unop}(\text{op}, \text{e1})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{E_Unop}(\text{op}, \text{e1}')}^{\text{new_e}}} \\
 \\
 \text{E_BINOP} \\
 \frac{\text{subst_expr}(\text{tenv}, \text{substs}, \text{e1}) \xrightarrow{\text{type}} \text{e1}' \quad \text{subst_expr}(\text{tenv}, \text{substs}, \text{e2}') \xrightarrow{\text{type}} \text{e2}'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Binop}(\text{op}, \text{e1}, \text{e2})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{E_Binop}(\text{op}, \text{e1}', \text{e2}')}^{\text{new_e}}}
 \end{array}$$

E_COND

$$\frac{\begin{array}{c} \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \\ \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e3') \xrightarrow{\text{type}} e3' \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Cond(e1, e2, e3)}^e) \xrightarrow{\text{type}} \overbrace{E_Cond(e1', e2', e3')}^{\text{new_e}}}$$

E_CONCAT

$$\frac{\begin{array}{c} i \in \text{indices}(e_s) : \text{subst_expr}(\text{tenv}, \text{subst}, e_s[i]) \xrightarrow{\text{type}} \text{new_e_s}_i \\ es' := [i \in \text{indices}(e_s) : \text{new_e_s}_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Concat(e_s)}^e) \xrightarrow{\text{type}} \overbrace{E_Concat(es')}^{\text{new_e}}}$$

E_CALL

$$\frac{\begin{array}{c} i \in \text{indices}(\text{args}) : \text{subst_expr}(\text{tenv}, \text{subst}, \text{args}[i]) \xrightarrow{\text{type}} e_i \\ \text{args}' := [i \in \text{indices}(\text{args}) : e_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Call(x, \text{args}, \text{param_args})}^e) \xrightarrow{\text{type}} \overbrace{E_Call(x, \text{args}', \text{param_args})}^{\text{new_e}}}$$

E_GETARRAY

$$\frac{\begin{array}{c} \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2' \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetArray(e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{E_GetArray(e1', e2')}^{\text{new_e}}}$$

E_GETFIELD

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetField(e1, x)}^e) \xrightarrow{\text{type}} \overbrace{E_GetField(e1', x)}^{\text{new_e}}}$$

E_GETFIELDS

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetFields(e1, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{E_GetFields(e1', \text{fields})}^{\text{new_e}}}$$

E_GETITEM

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetItem(e1, i)}^e) \xrightarrow{\text{type}} \overbrace{E_GetItem(e1', i)}^{\text{new_e}}}$$

$$\begin{array}{c}
\text{E_PATTERN} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Pattern(e1, ps)}^e) \xrightarrow{\text{type}} \overbrace{E_Pattern(e1', ps)}^{\text{new_e}}} \\
\\
\text{E_RECORD} \\
\frac{\begin{array}{c} (x, e1) \in \text{fields} : \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1_x \\ \text{fields}' := [(x, e1) \in \text{fields} : (x, e1_x)] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Record(t, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{E_Record(t, \text{fields}')}^{\text{new_e}}} \\
\\
\text{E_SLICE} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Slice(e1, \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{E_Slice(e1', \text{slices})}^{\text{new_e}}} \\
\\
\text{E_TUPLE} \\
\frac{\begin{array}{c} i \in \text{indices}(e_s) : \text{subst_expr}(\text{tenv}, \text{subst}, e_s[i]) \xrightarrow{\text{type}} \text{new_e}_i \\ es' := [i \in \text{indices}(e_s) : \text{new_e}_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Tuple(e_s)}^e) \xrightarrow{\text{type}} \overbrace{E_Tuple(es')}^{\text{new_e}}} \\
\\
\text{E_ATC} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_ATC(e1, t)}^e) \xrightarrow{\text{type}} \overbrace{E_ATC(e1', t)}^{\text{new_e}}} \\
\\
\text{OTHER} \\
\frac{\text{ast_label}(e) \in \{E_Literal, E_Unknown\}}{\text{subst_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{e}^{\text{new_e}}}
\end{array}$$

19.7 TypingRule.SubstConstraint

The function

$$\text{subst_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, (\overbrace{\text{identifier} \times \text{expr}}^{\text{eqs}})^*, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{new_c}}^{\text{int_constraint}}$$

transforms the integer constraint c in the static environment tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the integer constraint new_c . Otherwise, the result is a type error.

19.7.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an exact constraint for the expression e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *subst_expr_normalize* in tenv to eqs and e yields new_e ;
 - * define new_c as the exact constraint for the expression new_e , that is, $\text{Constraint_Exact}(\text{new_e})$.
- All of the following apply (RANGE):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *subst_expr_normalize* in tenv to eqs and $e1$ yields $e1'$;
 - * applying *subst_expr_normalize* in tenv to eqs and $e2$ yields $e2'$;
 - * define new_c as the range constraint for the expressions $e1'$ and $e2'$, that is, $\text{Constraint_Range}(e1', e2')$.

19.7.2 Formally

EXACT

$$\frac{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(\text{new_e})}^{\text{new_c}}}$$

RANGE

$$\frac{\begin{array}{l} \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e1) \xrightarrow{\text{type}} e1' \\ \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e2) \xrightarrow{\text{type}} e2' \end{array}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}}$$

19.8 TypingRule.CheckArgsTypeSat

The function

$$\text{check_args_typesat}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{callee_args}}, \overbrace{\text{ty}^*}^{\text{caller_arg_types}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}} \longrightarrow \underbrace{\{\text{TRUE}\} \cup \text{TTypeError}}_{\text{\#TE}})$$

checks that the types caller_arg_types *type-satisfy* the types of the corresponding formal arguments callee_args with the parameters substituted with their corresponding arguments as per eqs3 and results in a type error otherwise.

19.8.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `callee_args` and `caller_arg.types` are empty;
 - * the result is `TRUE`.
- All of the following apply (NON_EMPTY):
 - * view `callee_args` as a list with `head` (`callee_arg_name`, `callee_arg`) and `tail` `callee_args_one`;
 - * view `caller_arg.types` as a list with `head` `caller_arg` and `tail` `caller_arg.types1`;
 - * applying `rename_ty_eqs` to `eqs3` and `callee_arg` in `tenv` to substitute parameter arguments in `callee_arg` yields `callee_arg1` `// #TE`;
 - * checking that `caller_arg` `type-satisfies` `callee_arg1` in `tenv` yields `TRUE` `// #TE`;
 - * applying `check_args_typesat` to `callee_args_one`, `caller_arg.types1`, and `eqs3` in `tenv` yields `TRUE` `// #TE`;
 - * the result is `TRUE`.

19.8.2 Formally

We note that it is guaranteed by `TypingRule.AnnotateCallArgTyped` that `args` and `caller_args.typed` have the same length.

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{check_args_typesat}(\text{tenv}, \overbrace{[]^{\text{callee_args}}}, \overbrace{[]^{\text{caller_arg.types}}}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{callee_args} \stackrel{\text{is}}{=} [(\text{callee_arg_name}, \text{callee_arg})] + \text{callee_args_one} \\
 \text{caller_arg_types} \stackrel{\text{is}}{=} [\text{caller_arg}] + \text{caller_arg_types1} \\
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs3}, \text{callee_arg}) \xrightarrow{\text{type}} \text{callee_arg1} \text{ // } \#TE \\
 \text{checked_typesat}(\text{tenv}, \text{caller_arg}, \text{callee_arg1}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check_args_typesat}(\text{tenv}, \text{callee_args_one}, \text{caller_arg_types1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
 \hline
 \text{check_args_typesat}(\text{tenv}, \text{callee_args}, \text{caller_arg_types}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}
 \end{array}$$

19.9 TypingRule.AnnotateParameterDefining

The function

$$\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv} \\ \text{SE} , \\ \text{eqs1} \\ \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{args}} , \\ \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{caller_args_typed}} , \\ \overbrace{(\text{ty} \times \text{expr})^*}^{\text{callee_params}} , \\ \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*} \end{array} \right) \longrightarrow \begin{array}{c} \text{eqs} \\ \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}} \cup \\ \text{\#TE} \\ \text{TTypeError} \end{array}$$

checks that all parameter-defining arguments in `callee_params` are **statically evaluable** constrained integers. The result — `eqs` — is the list of parameter identifiers and their corresponding expressions, added to `eqs1`. Otherwise, the result is a type error.

19.9.1 Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * both `args` and `caller_args_typed` are empty;
 - * define `eqs` as `eqs1`.
- All of the following apply:
 - * view `args` as a list with `head` (`callee_x`, `_`) and `tail` `args1`;
 - * view `caller_args_typed` as a list with `head` (`caller_ty`, `caller_e`) and `tail` `caller_args_typed1`;
 - * define `callee_arg_is_param` as `TRUE` if and only if `callee_x` is listed as a parameter in `callee_params`;
 - * One of the following applies:
 - All of the following apply (`ARG_IS_PARAM`):
 - ▷ `callee_arg_is_param` is `TRUE`;
 - ▷ checking that `caller_e` is **statically evaluable** in `tenv` yields `TRUE//#TE`;
 - ▷ checking that `caller_ty` is a constrained integer in `tenv` yields `TRUE//#TE`;
 - ▷ applying `annotate_parameter_defining` to `args1`, `caller_args_typed1`, `eqs1` in `tenv` yields `eqs2//#TE`;
 - ▷ define `eqs` as a list with `head` (`callee_x`, `caller_e`) and `tail` `eqs2`.

– All of the following apply (`ARG_IS_NOT_PARAM`):

- ▷ `callee_arg_is_param` is `FALSE`;
- ▷ applying *annotate_parameter_defining* to `args1`, `caller_args_typed1`, `eqs1` in `tenv` yields `eqs` *#TE*.

19.9.2 Formally

We note that it is guaranteed by `TypingRule.AnnotateCallArgTyped` that `args` and `caller_args_typed` have the same length.

EMPTY

$$\text{annotate_parameter_defining}(\text{tenv}, \text{eqs1}, \underbrace{\text{args}}_{[]}, \underbrace{\text{caller_args_typed}}_{[]}, \text{callee_params}) \xrightarrow[\underbrace{\text{eqs}}_{\text{eqs1}}]{\text{type}}$$

ARG_IS_PARAM

$$\frac{\begin{array}{l} \text{args} \stackrel{\text{is}}{=} [(\text{callee_x}, _)] + \text{args1} \\ \text{caller_args_typed} \stackrel{\text{is}}{=} [(\text{caller_ty}, \text{caller_e})] + \text{caller_args_typed1} \\ \left(\begin{array}{l} \text{callee_arg_is_param} := \\ \exists i \in \text{indices}(\text{callee_params}). \text{callee_params}[i] = (\text{callee_x}, _) \end{array} \right) \\ \text{***** common prefix *****} \\ \text{callee_arg_is_param} = \text{TRUE} \\ \text{check_statically_evaluable}(\text{tenv}, \text{caller_e}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{check_constrained_integer}(\text{tenv}, \text{caller_ty}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{annotate_parameter_defining}(\text{tenv}, \text{args1}, \text{caller_args_typed1}, \text{eqs1}) \xrightarrow[\text{eqs2 // } \#TE]{\text{type}} \\ \text{eqs} := [(\text{callee_x}, \text{caller_e})] + \text{eqs2} \end{array}}{\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{eqs1}, \\ \text{args}, \\ \text{caller_args_typed}, \\ \text{callee_params} \end{array} \right) \xrightarrow{\text{type}} \text{eqs} \text{ // } \#TE}$$

$$\begin{array}{c}
\text{ARG_IS_NOT_PARAM} \\
\text{args} \stackrel{\text{is}}{=} [(\text{callee_x}, _)] + \text{args1} \\
\text{caller_args_typed} \stackrel{\text{is}}{=} [(\text{caller_ty}, \text{caller_e})] + \text{caller_args_typed1} \\
\left(\begin{array}{l} \text{callee_arg_is_param} := \\ \exists i \in \text{indices}(\text{callee_params}). \text{callee_params}[i] = (\text{callee_x}, _) \end{array} \right) \\
\text{***** common prefix *****} \\
\text{callee_arg_is_param} = \text{FALSE} \\
\text{annotate_parameter_defining}(\text{tenv}, \text{args1}, \text{caller_args_typed1}, \text{eqs1}) \xrightarrow{\text{type}} \text{eqs} \\
\hline
\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{eqs1}, \\ \text{args}, \\ \text{caller_args_typed}, \\ \text{callee_params} \end{array} \right) \xrightarrow{\text{type}} \text{eqs} \quad // \quad \#TE
\end{array}$$

19.10 TypingRule.AnnotateRetTy

The function

$$\text{annotate_ret_ty}(\underbrace{\text{SE}}_{\text{tenv}}, \underbrace{\text{sub_program_type}}_{\text{call_type}}, \underbrace{\langle \text{ty} \rangle}_{\text{callee_ret_ty_opt}}, \underbrace{(\text{identifier} \times \text{expr})^*}_{\text{eqs3}} \xrightarrow{\text{ret_ty_opt}} \underbrace{\langle \text{ty} \rangle \cup \text{TTypeError}}_{\#TE}$$

annotates the **optional** return type `callee_ret_ty_opt` given with the subprogram type `call_type` with respect to the parameter expressions `eqs3`, yielding the **optional** annotated type `ret_ty_opt`. Otherwise, the result is a type error.

19.10.1 Prose

One of the following applies:

- All of the following apply (FUNCTION_OR_GETTER):
 - * `call_type` is one of `ST_Function`, `ST_Getter`, or `ST_EmptyGetter`;
 - * `callee_ret_ty_opt` is `<ty>`;
 - * applying `rename_ty_eqs` to `eqs3` and `ty` yields `ty1` *//* `#TE`;
 - * `ret_ty_opt` is `<ty1>`.
- All of the following apply (PROCEDURE_OR_SETTER):
 - * `call_type` is one of `ST_Procedure`, `ST_Setter`, or `ST_EmptySetter`;
 - * `callee_ret_ty_opt` is `None`;
 - * define `ret_ty_opt` as `None`.

- All of the following apply (RET_TYPE_MISMATCH):
 - * the condition that `call_type` is one of `ST_Procedure`, `ST_Setter`, or `ST_EmptySetter` if and only if `callee_ret_ty_opt` is `None` does not hold;
 - * the result is a type error indicating the mismatch.

19.10.2 Formally

$$\begin{array}{c}
 \text{FUNCTION_OR_GETTER} \\
 \text{call_type} \in \{\text{ST_Function}, \text{ST_Getter}, \text{ST_EmptyGetter}\} \\
 \hline
 \text{rename_ty_eqs}(\text{eqs3}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \quad // \quad \#TE \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{callee_ret_ty_opt}}_{\langle \text{ty} \rangle}, \text{eqs3}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\langle \text{ty1} \rangle} \\
 \\
 \text{PROCEDURE_OR_SETTER} \\
 \text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}, \text{ST_EmptySetter}\} \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{callee_ret_ty_opt}}_{\text{None}}, \text{eqs3}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\text{None}} \\
 \\
 \text{RET_TYPE_MISMATCH} \\
 \neg \left(\begin{array}{c} \text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}, \text{ST_EmptySetter}\} \leftrightarrow \\ \text{callee_ret_ty_opt} = \text{None} \end{array} \right) \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{callee_ret_ty_opt}, \text{eqs3}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_MRV})
 \end{array}$$

19.11 TypingRule.SubprogramForName

The function

$$\text{subprogram_for_name}(\underbrace{\text{tenv}}_{\text{SE}}, \underbrace{\text{name}}_{\text{S}}, \underbrace{\text{caller_arg_types}}_{\text{ty}^*}) \longrightarrow \underbrace{\left((\text{identifier} \times \text{expr})^*, \underbrace{\text{extra_nargs}}_{\text{S}}, \underbrace{\text{callee}}_{\text{func}} \right)}_{\substack{\#TE \\ \cup \quad \text{TTypeError}}}$$

looks up the static environment `tenv` for a subprogram associated with `name` and the list of argument types `caller_arg_types` and determines which one of the following cases holds:

- there is no declared subprogram that matches `name` and `caller_arg_types`;
- there is exactly one subprogram that matches `name` and `caller_arg_types`;
- there is more than one subprogram that matches `name` and `caller_arg_types`;

The first and last cases result in a type error. If the second case holds, the function returns a tuple comprised of:

- **extra_nargs** — the list of extra named arguments (parameters);
- **name'** — the string that uniquely identifies this subprogram;
- **callee** — the AST node defining the called subprogram.

Otherwise, the result is a type error.

19.11.1 Prose

One of the following applies:

- All of the following apply (UNDEFINED):
 - * **tenv** does not contain a binding for **name** in the **subprogram_renamings** map ($G^{\text{tenv}}.\text{subprogram_renamings}$);
 - * the result is a type error indicating that the identifier has not been declared (as a subprogram).
- All of the following apply (NO_CANDIDATES):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 19.13) yields an empty set //\#TE ;
 - * the result is a type error indicating that the call given by **name** and **caller_arg_types** does not match any defined subprogram.
- All of the following apply (TOO_MANY_CANDIDATES):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 19.13) yields **matching_renamings** //\#TE ;
 - * **matching_renamings** contains at least two elements;
 - * the result is a type error indicating that the call given by **name** and **caller_arg_types** matches more than one defined subprogram.
- All of the following apply (ONE_CANDIDATE):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 19.13) yields **matching_renamings** //\#TE ;
 - * **matching_renamings** contains a single element — (**matched_name**, **func_sig**);
 - * deducing the argument values for the parameters via **deduce_eqs** with **caller_arg_types**, **func_sig.args** in **tenv** yields (**extra_nargs**, **name'**, **callee**) //\#TE .

19.11.2 Formally

$$\begin{array}{c}
\text{UNDEFINED} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \perp \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI}) \\
\\
\text{NO_CANDIDATES} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \emptyset \text{ // \#TE} \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_NCC}) \\
\\
\text{TOO_MANY_CANDIDATES} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \\
\text{matching_renamings} \text{ // \#TE} \\
|\text{matching_renamings}| \geq 2 \\
\hline
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_TMC}) \\
\\
\text{ONE_CANDIDATE} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \\
\text{matching_renamings} \text{ // \#TE} \\
\text{matching_renamings} = [(\text{matched_name}, \text{func_sig})] \\
\text{deduce_eqs}(\text{tenv}, \text{caller_arg_types}, \text{func_sig.args}) \xrightarrow{\text{type}} \\
(\text{extra_nargs}, \text{name}', \text{callee}) \text{ // \#TE} \\
\hline
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \\
(\text{extra_nargs}, \text{name}', \text{callee})
\end{array}$$

19.12 TypingRule.DeduceEqs

The function

$$\text{deduce_eqs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{caller_arg_types}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}} \longrightarrow \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}} \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

takes the types of the actual arguments of a call — `caller_arg_types`, the list of formal arguments — `args` — which consist of the names of a subprogram arguments and their associated types, and infers the expressions associated with parameters that correspond to bitvector widths, yielding the result in `eqs`. Otherwise, the result is a type error.

It is guaranteed that by `TypingRule.HasArgClash`, which is used by `TypingRule.FilterCallCandidates` before calling `deduce_eqs`, that `caller_arg_types` and `args` have the same length.

19.12.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `caller_arg_types` and `args` are empty lists;
 - * define `eqs` as the empty list.
- All of the following apply (NOT_BITS_PARAMETER):
 - * `caller_arg_types` has `head` `caller` and `tail` `caller_arg_types1`;
 - * `args` has `head` `(_, callee)` and `tail` `args1`;
 - * `caller` is not a bitvector type with a width expression that is a variable expression;
 - * applying `deduce_eqs` to `caller_arg_types1` and `args1` in `tenv` yields `eqs`.
- All of the following apply (BITS_PARAMETER):
 - * `caller_arg_types` has `head` `caller` and `tail` `caller_arg_types1`;
 - * `args` has `head` `(_, callee)` and `tail` `args1`;
 - * `caller` is bitvector type whose width expression is the variable expression for `x`;
 - * obtaining the `structure` of `caller` in `tenv` yields the bitvector type with width expression `e_caller` `// #TE`;
 - * applying `deduce_eqs` to `caller_arg_types1` and `args1` in `tenv` yields `eqs1`;
 - * define `eqs` as the list with `head` `(x, e_caller)`.

19.12.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{deduce_eqs}(\text{tenv}, \overbrace{[]^{\text{caller_arg_types}}}, \overbrace{[]^{\text{args}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{eqs}}}
 \end{array}$$

$$\begin{array}{c}
 \text{NOT_BITS_PARAMETER} \\
 \frac{\text{caller} \neq \text{T_Bits}(\text{E_Var}(_)) \quad \text{deduce_eqs}(\text{tenv}, \text{caller_arg_types1}, \text{args1}) \xrightarrow{\text{type}} \text{eqs}}{\text{deduce_eqs}(\text{tenv}, \overbrace{[\text{caller}] + \text{caller_arg_types1}}^{\text{caller_arg_types}}, \overbrace{[(_, \text{callee})] + \text{args1}}^{\text{args}}) \xrightarrow{\text{type}} \text{eqs}}
 \end{array}$$

$$\begin{array}{c}
 \text{BITS_PARAMETER} \\
 \begin{array}{l}
 \text{caller} = \text{T_Bits}(\text{E_Var}(\text{x})) \\
 \text{get_structure}(\text{tenv}, \text{caller}) \xrightarrow{\text{type}} \text{T_Bits}(\text{e_caller}, _) \text{ // \#TE} \\
 \text{deduce_eqs}(\text{tenv}, \text{caller_arg_types1}, \text{args1}) \xrightarrow{\text{type}} \text{eqs1} \\
 \text{eqs} := [(\text{x}, \text{e_caller})] + \text{eqs1}
 \end{array} \\
 \hline
 \text{deduce_eqs}(\text{tenv}, \overbrace{[\text{caller}] + \text{caller_arg_types1}}^{\text{caller_arg_types}}, \overbrace{[(_, \text{callee})] + \text{args1}}^{\text{args}}) \xrightarrow{\text{type}} \text{eqs}
 \end{array}$$

19.13 TypingRule.FilterCallCandidates

The helper function

$$\text{filter_call_candidates}(\overbrace{\mathcal{S}\mathcal{E}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{formal_types}}, \overbrace{\mathcal{P}(\mathcal{S})}^{\text{candidates}}) \longrightarrow \overbrace{(\mathcal{S} \times \text{func})^*}^{\text{matches}}$$

iterates over the list of unique subprogram names in `candidates` and checks whether their lists of arguments clash with the types in `formal_types` in `tenv`. The result is the set of pairs consisting of the names and function definitions of the subprograms whose arguments clash in `candidates`. Otherwise, the result is a type error.

The names `candidates` are assumed to exist in $G^{\text{tenv}}.\text{subprograms}$.

19.13.1 Prose

One of the following applies:

- All of the following apply (NO_CANDIDATES):
 - * `candidates` is empty;
 - * `matches` is empty.
- All of the following apply (CANDIDATES_EXIST):
 - * `candidates` is a list with `head` name and `tail` candidates1;
 - * the function definition associated with `name` in `tenv` is `func_def`;
 - * determining whether there is an argument clash between `formal_types` and the arguments in `func_def` (that is, `func_def.args`) yields `b // #TE`;
 - * filtering the call candidates in `candidates1` with `formal_types` in `tenv` yields `matches1 // #TE`;
 - * if `b` is `TRUE` then `matches` is the list with `head` (name, func_def) and `tail` matches1, and otherwise it is matches1.

19.13.2 Formally

$$\begin{array}{c}
 \text{NO_CANDIDATES} \\
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[]^{\text{candidates}}}^{\text{candidates}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{matches}}}^{\text{matches}} \\
 \\
 \text{CANDIDATES_EXIST} \\
 \begin{array}{l}
 \text{func_def} := G^{\text{tenv}}.\text{subprograms}(\text{name}) \\
 \text{has_arg_clash}(\text{tenv}, \text{formal_types}, \text{func_def.args}) \xrightarrow{\text{type}} \text{b} \text{ // } \#TE \\
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \text{candidates1}) \xrightarrow{\text{type}} \text{matches1} \text{ // } \#TE \\
 \text{matches} := \text{choice}(\text{b}, [(\text{name}, \text{func_def})] + \text{matches1}, \text{matches1})
 \end{array} \\
 \hline
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[\text{name}] + \text{candidates1}}^{\text{candidates}}) \xrightarrow{\text{type}} \text{matches}
 \end{array}$$

19.14 TypingRule.HasArgClash

The function

$$\text{has_arg_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{f_tys}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

checks whether a list of types `f_tys` clashes with the list of types appearing in the list of arguments `args` in `tenv`, yielding the result in `b`. Otherwise, the result is a type error.

19.14.1 Prose

All of the following apply:

- equating the list lengths of `f_tys` and `args` either yields `TRUE` or `FALSE`, which short-circuits the entire rule;
- `a_tys` is the list of types appearing in `args`, in the same order;
- for each `i` in the list of indices of `f_tys`, applying `type_clashes` to `f_tys[i]` and `a_tys[i]` in `tenv` yields `TRUE`/`FALSE`, `\#TE`;
- `b` is `TRUE` (unless the rule short-circuited with `FALSE` or a type error).

19.14.2 Formally

$$\frac{\text{equal_length}(\text{formal_types}, \text{args}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \quad \text{a_tys} := [(_, t) \in \text{args} : t] \quad \text{i} \in \text{indices}(\text{f_tys}) : \text{type_clashes}(\text{tenv}, \text{f_tys}[\text{i}], \text{a_tys}[\text{i}]) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE}, \text{\#TE}}{\text{has_arg_clash}(\text{tenv}, \text{f_tys}, \text{args}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}}$$

Chapter 20

Typing of Subprograms

The function

$$\text{annotate_subprogram}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\mathbf{f}}) \longrightarrow \overbrace{\text{func} \cup \mathbf{TTypeError}}^{\mathbf{f}' \quad \#TE}$$

annotates a subprogram \mathbf{f} in an environment tenv , resulting in an annotated subprogram \mathbf{f}' . Otherwise, the result is a type error.

Note that the return type in \mathbf{f} has already been annotated by *annotate_func_sig*.

20.1 TypingRule.Subprogram

20.1.1 Prose

All of the following apply:

- \mathbf{f} is a `func` AST node subprogram body `body`;
- annotating the block `body` in `tenv` as per Section 17.1 yields `new_body` *//* $\#TE$;
- \mathbf{f}' is \mathbf{f} with the subprogram body substituted with `new_body`.

20.1.2 Formally

$$\begin{array}{c}
f \stackrel{\text{is}}{=} \{ \\
\quad \text{name} : \text{id}, \\
\quad \text{parameters} : p, \\
\quad \text{args} : \text{args}, \\
\quad \text{body} : \text{SB_ASL}(\text{body}), \\
\quad \text{return_type} : t, \\
\quad \text{subprogram_type} : \text{sub_program_type} \\
\} \\
\text{annotate_block}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{new_body} \quad // \quad \#TE \\
f' := \{ \\
\quad \text{name} : \text{id}, \\
\quad \text{parameters} : p, \\
\quad \text{args} : \text{args}, \\
\quad \text{body} : \text{SB_ASL}(\text{new_body}), \\
\quad \text{return_type} : t, \\
\quad \text{subprogram_type} : \text{sub_program_type} \\
\} \\
\hline
\text{annotate_subprogram}(\text{tenv}, f) \xrightarrow{\text{type}} f'
\end{array}$$

Chapter 21

Typing of Global Declarations

The function

$$\text{typecheck_decl}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}}^{\text{d}}) \longrightarrow (\overbrace{\text{decl} \times \text{SE}}^{\text{new_d} \quad \text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a global declaration `d` in the static environment `tenv`, yielding an annotated global declaration `new_d` and modified environment `new_tenv`. Otherwise, the result is a type error.

One of the following applies:

- `TypingRule.TypecheckFunc` (see Section 21.1)
- `TypingRule.TypecheckGlobalStorage` (see Section 21.2)
- `TypingRule.TypecheckTypeDecl` (see Section 21.3)

We also define the following helper rules:

- `TypingRule.AnnotateAndDeclareFunc` (see Section 21.4)
- `TypingRule.AnnotateFuncSig` (see Section 21.5)
- `TypingRule.UseFuncSig` (Section 21.6)
- `TypingRule.GetUndeclaredDefining` (see Section 21.7)
- `TypingRule.ScanForParams` (see Section 21.8)
- `TypingRule.AnnotateParams` (see Section 21.9)
- `TypingRule.AnnotateOneParam` (see Section 21.10)
- `TypingRule.ArgsAsParams` (see Section 21.11)
- `TypingRule.ArgAsParam` (see Section 21.12)

- `TypingRule.AnnotateParamType` (see Section 21.13)
- `TypingRule.AnnotateArgs` (see Section 21.14)
- `TypingRule.AnnotateOneArg` (see Section 21.15)
- `TypingRule.AnnotateReturnType` (see Section 21.16)
- `TypingRule.DeclareOneFunc` (see Section 21.17)
- `TypingRule.SubprogramClash` (see Section 21.18)
- `TypingRule.AddNewFunc` (see Section 21.19)
- `TypingRule.CheckSetterHasGetter` (see Section 21.20)
- `TypingRule.AddSubprogram` (see Section 21.21)
- `TypingRule.DeclareGlobalStorage` (see Section 21.22)
- `TypingRule.AnnotateTypeOpt` (see Section 21.23)
- `TypingRule.AnnotateExprOpt` (see Section 21.24)
- `TypingRule.AnnotateInitType` (see Section 21.25)
- `TypingRule.AddGlobalStorage` (see Section 21.26)
- `TypingRule.DeclareType` (see Section 21.27)
- `TypingRule.AnnotateExtraFields` (see Section 21.28)
- `TypingRule.AnnotateEnumLabels` (see Section 21.29)
- `TypingRule.DeclareConst` (see Section 21.30)

21.1 TypingRule.TypecheckFunc

21.1.1 Prose

All of the following apply:

- `d` is a subprogram AST node with a subprogram definition `f`, that is, `D_Func(f)`;
- annotating and declaring the subprogram for `f` in `tenv` as per Section 21.4 yields the new environment `new_tenv` and a subprogram definition `f1` *//^{#TE}*;
- annotating the subprogram definition `f1` in `tenv` as per Chapter 20 yields the annotated subprogram definition `f2` *//^{#TE}*;
- define `new_d` as the subprogram AST node with `f2`, that is, `D_Func(f2)`.

21.1.2 Formally

$$\begin{array}{c}
 f \stackrel{\text{is}}{=} \{\text{body} : \text{SB_ASL}, \dots\} \\
 \text{annotate_and_declare_func}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{new_tenv}, f1) \quad // \text{ \#TE} \\
 \text{annotate_subprogram}(\text{new_tenv}, f1) \xrightarrow{\text{type}} f2 \quad // \text{ \#TE} \\
 \hline
 \text{typecheck_decl}(\text{tenv}, \underbrace{\text{D_Func}(f)}_d) \xrightarrow{\text{type}} (\underbrace{\text{D_Func}(f2)}_{\text{new_d}}, \text{new_tenv})
 \end{array}$$

21.2 TypingRule.TypecheckGlobalStorage

21.2.1 Prose

All of the following apply:

- d is a global storage declaration with description gsd , that is, $\text{D_GlobalStorage}(\text{gsd})$;
- declaring the global storage with description gsd in tenv yields the new environment new_tenv and new global storage description gsd' // \#TE ;
- define new_d as the global storage declaration with description gsd' , that is, $\text{D_GlobalStorage}(\text{gsd}')$.

21.2.2 Formally

$$\begin{array}{c}
 \text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{gsd}') \quad // \text{ \#TE} \\
 \hline
 \text{typecheck_decl}(\text{tenv}, \underbrace{\text{D_GlobalStorage}(\text{gsd})}_d) \xrightarrow{\text{type}} \\
 \underbrace{(\text{D_GlobalStorage}(\text{gsd}'))}_{\text{new_d}}, \text{new_tenv}
 \end{array}$$

21.3 TypingRule.TypecheckTypeDecl

21.3.1 Prose

All of the following apply:

- d is a type declaration with identifier x , type ty , and **optional** field initializers s , that is, $\text{D_TypeDecl}(x, \text{ty}, s)$;
- declaring the type described by (x, ty, s) in tenv as per Section 26.9 yields the modified environment new_tenv // \#TE ;
- define new_d as d .

21.3.2 Formally

$$\frac{\text{declare_type}(\text{tenv}, x, \text{ty}, s) \xrightarrow{\text{type}} \text{new_tenv} \text{ // } \#TE}{\text{typecheck_decl}(\text{tenv}, \overbrace{D_TypeDecl(x, \text{ty}, s)}^d) \xrightarrow{\text{type}} (\overbrace{d}^{\text{new.d}}, \text{new_tenv})}$$

21.4 TypingRule.AnnotateAndDeclareFunc

The function

$$\text{annotate_and_declare_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a subprogram definition `func_sig` in the static environment `tenv`, yielding a new subprogram definition `new_func_sig` and modified static environment `new_tenv`. Otherwise, the result is a type error.

21.4.1 Prose

All of the following apply:

- annotating the signature of `func_sig` in `tenv` as per Section 21.5 yields the environment `tenv1` and subprogram definition `func_sig1` *//* `#TE`;
- declaring the subprogram defined by `func_sig1` in `tenv1` as per Section 21.17 yields the environment `new_tenv` and new func node `new_func_sig` *//* `#TE`.

21.4.2 Formally

$$\frac{\begin{array}{l} \text{annotate_func_sig}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{tenv1}, \text{func_sig1}) \text{ // } \#TE \\ \text{declare_one_func}(\text{tenv1}, \text{func_sig1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig}) \text{ // } \#TE \end{array}}{\text{annotate_and_declare_func}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig})}$$

21.5 TypingRule.AnnotateFuncSig

The function

$$\text{annotate_func_sig}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the signature of a function definition `func_sig` in the static environment `tenv`, yielding a new function definition `new_func_sig` and modified static environment `new_tenv`. Otherwise, the result is a type error.

21.5.1 Prose

All of the following apply:

- `tenv1` is the static environment comprised of the global environment of `tenv` and an empty local environment;
- obtaining the variables appearing in the formal types in `func_sig` that may be parameter-defining in `tenv1` via *get_undeclared_defining* yields `potential_params`;
- annotating the parameters explicitly listed in `func_sig` (`func_sig.parameters`) yields environment `tenv2`, in which the parameters are declared, and the function `declared_params`, which binds parameter identifiers to their types`//#TE`;
- annotating arguments from `func_sig` that serve as parameters in `tenv2` yields the list of parameters `arg_params` and modified environment `tenv3``//#TE`;
- `parameters` is the list `declared_params` concatenated with `arg_params` with each type transformed to an *optional* type;
- annotating the arguments listed in `func_sig` in `tenv3` with `tenv2` via *annotate_args* yields the list of annotated arguments `args` and modified environment `tenv4``//#TE`;
- annotating the return type of `func_sig` in `tenv4` with `tenv3` via *annotate_return_type* yields the annotated return type `return_type` and modified environment `tenv5``//#TE`;
- `new_func_sig` is `func_sig` with the listed of parameters substituted with `parameters`, the list of arguments substituted with `args`, and return type substituted with `return_type`;
- `new_tenv` is `tenv5`.

21.5.2 Formally

$$\begin{array}{c}
\text{tenv1} := (G^{\text{tenv}}, L^{\emptyset_{\text{tenv}}}) \\
\text{get_undeclared_defining}(\text{tenv1}, \text{func_sig}) \xrightarrow{\text{type}} \text{potential_params} \\
\text{annotate_params}(\text{tenv1}, \text{potential_params}, \text{func_sig.parameters}, (\text{tenv1}, \emptyset_{\lambda})) \xrightarrow{\text{type}} \\
\quad (\text{tenv2}, \text{declared_params}) \quad // \text{ \#TE} \\
\text{args_as_params}(\text{tenv2}, \text{func_sig}) \xrightarrow{\text{type}} (\text{tenv3}, \text{arg_params}) \quad // \text{ \#TE} \\
\text{parameters} := \begin{array}{l} [(id, t) \in \text{declared_params} : (id, \langle t \rangle)] + \\ [(id, t) \in \text{arg_params} : (id, \langle t \rangle)] \end{array} \\
\text{annotate_args}(\text{tenv3}, \text{tenv2}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\text{tenv4}, \text{args}) \quad // \text{ \#TE} \\
\text{annotate_return_type}(\text{tenv4}, \text{tenv3}, \text{func_sig.return_type}) \xrightarrow{\text{type}} \\
\quad (\text{tenv5}, \text{return_type}) \quad // \text{ \#TE} \\
\text{new_func_sig} := \{ \quad \begin{array}{ll} \text{name} & : \text{func_sig.name}, \\ \text{parameters} & : \text{parameters}, \\ \text{args} & : \text{args}, \\ \text{body} & : \text{func_sig.body}, \\ \text{return_type} & : \text{return_type}, \\ \text{subprogram_type} & : \text{func_sig.sub_program_type} \end{array} \\
\quad \} \\
\hline
\text{annotate_func_sig}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\overbrace{\text{tenv5}}^{\text{new_tenv}}, \text{new_func_sig}, \text{arg_params})
\end{array}$$

21.6 TypingRule.UseFuncSig

The function

$$\text{use_func_sig}(\overbrace{\text{func}}^f) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the subprogram signature given by **f** depends on.

21.6.1 Prose

Define **ids** as the union of applying *use_ty* to every type of an argument of **f** and applying *use_ty* to the *optional* return type of **f**.

21.6.2 Formally

$$\text{use_func_sig}(\text{f}) \xrightarrow{\text{type}} \overbrace{\bigcup_{(_, t) \in \text{f.args}} \text{use_ty}(t) \cup \text{use_ty}(\text{f.return_type})}^{\text{ids}}$$

21.7 TypingRule.GetUndeclaredDefining

The function

$$\text{get_undeclared_defining}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}$$

scans the list of types appearing in `func_sig.args` and in the return type and returns the set of identifiers that may be parameter-defining in `tenv`.

21.7.1 Example

In the following specification, the set of identifiers that may correspond to parameters of the function `signature_example` is $\{A\}$, since `A` appears in the type `bits(A)` of the argument `bv`.

```
constant W = 4;

func signature_example{A}{
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

21.7.2 Prose

All of the following apply:

- define `formal_types` to consist of the types associated with the list of arguments in `func_sig` and the return type in `func_sig`, if one exists;
- scanning each type `t` in `formal_types` via `scan_for_params` yields the set `paramst`;
- `potential_params` is the union of `paramst` for each type `t` in `formal_types`.

21.7.3 Formally

$$\frac{\begin{array}{l} \text{formal_types} := [(_, t) \in \text{func_sig.args} : t] + \\ \text{choice}(\text{func_sig.return_type} = \langle \text{ret_ty} \rangle, [\text{ret_ty}], []) \\ t \in \text{formal_types} : \text{scan_for_params}(\text{tenv}, t) \xrightarrow{\text{type}} \text{params}_t \end{array}}{\text{get_undeclared_defining}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \overbrace{\bigcup_{t \in \text{tys}} \text{params}_t}^{\text{potential_params}}}$$

21.8 TypingRule.ScanForParams

The function

$$\text{scan_for_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}$$

scans a single type `ty` in `tenv` and returns the set of identifiers that may be parameters in `tenv`.

21.8.1 Example

Consider the following specification:

```
constant W = 4;

func signature_example{A}{
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

Scanning each type in the signature of the function `signature_example` yields the following results:

Expression	Result	Reason
<code>bits(A)</code>	<code>{A}</code>	A is a variable expression and A is not defined in the environment.
<code>bits(W)</code>	<code>∅</code>	W is defined in the environment.
<code>bits(A+B)</code>	<code>∅</code>	A+B is not a variable expression.

21.8.2 Prose

One of the following applies:

- All of the following apply (TBITS_EVAR):
 - * `ty` is a bitvector type over a variable expression for `x`, that is, `T_Bits(E_Var(x), _)`;
 - * `potential_params` is the singleton set consisting of `x` if `x` is not defined as a storage type in `tenv` and the empty set, otherwise.
- All of the following apply (TBITS_OTHER):
 - * `ty` is a bitvector type where the bitwidth expression is not a variable expression;
 - * `potential_params` is the empty set.

- All of the following apply (TTUPLE):
 - * `ty` is a tuple type over a list of types `tys`;
 - * obtaining the set of potential parameters for each type `t` of `tys` in `tenv` yields `paramst`;
 - * `potential_params` is the union of sets `paramst`, for each type `t` of `tys`.
- All of the following apply (OTHER):
 - * `ty` is neither a bitvector type or a tuple type;
 - * `potential_params` is the empty set.

TBITS_EVAR

$$\frac{\text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b \quad \text{potential_params} := \text{choice}(b, \{x\}, \emptyset)}{\text{scan_for_params}(\text{tenv}, \overbrace{T_Bits(E_Var(x), _)}^{\text{ty}}) \xrightarrow{\text{type}} \text{potential_params}}$$

TBITS_OTHER

$$\frac{\text{ast_label}(e) \neq E_Var}{\text{scan_for_params}(\text{tenv}, \overbrace{T_Bits(e, _)}^{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\text{potential_params}}_{\emptyset}}$$

TTUPLE

$$\frac{t \in \text{tys} : \text{scan_for_params}(\text{tenv}, t) \xrightarrow{\text{type}} \text{params}_t}{\text{scan_for_params}(\text{tenv}, \overbrace{T_Tuple(\text{tys})}^{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\bigcup_{t \in \text{tys}} \text{params}_t}_{\text{potential_params}}}$$

OTHER

$$\frac{\text{ast_label}(\text{ty}) \notin \{T_Bits, T_Tuple\}}{\text{scan_for_params}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \underbrace{\text{potential_params}}_{\emptyset}}$$

21.9 TypingRule.AnnotateParams

The function

$$\text{annotate_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{parameters}}, (\overbrace{\text{SE}}^{\text{tenv1'}} \times \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{acc}})) \longrightarrow \\ (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{declared_params}}) \cup \overbrace{T_TypeError}^{\#TE}$$

scans the list of explicitly defined parameters `parameters` with respect to the set of potential parameters `potential_params` in `tenv` and then updates a pair consisting of

an updated environment `tenv1'`, which accumulates local storage declarations for the parameters, and a function `acc`, which maps identifiers corresponding to parameters to their associated types. The updated pair is given in `new_tenv` and `declared_params`. Otherwise, the result is a type error.

21.9.1 Example

In the following specification, the list of explicitly defined parameters of the function `signature_example` is `{A}`. Therefore, `declared_params` binds `A` to the type `integer{A}` and `new_tenv` effectively reflects an added declaration `let A: integer{A}`.

```
constant W = 4;

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

21.9.2 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `parameters` is the empty list;
 - * `new_tenv` is `tenv1'`;
 - * `declared_params` is `acc`.
- All of the following apply (NON_EMPTY):
 - * `parameters` is a list with `(x, ty_opt)` as its `head` and `parameters1` as its `tail`;
 - * applying *annotate_one_param* to the parameter `(x, ty_opt)` in `tenv1` with `potential_params` and the pair `(tenv1', acc)` yields the pair `(tenv1'', acc')`^{#TE};
 - * annotating the parameter list `parameters1` in `tenv1` with `potential_params`, starting with the pair `(tenv1'', acc')` yields the pair `(new_tenv, declared_params)`.

EMPTY

21.10 TypingRule.AnnotateOneParam

$$\text{annotate_one_param}(\overset{\text{tenv}}{\text{SE}}, \overset{\text{potential_params}}{\mathcal{P}(\text{identifier})}, \overset{x}{(\text{identifier} \times \overset{\text{ty_opt}}{\langle \text{ty} \rangle})}, \overset{\text{tenv1}'}{(\text{SE} \times \overset{\text{acc}}{\text{identifier} \rightarrow \text{ty}})}) \\ \rightarrow (\overset{\text{new_tenv}}{(\text{SE} \times \overset{\text{declared_params}}{\text{identifier} \rightarrow \text{ty}})} \cup \overset{\text{\#TE}}{\text{TTypeError}})$$

21.10.1 Prose

- checking that `x` is not defined as a variable in `tenv1` yields `TRUE//#TE`;
- checking whether `x` is included in the set `potential_params` yields `TRUE` or a type error indicating that each parameter must have a defining argument, thus short-circuiting the entire rule;
- One of the following applies:
 - * All of the following apply (`TYPE_PARAMETERIZED`):

- `ty_opt` is either `None` or a `parameterized integer type`;
- `t` is defined as the `parameterized integer type` for the identifier `x`.
- * All of the following apply (`TYPE_ANNOTATED`):
 - `ty_opt` is the type `t1`, which is not the unconstrained integer type;
 - annotating `t1` in `tenv1` yields `t//#TE`.
- checking that `t` is a constrained integer in `tenv1` via `check_constrained_integer` yields `TRUE//#TE`;
- adding the local storage element given by the identifier `x`, type `t`, and local declaration keyword `LDK_Let` in `tenv1'` yields `new_tenv`;
- `declared_params` is `acc` updated by the binding of `x` to `t`.

21.10.2 Formally

TYPE_PARAMETERIZED

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv1}', x) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
 \text{check}(x \in \text{potential_params}, \text{TE_PWD}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\
 (\text{ty_opt} = \text{None} \vee \text{ty_opt} = \langle \text{unconstrained_integer} \rangle) \\
 \hline
 t := \text{T_Int}(\text{Parameterized}(x)) \quad \text{check_constrained_integer}(\text{tenv1}, t) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
 \text{add_local}(\text{tenv1}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{declared_params} := \text{acc}[x \mapsto t] \\
 \hline
 \text{annotate_one_param}(\text{tenv1}, \text{potential_params}, (x, \text{ty_opt}), (\text{tenv1}', \text{acc})) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{declared_params})
 \end{array}$$

TYPE_ANNOTATED

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv1}', x) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
 \text{check}(x \in \text{potential_params}, \text{TE_PWD}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\
 t1 \neq \text{unconstrained_integer} \quad \text{annotate_type}(\text{FALSE}, \text{tenv1}, t1) \xrightarrow{\text{type}} t \parallel \text{\#TE} \\
 \text{check_constrained_integer}(\text{tenv1}, t) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
 \hline
 \text{add_local}(\text{tenv1}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{declared_params} := \text{acc}[x \mapsto t] \\
 \hline
 \text{annotate_one_param}(\text{tenv1}, \text{potential_params}, (x, \overbrace{\langle t1 \rangle}^{\text{ty_opt}}), (\text{tenv1}', \text{acc})) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{declared_params})
 \end{array}$$

21.11 TypingRule.ArgsAsParams

The function

$$\begin{array}{c}
 \text{args_as_params}(\overbrace{\text{SE}}^{\text{tenv1}}, \overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{func}}^{\text{func_sig}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{declared_params}}) \longrightarrow \\
 \text{new_tenv} \quad \overbrace{\text{arg_params}}^{\text{\#TE}} \\
 \hline
 (\overbrace{\text{SE}}^{\text{new_tenv}} \times \text{identifier} \rightarrow \text{ty}) \cup \text{TTypeError}
 \end{array}$$

scans the list of arguments in `func` (`func.args`) to find the ones that serve as implicit parameters in `tenv1` and are not already included in the domain of `declared_params`. The found parameters are added as local declarations to `tenv2`, resulting in `new_tenv`, and are used to update `arg_params`. Otherwise, the result is a type error.

21.11.1 Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * the subprogram defined by `func_sig` has an empty list of arguments;
 - * `new_tenv` is `tenv2`;
 - * `arg_params` is the empty list.
- All of the following apply (`NON_EMPTY`):
 - * the subprogram defined by `func_sig` has arguments `arg1..k`;
 - * obtaining the identifiers that can serve as parameters in the types of the formal arguments of `func_sig` and in its return type yields `used1`;
 - * the set `used` contains all identifiers `s` from `used1` that are undefined in `tenv1` and are not bound in `declared_params`;
 - * the following premises define the sequences `arg_params1..k` and `tenv20..k` as follows;
 - * `arg_params1` is `declared_params`;
 - * `tenv20` is `tenv2`;
 - * for $i = 1..k$, annotating the argument `argi` in `tenv2` with `used` and the static environment `tenv2i-1` via *arg-as-param* yields `tenv2i` and `arg_paramsi` */*TE*;
 - * `new_tenv` is `tenv2k`;
 - * `arg_params` is `arg_paramsk`.

21.11.2 Example

In the following specification, the argument `B` of the function `signature_example` is an implicit parameter as it appears in the type `bits(A+B)` (both for the argument `bv3` and as the return type) and it is not listed as an explicit parameter. Therefore, `new_tenv` will effectively contain the declaration `let B: integer{B}`.

```
constant W = 4;
```

```
func signature_example{A}{
  B: integer,
  bv: bits(A),
  bv2: bits(W),
```

```

    bv3: bits(A+B),
    C: integer) => bits(A+B)
begin
    return [bv, Ones(B)];
end

```

21.11.3 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{func_sig.args} = [] \\
 \hline
 \text{args_as_params}(\text{tenv1}, \text{tenv2}, \text{func_sig}, \text{declared_params}) \xrightarrow{\text{type}} \\
 \quad \underbrace{\text{new_tenv}}_{(\text{tenv2}, \quad \underbrace{\text{arg_params}}_{[]})}
 \end{array}$$

$$\begin{array}{c}
 \text{NON_EMPTY} \\
 \text{func_sig.args} \stackrel{\text{is}}{=} \text{arg}_{1..k} \\
 \text{use_func_sig}(\text{func_sig}) \xrightarrow{\text{type}} \text{used1} \quad s \in \text{used1} : \text{is_undefined}(\text{tenv1}, s) \xrightarrow{\text{type}} b_s \\
 \text{used} := \{s \in \text{used1} \wedge b_s \wedge \text{declared_params}(s) = \perp : s\} \\
 \text{arg_params}_1 := \text{declared_params} \quad \text{tenv2}_0 := \text{tenv2} \\
 i = 1..k : \text{arg_as_param}(\text{tenv2}, \text{used}, \text{arg}_i, \text{tenv2}_{i-1}) \xrightarrow{\text{type}} (\text{tenv2}_i, \text{arg_params}_i) \quad // \text{ \#TE} \\
 \hline
 \text{args_as_params}(\text{tenv1}, \text{tenv2}, \text{func_sig}, \text{declared_params}) \xrightarrow{\text{type}} (\underbrace{\text{new_tenv}}_{\text{tenv2}_k}, \underbrace{\text{arg_params}}_{\text{arg_params}_k})
 \end{array}$$

21.12 TypingRule.ArgAsParam

The function

$$\text{arg_as_param} \left(\begin{array}{c} \text{tenv2} \\ \underbrace{\text{SE}} \\ \text{used} \\ \underbrace{\mathcal{P}(\text{identifier})}_{\text{x}} \text{ , } \underbrace{\text{ty}}_{\text{ty}} \\ (\text{identifier} \times \text{ty}) \\ \text{tenv2}' \quad \text{acc} \\ (\underbrace{\text{SE}} \times \text{identifier} \rightarrow \text{ty}) \end{array} \right) \longrightarrow (\underbrace{\text{new_tenv}}_{\text{SE}} \times \underbrace{\text{acc}'}_{\text{identifier} \rightarrow \text{ty}}) \cup \underbrace{\text{\#TE}}_{\text{TTypeError}}$$

checks whether the argument given by x and type ty is an implicit parameter by checking whether it appears in used and not in acc . If it is identified as an implicit parameter, it is used to update $\text{tenv2}'$ to yield new_tenv and to update acc to yield acc' . Otherwise, the result is a type error.

21.12.1 Prose

One of the following applies:

- All of the following apply (NOT_USED):

- * x is not a member of `used`;
- * `new_tenv` is `tenv2'`;
- * `acc'` is `acc`.

- All of the following apply (`USED`):

- * x is a member of `used`;
- * checking that x is not declared in `tenv2'` yields `TRUE` *//* `#TE`;
- * annotating `ty` with identifier x as a potential parameter type in `tenv2`, which is an environment to which all explicit parameters have been added but implicit parameters were not added to, via `annotate_param_type` yields `t` *//* `#TE`;
- * checking whether `t` is a constrained integer type in `tenv2` yields `TRUE` *//* `#TE`;
- * adding x as a local storage element to `tenv2'` with type `t` and local declaration keyword `LDK_Let` yields `new_tenv`;
- * `acc'` is `acc` updated by binding x to `t`.

21.12.2 Formally

$$\begin{array}{c}
 \text{NOT_USED} \\
 \hline
 x \notin \text{used} \\
 \hline
 \text{arg_as_param}(\text{tenv2}, \text{used}, (x, \text{ty}), (\text{tenv2}', \text{acc})) \xrightarrow{\text{type}} (\overbrace{\text{tenv2}'}^{\text{new_tenv}}, \overbrace{\text{acc}'}^{\text{acc}'})
 \end{array}$$

$$\begin{array}{c}
 \text{USED} \\
 \hline
 x \in \text{used} \quad \text{check_var_not_in_env}(\text{tenv2}', x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \quad \text{annotate_param_type}(\text{tenv2}, \text{ty}, x) \xrightarrow{\text{type}} t \text{ // } \#TE \\
 \quad \text{check_constrained_integer}(\text{tenv2}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{add_local}(\text{tenv2}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{acc}' := \text{acc}[x \mapsto t] \\
 \hline
 \text{arg_as_param}(\text{tenv2}, \text{used}, (x, \text{ty}), (\text{tenv2}', \text{acc})) \xrightarrow{\text{type}} (\text{new_tenv}, \text{acc}')
 \end{array}$$

21.13 TypingRule.AnnotateParamType

The function

$$\text{annotate_param_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{identifier}}^x) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}$$

annotates the type `ty` in `tenv`, considering it as a subprogram parameter with identifier x , yielding the type `new_tenv`. It is assumed that `tenv` is an environment to which all explicitly defined parameters of the subprogram in context were added to, but the implicitly defined parameters were not added to. Otherwise, the result is a type error.

21.13.1 Prose

One of the following applies:

- All of the following apply (TINT_UNCONSTRAINED):
 - * `ty` is the unconstrained integer type;
 - * `new_ty` is the [parameterized integer type](#) for the identifier `x`.
- All of the following apply (OTHER):
 - * `ty` is not the unconstrained integer type;
 - * annotating the type `ty` in `tenv` yields `new_ty` [//](#) `#TE`.

TINT_UNCONSTRAINED

$$\text{annotate_param_type}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^{\text{ty}}, x) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{Parameterized}(x))}^{\text{new_ty}}$$

OTHER

$$\frac{\text{ty} \neq \text{unconstrained_integer} \quad \text{annotate_type}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{new_ty} \quad \text{//} \quad \text{\#TE}}{\text{annotate_param_type}(\text{tenv}, \text{ty}, x) \xrightarrow{\text{type}} \text{new_ty}}$$

21.14 TypingRule.AnnotateArgs

The function

$$\text{annotate_args}(\overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{func}}^{\text{func_sig}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{arg_params}}) \longrightarrow \overbrace{(\overbrace{\text{SE}}^{\text{new_tenv}} \times (\text{identifier} \times \text{ty})^*) \cup \text{\#TE}}^{\text{new_args}}$$

annotates the arguments listed in `func_sig` in the context of of `arg_params`, which binds parameters to their types, and two static environments: `tenv2` — the environment to which only explicit parameters were added, and `tenv3` — the environment to which all parameters (explicit and implicit) were added. The result is the environment `new_tenv` where all arguments have been declared and the annotated list of arguments `new_args`. Otherwise, the result is a type error.

21.14.1 Prose

One of the following applies:

- All of the following apply (NO_ARGS):
 - * the function defined by `func_sig` has an empty list of arguments;

- * `new_tenv` is `tenv3`;
- * `new_args` is the empty list.
- All of the following apply (`SOME_ARGS`):
 - * the function defined by `func_sig` has arguments `arg1..k`;
 - * the following premises define the sequence of static environments `tenv30..k` and list of typed identifiers `new_arg1..k`;
 - * `tenv30` is `tenv3`;
 - * annotating the argument `argi` in the context of `tenv2`, `tenv3`, `arg_params`, and `tenv3i-1` via `annotate_one_arg` yields `(tenv3i, new_argi)` *//TE*;
 - * `new_tenv` is `tenv3k`;
 - * `new_args` is the list `new_arg1..k`.

21.14.2 Example

In the following specification, the annotated arguments are `bv`, `bv2`, `bv3`, and `C`. The argument `B` is not annotated as an argument since it is classified and annotated as a parameter.

```
constant W = 4;

func signature_example{A}{
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

21.14.3 Formally

$$\begin{array}{c}
\text{NO_ARGS} \\
\hline
\text{func_sig.args} = [] \\
\hline
\text{annotate_args}(\text{tenv2}, \text{tenv3}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\overbrace{\text{tenv3}}^{\text{new_tenv}}, \overbrace{[]}^{\text{new_args}}) \\
\\
\text{SOME_ARGS} \\
\text{func_sig.args} \stackrel{\text{is}}{=} \text{arg}_{1..k} \quad \text{tenv3}_0 := \text{tenv3} \\
i = 1..k : \text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3}_{i-1}, \text{arg}_i)) \xrightarrow{\text{type}} (\text{tenv3}_i, \text{new_arg}_i) \text{ // \#TE} \\
\text{new_args} := [i = 1..k : \text{new_arg}_i] \\
\hline
\text{annotate_args}(\text{tenv2}, \text{tenv3}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\overbrace{\text{tenv3}_k}^{\text{new_tenv}}, \text{new_args})
\end{array}$$

21.15 TypingRule.AnnotateOneArg

The function

$$\text{annotate_one_arg}(\overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{arg_params}}, (\overbrace{\text{SE}}^{\text{tenv3'}} \times (\overbrace{x}^{\text{identifier}} \times \overbrace{\text{ty}}^{\text{ty}}))) \rightarrow \\ (\overbrace{\text{SE}}^{\text{new_tenv}} \times (\overbrace{x}^{\text{identifier}} \times \overbrace{\text{ty}'}^{\text{ty'}})) \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

annotates the argument x of type ty in the context of arg_params , which binds parameters to their types, and the following static environments: tenv2 — the environment to which only explicit parameters were added, tenv3 — tenv2 with the addition of implicit parameters, and tenv3' — same as tenv3 but updated with previously annotated arguments. The result is the updated environment new_tenv with the added declaration for the current argument and the annotated argument, which has the same identifier x and the annotated type ty' . Otherwise, the result is a type error.

21.15.1 Prose

One of the following applies:

- All of the following apply (PARAM):
 - * x is not bound in arg_params ;
 - * annotating the type ty in tenv2 yields ty' ;
 - * new_tenv is tenv3' .
- All of the following apply (NOT_PARAM):
 - * x is bound in arg_params ;
 - * checking that x is not defined in tenv3' yields $\text{TRUE} \text{ \#TE}$;
 - * annotating the type ty in tenv3 yields ty' ;
 - * adding a local storage element x with type ty' and local declaration keyword `LDK_Let` yields new_tenv .

21.15.2 Formally

PARAM

$$\frac{\text{arg_params}(x) \neq \perp \quad \text{annotate_type}(\text{tenv2}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \text{ \#TE}}{\text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3'}, (x, \text{ty}))) \xrightarrow{\text{type}} (\overbrace{\text{tenv3'}}^{\text{new_tenv}}, (x, \text{ty}'))}$$

NOT_PARAM

$$\frac{\text{arg_params}(x) = \perp \quad \text{check_var_not_in_env}(\text{tenv3'}, x) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \quad \text{annotate_type}(\text{tenv3}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \text{ \#TE} \quad \text{add_local}(\text{tenv3'}, x, \text{ty}', \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv}}{\text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3'}, (x, \text{ty}))) \xrightarrow{\text{type}} (\text{new_tenv}, (x, \text{ty}'))}$$

21.16 TypingRule.AnnotateReturnType

The function

$$\text{annotate_return_type}(\overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{SE}}^{\text{tenv4}}, \overbrace{\langle \text{ty} \rangle}^{\text{return_type}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{ty}}^{\text{new_return_type}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the **optional** return type **return_type** in the context of the static environment **tenv3** where all parameters have been added to, and **tenv4** where all parameters and arguments have been added to. The result is the static environment **new_tenv**, which is **tenv4** with the annotated return type and the **optional** annotated return type **new_return_type**. Otherwise, the result is a type error.

21.16.1 Prose

One of the following applies:

- All of the following apply (NO_RETURN_TYPE):
 - * **return_type** is **None**;
 - * **new_tenv** is **tenv4**;
 - * **new_return_type** is **None**.
- All of the following apply (HAS_RETURN_TYPE):
 - * **return_type** is $\langle \text{ty} \rangle$;
 - * annotating **ty** in **tenv3** yields **ty'** // **#TE**;
 - * **new_return_type** is $\langle \text{ty}' \rangle$;
 - * **new_tenv** is **tenv4** with its local environment updated by binding its **return_type** field to **new_return_type**.

21.16.2 Formally

$$\begin{array}{c}
 \text{NO_RETURN_TYPE} \\
 \text{annotate_return_type}(\text{tenv3}, \text{tenv4}, \overbrace{\text{None}}^{\text{return_type}}) \xrightarrow{\text{type}} (\overbrace{\text{tenv4}}^{\text{new_tenv}}, \overbrace{\text{None}}^{\text{new_return_type}}) \\
 \\
 \text{HAS_RETURN_TYPE} \\
 \frac{\text{annotate_type}(\text{tenv3}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \quad \text{new_return_type} := \langle \text{ty}' \rangle \quad \text{new_tenv} := (G^{\text{tenv4}}, L^{\text{tenv4}}[\text{return_type} \mapsto \text{new_return_type}])}{\text{annotate_return_type}(\text{tenv3}, \text{tenv4}, \overbrace{\langle \text{ty} \rangle}^{\text{return_type}}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_return_type})}
 \end{array}$$

21.17 TypingRule.DeclareOneFunc

The function

$$\text{declare_one_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

checks that a subprogram defined by `func_sig` can be added to the static environment `tenv`, resulting in an annotated function definition `new_func_def` and new static environment `new_tenv`. Otherwise, the result is a type error.

21.17.1 Prose

All of the following apply:

- `func_sig` has name `name`, arguments `args`, and type `sub_program_type`, that is,

```
func_sig := {
    name      : name,
    parameters : p,
    args      : args,
    body      : SB_ASL(bd),
    return_type : t,
    subprogram_type : sub_program_type
};
```

- adding a new subprogram with `name`, `args`, and `sub_program_type` to `tenv` yields the new environment `tenv1` and new name `name' // \#TE`;
- checking that `name'` is not already declared in the global environment of `tenv1` yields `TRUE // \#TE`;
- ensuring that each setter has a getter given `func_sig` in `tenv` yields `TRUE // \#TE`;
- `func_sig1` is `func_sig` with `name` substituted by `name1`;
- adding a subprogram with name `name'` and definition `func_sig1` to `tenv1` yields `new_tenv // \#TE`.

21.17.2 Formally

$$\begin{array}{l}
 \text{func_sig} := \{ \\
 \quad \text{name} : \text{name}, \\
 \quad \text{parameters} : \text{p}, \\
 \quad \text{args} : \text{args}, \\
 \quad \text{body} : \text{SB_ASL}(\text{bd}), \\
 \quad \text{return_type} : \text{t}, \\
 \quad \text{subprogram_type} : \text{sub_program_type} \\
 \} \\
 \text{add_new_func}(\text{tenv}, \text{name}, \text{args}, \text{sub_program_type}) \xrightarrow{\text{type}} (\text{tenv1}, \text{name}') \text{ // \#TE} \\
 \text{check_var_not_in_genv}(\text{tenv1}, \text{name}') \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{check_setter_has_getter}(\text{tenv1}, \text{func_sig}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{new_func_sig} := \{ \\
 \quad \text{name} : \text{name}', \\
 \quad \text{parameters} : \text{p}, \\
 \quad \text{args} : \text{args}, \\
 \quad \text{body} : \text{SB_ASL}(\text{bd}), \\
 \quad \text{return_type} : \text{t}, \\
 \quad \text{subprogram_type} : \text{sub_program_type} \\
 \} \\
 \hline
 \text{add_subprogram}(\text{tenv1}, \text{name}', \text{func_sig1}) \xrightarrow{\text{type}} \text{new_tenv} \text{ // \#TE} \\
 \text{declare_one_func}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig})
 \end{array}$$

21.18 TypingRule.SubprogramClash

The function

$$\text{subprogram_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type}}, \overbrace{\text{ty}^*}^{\text{formal_types}}) \longrightarrow \overbrace{\text{B}}^{\text{b}} \cup \overbrace{\text{TE}}^{\text{\#TE}}$$

checks whether the unique subprogram associated with **name** clashes with another subprogram that has subprogram type **subpgm_type** and list of formal types **formal_types**, yielding a Boolean value in **b**. Otherwise, the result is a type error.

The function is only defined when there exists a binding for **name** in the **subprograms** map of **tenv**.

21.18.1 Prose

All of the following apply:

- the subprogram type associated with the unique subprogram named by **name** is **name_subpgmtype**;
- applying *subprogram_types.clash* to **name_subpgmtype** and **subpgm_type** yields **TRUE**/**FALSE** (that is, if both **name_subpgmtype** and **subpgm_type** are **ST_Getter** or both are **ST_Setter** then the subprogram types are considered to be non-clashing and the entire rule short-circuits to **FALSE**);

- `name_args` is the list of pairs of types and identifiers associated with the function definition of `name` in `tenv`;
- determining whether there is an argument clash between `formal_types` and `name_formals` in `tenv` yields `b` $\#TE$.

21.18.2 Formally

We first introduce the helper predicate

$$\text{subprogram_types_clash}(\overbrace{\text{sub_program_type}}^{\text{subpgm_type1}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type2}}) \longrightarrow \overbrace{\mathbb{B}}^b$$

which defines whether two subprogram types are considered to be clashing:

$$\begin{array}{c} b1 := (\text{subpgm_type1} = \text{ST_Getter} \wedge \text{subpgm_type2} = \text{ST_Setter}) \vee \\ (\text{subpgm_type1} = \text{ST_Setter} \wedge \text{subpgm_type2} = \text{ST_Getter}) \\ b := \neg b1 \\ \hline \text{subprogram_types_clash}(\text{subpgm_type1}, \text{subpgm_type2}) \xrightarrow{\text{type}} b \\ \\ \text{name_subpgmtype} := G^{\text{tenv}}.\text{subprograms}(\text{name}).\text{sub_program_type} \\ \text{subprogram_types_clash}(\text{name_subpgmtype}, \text{subpgm_type}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\ \text{name_args} := G^{\text{tenv}}.\text{subprograms}(\text{name}).\text{args} \\ \text{has_arg_clash}(\text{formal_types}, \text{name_args}) \xrightarrow{\text{type}} b \\ \hline \text{subprogram_clash}(\text{tenv}, \text{name}', \text{subpgm_type}, \text{formal_types}) \xrightarrow{\text{type}} b \end{array}$$

21.19 TypingRule.AddNewFunc

The function

$$\text{add_new_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{typed_identifier}^*}^{\text{formals}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type}}) \longrightarrow \\ (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{S}}^{\text{new_name}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

ensures that the subprogram given by the identifier `name`, list of formals `formals`, and subprogram type `subpgm_type` has a unique name among all the potential subprograms that overload `name`. The result is the unique subprogram identifier `new_name`, which is used to distinguish it in the set of overloaded subprograms (that is, other subprograms that share the same name) and the environment `new_tenv`, which is updated with `new_name`. Otherwise, the result is a type error.

21.19.1 Prose

One of the following applies:

- All of the following apply (FIRST_NAME):
 - * the `subprogram_renamings` map in the global environment of `tenv` does not have a binding for `name`;
 - * `new_tenv` is `tenv` with the `subprogram_renamings` updated by binding `name` to the singleton set containing `name`.
- All of the following apply (NAME_EXISTS):
 - * the `subprogram_renamings` map in the global environment of `tenv` binds `name` to the set of strings `other_names`;
 - * `new_name` is the unique name that will be associated with the subprogram given by the identifier `name`, list of formals `formals`, and subprogram type `subpgm_type`. It is constructed by concatenating a hyphen (-) to `name`, followed by a string corresponding to the number of strings in `other_names`. Notice that this is not an ASL identifier, as ASL identifiers do not contain hyphens, which ensures that this string does not occur in any specification;
 - * `formal_types` is the list of types that appear in `formals` in the same order;
 - * checking for each `name'` in `other_names` whether the subprogram associated with `name'` clashes with the subprogram type `subpgm_type` and list of types `formal_types` yields `FALSE` or a type error that indicates there are multiply defined subprograms, which short-circuits the entire rule;
 - * `new_tenv` is `tenv` with the `subprogram_renamings` updated by binding `name` to the union of `other_names` and `{new_name}`.

21.19.2 Formally

We use the following functions to construct a unique string for each subprogram:

- The function $+: \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ concatenates two strings.
- The function `string_of_nat` : $\mathbb{N} \rightarrow \mathbb{S}$ converts a natural number to the corresponding string.

$$\begin{array}{c}
 \text{FIRST_NAME} \\
 \frac{G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \perp \quad \text{new_tenv} := (G^{\text{tenv}}.\text{subprogram_renamings}[\text{name} \mapsto \{\text{name}\}], L^{\text{tenv}})}{\text{add_new_func}(\text{tenv}, \text{name}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \overbrace{\text{name}}^{\text{new_name}})}
 \end{array}$$

$$\begin{array}{c}
\text{NAME_EXISTS} \\
\frac{
\begin{array}{l}
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{other_names} \\
k := |\text{other_names}| \quad \text{new_name} := \text{name} + "-" + \text{string_of_nat}(k) \\
\text{formal_types} := [(id, t) \in \text{formals} : t] \\
\left(\begin{array}{l}
\text{name}' \in \text{other_names} : \\
\text{subprogram_clash}(\text{tenv}, \text{name}', \text{subpgm_type}, \text{formal_types}) \xrightarrow{\text{type}} b_{\text{name}'} \quad // \quad \#TE
\end{array} \right) \\
\text{name}' \in \text{other_names} : \text{check}(\neg b_{\text{name}'}, \text{TE_SDM}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{new_tenv} := (G^{\text{tenv}}.\text{subprogram_renamings}[\text{name} \mapsto \text{other_names} \cup \{\text{new_name}\}], L^{\text{tenv}})
\end{array}
}{
\text{add_new_func}(\text{tenv}, \text{name}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_name})
}
\end{array}$$

21.20 TypingRule.CheckSetterHasGetter

The function

$$\text{check_setter_has_getter}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\text{TRUE}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the setter procedure given by `func.sig` has a corresponding getter, returning `TRUE` if this condition holds and a type error otherwise.

21.20.1 Prose

All of the following apply:

- checking that the subprogram type of `func.sig` is one of `ST.Setter` and `ST.EmptySetter` has one of two outcomes: `FALSE`, which satisfies the premise; or `TRUE`, which short-circuits the entire rule (since the subprogram is not any kind of setter and no getter is required);
- `view` the list of arguments of `func.sig` (that is, `func.sig.args`) as follows: the `head` is an argument that has the type `ret_type`; the `tail` is a list with arguments that have the types `arg_types`;
- applying `subprogram_for_name` to look up `tenv` for a subprogram with the name given by `func.sig` (that is, `func.sig.name`) yields a subprogram definition AST node `func.sig' // #TE`;
- define `wanted_getter_type` as `ST.Getter` if `func.sig.sub_program_type` is `ST.Setter` and `ST.EmptyGetter` otherwise (meaning, `func.sig.sub_program_type` is `ST.EmptySetter`);
- checking that `wanted_getter_type` is the same as `func.sig'.subprogram_type` yields `TRUE // #TE`;
- define `arg_types'` as the list of types appearing in the signature of `func.sig'` (that is, in `func.sig'.args`);

- checking, for each index i in the indices for `arg_types`, that the type at `arg_types[i]` and the type at `arg_types'[i]` are *type-equivalent* yields *TRUE* // #TE;
- checking that `ret_type` and `func_sig'.return_type` are *type-equivalent* yields *TRUE* // #TE;
- define b as *TRUE* (that is, unless the rule short-circuited with a type error).

21.20.2 Formally

We define the helper function

match_setter_type \triangleq [ST_Setter \mapsto ST_Getter, ST_EmptySetter \mapsto ST_EmptyGetter] .

$$\begin{array}{c}
\text{is_setter} := \text{func_sig.sub_program.type} \in \{\text{ST_Setter}, \text{ST_EmptySetter}\} \\
\text{bool.transition}(\text{is_setter}) \longrightarrow \text{FALSE} \text{ // } \text{TRUE} \\
\text{func_sig.args} \stackrel{\text{is}}{=} (_, \text{ret_type}) + \text{args} \quad \text{arg_types} := [(_, t) \in \text{args} : t] \\
\text{subprogram_for_name}(\text{tenv}, \text{func_sig.name}, \text{arg_types}) \xrightarrow{\text{type}} (_, _, \text{func_sig}') \text{ // } \#TE \\
\text{match_setter_type}(\text{func_sig.sub_program.type}) \xrightarrow{\text{type}} \text{wanted_getter.type} \\
\text{check}(\text{wanted_getter.type} = \text{func_sig'.sub_program.type}, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{arg_types}' := [(_, t) \in \text{func_sig'.args} : t] \\
i \in \text{indices}(\text{arg_types}) : \text{type_equal}(\text{arg_types}[i], \text{arg_types}'[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\
i \in \text{indices}(\text{arg_types}) : \text{check}(b_i, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{type_equal}(\text{ret_type}, \text{func_sig'.return_type}) \xrightarrow{\text{type}} b_{\text{ret}} \text{ // } \#TE \\
\text{check}(b_{\text{ret}}, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_setter_has_getter}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

21.21 TypingRule.AddSubprogram

The function

$$\text{add_subprogram}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{func}}^{\text{func_def}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

updates the global environment of `tenv` by mapping the (unique) subprogram identifier `name` to the function definition `func_def` in `tenv`, resulting in a new static environment `new_tenv`.

21.21.1 Prose

`new_tenv` is `tenv` with its *subprograms* component updated by binding `name` to `func_def`.

21.21.2 Formally

$$\frac{\text{new_tenv} := (G^{\text{tenv}}.\text{subprograms}[\text{name} \mapsto \text{func_def}], L^{\text{tenv}})}{\text{add_subprogram}(\text{tenv}, \text{name}, \text{func_def}) \xrightarrow{\text{type}} \text{new_tenv}}$$

21.22 TypingRule.DeclareGlobalStorage

The function

$$\text{declare_global_storage}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{global_decl}}^{\text{gsd}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}, \overbrace{\text{global_decl} \cup \text{TypeError}}^{\text{new_gsd} \quad \#TE}$$

annotates the global storage declaration `gsd` in the static environment `tenv`, yielding a modified static environment `new_tenv` and annotated global storage declaration `new_gsd`. Otherwise, the result is a type error.

21.22.1 Prose

All of the following apply:

- `gsd` is a global storage declaration with keyword `keyword`, initial value `initial_value`, optional type `ty_opt`, and name `name`;
- checking that `name` is not already declared in the global environment of `tenv` yields `TRUE/#TE`;
- annotating the optional type `ty_opt` in `tenv` via `annotate_type_opt` yields `ty_opt'/#TE`;
- annotating the optional expression `initial_value` in `tenv` via `annotate_expr_opt` yields `(initial_value_type, initial_value')/#TE`;
- choosing the correct type between `initial_value_type` and `ty_opt'` in `tenv` via `annotate_init_type` yields `declared_t`;
- adding a global storage element with name `name`, global declaration `keyword` and type `declared_t` to `tenv` via `add_global_storage` yields `tenv1/#TE`;
- One of the following applies:
 - * All of the following apply (CONSTANT):
 - `keyword` is `GDK.Constant` and therefore `initial_value` is some expression `e` (the ASL parser guarantees that the expression exists);
 - symbolically simplifying `e` in `tenv1` via `reduce_constants` yields the literal `v/#TE`;
 - `tenv2` is `tenv1` with its `constant_values` component updated by binding `name` to `v`;

- `new_gsd` is `gsd` with its type component as `ty_opt'`;
- `new_tenv` is `tenv2`.

* All of the following apply (`NON_CONSTANT`):

- `keyword` is not `GDK_Constant`;
- `new_tenv` is `tenv1`.

- `new_gsd` is `gsd` with its type component as `ty_opt'` and `initial_value` component as `initial_value'`;

21.22.2 Formally

CONSTANT

$\text{gsd} \stackrel{\text{is}}{=} \{\text{keyword} : \text{keyword}, \text{initial_value} : \text{initial_value}, \text{ty} : \text{ty_opt}, \text{name} : \text{name}\}$

$\text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE$
 $\text{annotate_type_opt}(\text{tenv}, \text{ty_opt}) \xrightarrow{\text{type}} \text{ty_opt}' \quad // \quad \#TE$
 $\text{annotate_expr_opt}(\text{tenv}, \text{initial_value}) \xrightarrow{\text{type}}$
 $\quad (\text{initial_value_type}, \text{initial_value}') \quad // \quad \#TE$
 $\text{annotate_init_type}(\text{tenv}, \text{initial_value_type}, \text{ty_opt}') \xrightarrow{\text{type}} \text{declared_t}$
 $\text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{tenv1} \quad // \quad \#TE$

***** common prefix *****

$\text{keyword} = \text{GDK_Constant}$
 $\text{initial_value}' \stackrel{\text{is}}{=} \langle e \rangle \quad \text{reduce_constants}(\text{tenv1}, e) \xrightarrow{\text{type}} v \quad // \quad \#TE$
 $\text{tenv2} := (G^{\text{tenv1}}.\text{constant_values}[\text{name} \mapsto v], L^{\text{tenv1}})$
 $\text{new_gsd} := \left\{ \begin{array}{ll} \text{keyword} & : \text{keyword}, \\ \text{initial_value} & : \text{initial_value}, \\ \text{ty} & : \text{ty_opt}', \\ \text{name} & : \text{name} \end{array} \right\}$

$\text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\overbrace{\text{tenv2}}^{\text{new_tenv}}, \text{new_gsd})$

NON_CONSTANT

$$\begin{array}{c}
 \text{gsd} \stackrel{\text{is}}{=} \{ \text{keyword} : \text{keyword}, \text{initial_value} : \text{initial_value}, \text{ty} : \text{ty_opt}, \text{name} : \text{name} \} \\
 \text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{annotate_type_opt}(\text{tenv}, \text{ty_opt}) \xrightarrow{\text{type}} \text{ty_opt}' \quad // \quad \#TE \\
 \text{annotate_expr_opt}(\text{tenv}, \text{initial_value}) \xrightarrow{\text{type}} \\
 (\text{initial_value.type}, \text{initial_value}') \quad // \quad \#TE \\
 \text{annotate_init_type}(\text{tenv}, \text{initial_value.type}, \text{ty_opt}') \xrightarrow{\text{type}} \text{declared_t} \\
 \text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{tenv1} \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \text{keyword} \neq \text{GDK_Constant} \quad \text{new_gsd} := \left\{ \begin{array}{l} \text{keyword} : \text{keyword}, \\ \text{initial_value} : \text{initial_value}, \\ \text{ty} : \text{ty_opt}', \\ \text{name} : \text{name} \end{array} \right\} \\
 \hline
 \text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\overbrace{\text{tenv1}}^{\text{new_tenv}}, \text{new_gsd})
 \end{array}$$

21.23 TypingRule.AnnotateTypeOpt

The function

$$\text{annotate_type_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the type t inside an **optional** ty_opt , if there is one, and leaves it as is if ty_opt is **None**. Otherwise, the result is a type error.

21.23.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * ty_opt is **None**;
 - * $\text{ty_opt}'$ is ty_opt .
- All of the following apply (SOME):
 - * ty_opt contains the type t ;
 - * annotating t in tenv yields $t1$ $// \#TE$;
 - * $\text{ty_opt}'$ is $\langle t1 \rangle$.

21.23.2 Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{annotate_type_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\text{ty_opt}}^{\text{ty_opt}'} \\
 \\
 \text{SOME} \\
 \frac{\text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \text{ // } \#TE}{\text{annotate_type_opt}(\text{tenv}, \overbrace{\langle t \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle t1 \rangle}^{\text{ty_opt}'}}
 \end{array}$$

21.24 TypingRule.AnnotateExprOpt

The function

$$\text{annotate_expr_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle e \rangle}^{\text{expr_opt}}) \longrightarrow \overbrace{(\langle \text{expr} \rangle \times \langle \text{ty} \rangle)}^{\text{res}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the **optional** expression **expr_opt** in **tenv** and returns a pair of **optional** expressions for the type and annotated expression in **res**. Otherwise, the result is a type error.

21.24.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * **expr_opt** is **None**;
 - * **res** is **(None, None)**.
- All of the following apply (SOME):
 - * **expr_opt** contains the expression **e**;
 - * annotating **e** in **tenv** yields **(t, e')** // **#TE**;
 - * **res** is **(⟨t⟩, ⟨e'⟩)**.

21.24.2 Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{annotate_expr_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{expr_opt}}) \xrightarrow{\text{type}} (\text{None}, \text{None}) \\
 \\
 \text{SOME} \\
 \frac{\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \text{ // } \#TE}{\text{annotate_expr_opt}(\text{tenv}, \overbrace{\langle e \rangle}^{\text{expr_opt}}) \xrightarrow{\text{type}} \overbrace{(\langle t \rangle, \langle e' \rangle)}^{\text{res}}}
 \end{array}$$

21.25 TypingRule.AnnotateInitType

The function

$$\text{annotate_init_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{ty} \rangle}^{\text{initial_value_type}}, \overbrace{\langle \text{ty} \rangle}^{\text{type_annotation}}) \longrightarrow \overbrace{\text{ty}}^{\text{declared_type}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

takes the [optional](#) type associated with the initialization value of a global storage declaration — `initial_value_type` — and the [optional](#) type annotation for the same global storage declaration — `type_annotation` — and chooses the type that should be associated with the declaration — [declared_type](#) — in `tenv`. Otherwise, the result is a type error.

The ASL parser ensures that at least one of `initial_value_type` and `type_annotation` should not be [None](#).

21.25.1 Prose

One of the following applies:

- All of the following apply (BOTH):
 - * `initial_value_type` is $\langle t1 \rangle$ and `type_annotation` is $\langle t2 \rangle$;
 - * checking that `t1` [type-satisfies](#) `t2` in `tenv` yields [TRUE](#) // [#TE](#);
 - * [declared_type](#) is `t1`.
- All of the following apply (ANNOTATED):
 - * `initial_value_type` is [None](#) and `type_annotation` is $\langle t2 \rangle$;
 - * [declared_type](#) is `t2`.
- All of the following apply (INITIAL):
 - * `initial_value_type` is $\langle t1 \rangle$ and `type_annotation` is [None](#);
 - * [declared_type](#) is `t1`.

BOTH

$$\frac{\text{checked_typesat}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{annotate_init_type}(\text{tenv}, \overbrace{\langle t1 \rangle}^{\text{initial_value_type}}, \overbrace{\langle t2 \rangle}^{\text{type_annotation}}) \xrightarrow{\text{type}} t2}$$

ANNOTATED

$$\text{annotate_init_type}(\text{tenv}, \overbrace{\text{None}}^{\text{initial_value_type}}, \overbrace{\langle t2 \rangle}^{\text{type_annotation}}) \xrightarrow{\text{type}} t2$$

INITIAL

$$\text{annotate_init_type}(\text{tenv}, \overbrace{\langle t1 \rangle}^{\text{initial_value_type}}, \overbrace{\text{None}}^{\text{type_annotation}}) \xrightarrow{\text{type}} t1$$

21.26 TypingRule.AddGlobalStorage

The function

$$\text{add_global_storage}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{keyword}}^{\text{keyword}}, \overbrace{\text{ty}}^{\text{declared_t}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}, \cup \overbrace{\text{TTypeError}}^{\#TE}$$

modifies the static environment `tenv` by adding a global storage for the identifier `name` with global storage keyword `keyword` and type `declared_types`, resulting in the environment `new_tenv`. Otherwise, the result is a type error.

21.26.1 Prose

All of the following apply:

- checking that `name` is not declared in the global environment of `tenv` yields `TRUE//#TE`;
- `new_tenv` is `tenv` with its `global_storage_types` component updated by binding `name` to `(declared_t, keyword)`.

21.26.2 Formally

$$\frac{\text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#TE}{\text{new_tenv} := (G^{\text{tenv}}.\text{global_storage_types}[\text{name} \mapsto (\text{declared_t}, \text{keyword})], L^{\text{tenv}})} \text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{new_tenv}$$

21.27 TypingRule.DeclareType

The function

$$\text{declare_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\langle (\text{identifier} \times \text{field}^*) \rangle}^{\text{s}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}, \cup \overbrace{\text{TTypeError}}^{\#TE}$$

declares a type named `name` with type `ty` and `optional` additional fields over another type `s` in the static environment `tenv`, resulting in the modified environment `new_tenv`. Otherwise, the result is a type error.

21.27.1 Prose

All of the following apply:

- checking that `name` is not already declared in the global environment of `tenv` yields `TRUE//#TE`;
- annotating the `optional` extra fields `s` for `ty` in `tenv` yields via `annotate_extra_fields` yields the modified environment `tenv1` and type `t1//#TE`;
- annotating `t1` in `tenv1` yields `t2//#TE`;

- `tenv2` is `tenv1` with its `declared_types` component updated by binding `name` to `t2`;
- One of the following applies:
 - * All of the following apply (ENUM):
 - `t2` is an enumeration type with labels `ids`, that is, `T_Enum(ids)`;
 - applying `declare_enum_labels` to `t2` in `tenv2 new_tenv` *#TE*.
 - * All of the following apply (NOT_ENUM):
 - `t2` is not an enumeration type;
 - `new_tenv` is `tenv2`.

21.27.2 Formally

ENUM

$$\begin{array}{c}
 \text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} \text{t2} \text{ // } \text{\#TE} \\
 \text{tenv2} := (G^{\text{tenv1}}.\text{declared_types}[\text{name} \mapsto \text{t2}], L^{\text{tenv1}}) \\
 \hline
 \text{t2} = \text{T_Enum}(\text{ids}) \quad \text{declare_enum_labels}(\text{tenv2}, \text{t2}) \xrightarrow{\text{type}} \text{new_tenv} \text{ // } \text{\#TE} \\
 \text{declare_type}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \text{new_tenv}
 \end{array}$$

NOT_ENUM

$$\begin{array}{c}
 \text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} \text{t2} \text{ // } \text{\#TE} \\
 \text{tenv2} := (G^{\text{tenv1}}.\text{declared_types}[\text{name} \mapsto \text{t2}], L^{\text{tenv1}}) \quad \text{ast_label}(\text{t2}) \neq \text{T_Enum} \\
 \hline
 \text{declare_type}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{tenv2}}^{\text{new_tenv}}
 \end{array}$$

21.28 TypingRule.AnnotateExtraFields

The function

$$\text{annotate_extra_fields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\langle (\overbrace{\text{super}}^{\text{super}} \times \overbrace{\text{field}^*}^{\text{extra_fields}}) \rangle}^{\text{s}}) \longrightarrow \\
 (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{ty}}^{\text{new_ty}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates the type `ty` with the `optional` extra fields `s` in `tenv`, yielding the modified environment `new_tenv` and type `new_ty`. Otherwise, the result is a type error.

21.28.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * `s` is `None`;
 - * `new_tenv` is `tenv`;
 - * `new_ty` is `ty`.
- All of the following apply (EMPTY_FIELDS):
 - * `s` is $\langle(\text{super}, \text{extra_fields})\rangle$;
 - * checking that `ty` *subtype-satisfies* the named type `super` (that is, `T_Named(super)`) yields `TRUE`_{#TE};
 - * `extra_fields` is the empty list;
 - * `new_tenv` is `tenv`;
 - * `new_ty` is `ty`.
- All of the following apply (NO_SUPER):
 - * `s` is $\langle(\text{super}, \text{extra_fields})\rangle$;
 - * checking that `ty` *subtype-satisfies* the named type `super` (that is, `T_Named(super)`) yields `TRUE`_{#TE};
 - * `extra_fields` is not an empty list;
 - * `super` is not bound to a type in `tenv`;
 - * the result is a type error indicating that `super` is not a declared type.
- All of the following apply (STRUCTURED):
 - * `s` is $\langle(\text{super}, \text{extra_fields})\rangle$;
 - * checking that `ty` *subtype-satisfies* the named type `super` (that is, `T_Named(super)`) yields `TRUE`_{#TE};
 - * `extra_fields` is not an empty list;
 - * `super` is bound to a type `t` in `tenv`;
 - * checking that `t` is a *structured type* yields `TRUE` or a type error indicating that a *structured type* was expected, thereby short-circuiting the entire rule;
 - * `t` has AST label `L` and fields `fields`;
 - * `new_ty` is the type with AST label `L` and list fields that is the concatenation of `fields` and `extra_fields`;
 - * `new_tenv` is `tenv` with its *subtypes* component updated by binding `name` to `super`.

21.28.2 Formally

$$\begin{array}{c}
\text{NONE} \\
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\text{None}}^s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ty}}^{\text{new_ty}}) \\
\\
\text{EMPTY_FIELDS} \\
\begin{array}{c}
\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} \text{b} \\
\text{check}(\text{b}, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{//} } \text{\#TE} \\
\text{extra_fields} = []
\end{array} \\
\hline
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\langle (\text{super}, \text{extra_fields}) \rangle}^s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ty}}^{\text{new_ty}}) \\
\\
\text{NO_SUPER} \\
\begin{array}{c}
\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} \text{b} \\
\text{check}(\text{b}, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{//} } \text{\#TE} \\
\text{extra_fields} \neq [] \\
G^{\text{tenv}}.\text{declared_types}(\text{super}) = \perp
\end{array} \\
\hline
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\langle (\text{super}, \text{extra_fields}) \rangle}^s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI}) \\
\\
\text{STRUCTURED} \\
\begin{array}{c}
\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} \text{b} \\
\text{check}(\text{b}, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{//} } \text{\#TE} \\
\text{extra_fields} \neq [] \\
G^{\text{tenv}}.\text{declared_types}(\text{super}) = \text{t} \\
\text{check}(\text{ast_label}(\text{t}) \in \{\text{T_Record}, \text{T_Exception}\}, \text{ExpectedStructuredType}) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{//} } \text{\#TE} \\
\text{t} \stackrel{\text{is}}{=} L(\text{fields}) \quad \text{new_ty} := L(\text{fields} + \text{extra_fields}) \\
\text{new_tenv} := (G^{\text{tenv}}.\text{subtypes}[\text{name} \mapsto \text{super}], L^{\text{tenv}})
\end{array} \\
\hline
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\langle (\text{super}, \text{extra_fields}) \rangle}^s) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ty})
\end{array}$$

21.29 TypingRule.AnnotateEnumLabels

The function

$$\text{declare_enum_labels}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{identifier}^+}^{\text{ids}}, \longrightarrow \overbrace{\text{SE} \cup \text{\#TE}}^{\text{new_tenv}})$$

updates the static environment `tenv` with the identifiers `ids` listed by an enumeration type, yielding the modified environment `new_tenv`. Otherwise, the result is a type error.

21.29.1 Prose

All of the following apply:

- `ids` is the (non-empty) list of labels `id1..k`;
- `tenv0` is `tenv`;
- declaring the constant `idi` with the type `T_Named(name)` and literal `L_Int(i-1)` in `tenvi-1` via `declare_const` yields `tenvi`, for $i = 1$ to k (if $k > 1$) *// #TE*;
- `new_tenv` is `tenvk`.

21.29.2 Formally

$$\frac{\begin{array}{c} \text{ids} \stackrel{\text{is}}{=} \text{id}_{1..k} \quad \text{tenv}_0 := \text{tenv} \\ i = 1..k : \text{declare_const}(\text{tenv}_{i-1}, \text{id}_i, \text{T_Named}(\text{name}), \text{L_Int}(i-1)) \xrightarrow{\text{type}} \text{tenv}_i \quad \text{// \#TE} \end{array}}{\text{declare_enum_labels}(\text{tenv}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \overbrace{\text{tenv}_k}^{\text{new_tenv}}}$$

21.30 TypingRule.DeclareConst

The function

$$\text{declare_const}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{literal}}^{\text{vv}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

adds a constant given by the identifier `name`, type `ty`, and literal `v` to the static environment `tenv`, yielding the modified environment `new_tenv`. Otherwise, the result is a type error.

21.30.1 Prose

All of the following apply:

- adding the global storage given by the identifier `name`, global declaration keyword `GDK_Constant`, and type `ty` to `tenv` yields `tenv1`;
- `new_tenv` is `tenv1` with its `constant_values` component updated by binding `name` to `v`.

$$\frac{\begin{array}{c} \text{add_global_storage}(\text{tenv}, \text{name}, \text{GDK_Constant}, \text{ty}) \xrightarrow{\text{type}} \text{tenv1} \\ \text{new_tenv} := (G^{\text{tenv1}}.\text{constant_values}[\text{name}, \text{v}], L^{\text{tenv1}}) \end{array}}{\text{declare_const}(\text{tenv}, \text{name}, \text{ty}, \text{v}) \xrightarrow{\text{type}} \text{new_tenv}}$$

Chapter 22

Typing of Specifications

The untyped AST of an ASL specification consists of a list of declarations. Type-checking the untyped AST succeeds if all declarations can be successfully annotated. This is achieved via the function *type_check_ast*, which is detailed in `TypingRule.TypeCheckAST` (see Section 22.1).

We define the following helper rules:

- `TypingRule.AnnotateDeclComps` (see Section 22.3)
- `TypingRule.BuildDependencies` (see Section 22.4)
- `TypingRule.DeclDependencies` (see Section 22.5)
- `TypingRule.TypeCheckMutuallyRec` (see Section 22.6)
- `TypingRule.FoldEnvAndFs` (see Section 22.7)
- `TypingRule.DefDecl` (see Section 22.8)
- `TypingRule.DefEnumLabels` (see Section 22.9)
- `TypingRule.UseDecl` (see Section 22.10)
- `TypingRule.UseTy` (see Section 22.11)
- `TypingRule.UseSubtypes` (see Section 22.12)
- `TypingRule.UseExpr` (see Section 22.13)
- `TypingRule.UseLexpr` (see Section 22.14)
- `TypingRule.UsePattern` (see Section 22.15)
- `TypingRule.UseSlice` (see Section 22.16)
- `TypingRule.UseBitfield` (see Section 22.17)

- `TypingRule.UseConstraint` (see Section 22.18)
- `TypingRule.UseStmt` (see Section 22.19)
- `TypingRule.UseLDI` (see Section 22.20)
- `TypingRule.UseCase` (see Section 22.21)
- `TypingRule.UseCatcher` (see Section 22.22)

22.1 TypingRule.TypeCheckAST

The relation

$$\text{type_check_ast}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}^*}^{\text{decls}}) \times (\overbrace{\text{decl}^*}^{\text{new_decls}} \times \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a list of declarations `decls` in an input static environment `tenv`, yielding an output static environment `new_tenv` and annotated list of declarations `new_decls`. Otherwise, the result is a type error.

Definition 30 (Strongly Connected Components) *Given a graph $G = (V, E)$, a subset of its nodes $C \subseteq V$ is called a strongly connected component of G if every pair of nodes $u, v \in C$ reachable from one another.*

The strongly connected components of a graph (V, E) uniquely partitions its set of nodes V into a set of strongly connected components:

$$\text{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C. (u, v), (v, u) \in E^*\}.$$

We write E^ to denote the reflexive-transitive closure of E .*

Definition 31 (Topological Ordering) *For a graph $G = (V, E)$ and its strongly connected components $\text{comps} \triangleq \text{SCC}(V, E)$, we say that $C_1 \in \text{comps}$ is ordered before $C_2 \in \text{comps}$, denoted $C_1 < C_2$, if the following condition holds:*

$$C_1 < C_2 \Leftrightarrow \exists c_1 \in C_1. c_2 \in C_2. (c_1, c_2) \in E^*.$$

This ordering is not total. That is, there may exist strongly connected components $A, B \in \text{comps}$ such that $A \not< B$ and $B \not< A$.

We say that a list of subsets of V — `comps2` — respects the topological ordering of `comps` if each element of `comps2` is a member of `comps` and for every $C_1, C_2 \in \text{comps}$ such that $C_1 < C_2$ we have that C_1 appears before C_2 in `comps2`. We denote this as $\text{comps2} \in \text{topological_ordering}(V, E, \text{comps})$.

22.1.1 Prose

All of the following apply:

- applying *build_dependencies* to `decls` yields the dependency graph `(defs, depends)`;
- partitioning the nodes of the dependency graph `(defs, depends)` into strongly connected components yields `comps`;
- `comps2` is an ordering of `comps` that respects the topological ordering induced by `depends`;
- `comp_decls` applies *decls_of_comp* to each component `c` in `comps2` to transform it into a list, yielding a list of lists where each sublist corresponds to one strongly connected component;
- applying *annotate_decl_comps* to `comp_decls` in `tenv` yields `(new_decls, new_tenv)` *#TE*.

22.1.2 Formally

$$\begin{array}{c}
 \text{build_dependencies}(\text{decls}) \xrightarrow{\text{type}} (\text{defs}, \text{depends}) \\
 \text{SCC}(\text{defs}, \text{depends}) = \text{comps1} \quad \text{comps2} \in \text{topological_ordering}(\text{comps1}, \text{depends}) \\
 \text{comp_decls} := [c \in \text{comps2} : \text{decls_of_comp}(c, \text{decls})] \\
 \text{annotate_decl_comps}(\text{tenv}, \text{comp_decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv}) \quad // \quad \text{\#TE} \\
 \hline
 \text{type_check_ast}(\text{tenv}, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv})
 \end{array}$$

22.1.3 Comments

It is crucial to process the strongly connected components obtained from the dependency graph according to the topological ordering of the components. Otherwise, the type-system could falsely result in a type error indicating that some identifier is not defined. However, a topological ordering is not unique, which is why *type_check_ast* is a relation rather than a function. It is possible to obtain a deterministic ordering by ordering components so as to respect the order of declarations in `decls`, that is, the order in which declarations appear in the specification. Similarly, any ordering of the declarations within a single strongly connected component is correct, but it is possible to order the declarations according to their order of appearance in `decls`, as demonstrated in `TypingRule.DeclsOfComp`.

22.2 TypingRule.DeclsOfComp

The helper function

$$\text{decls_of_comp}(\overbrace{\mathcal{P}(\text{identifier})}^{\text{comp}}, \overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{comp_decls}}$$

lists the sublist of declarations in `decls` corresponding to the identifiers in `comp` yielding `comp.decls`

22.2.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `decls` is the empty list;
 - * define `comp.decls` as the empty list.
- All of the following apply (NON_EMPTY):
 - * `decls` is the list with **head** `d` and **tail** `declsone`;
 - * define `decls2` as the singleton list for `d` if applying *def-decl* to `d` yields an identifier that is a member of `comp` and the empty list, otherwise;
 - * applying *decls-of-comp* to `comp` and `decls1` yields `decls3`;
 - * define `comp.decls` as the concatenation of `decls2` and `decls3`.

22.2.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{decls_of_comp}(\text{comp}, \overbrace{[]^{\text{decls}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{comp.decls}}} \\
 \\
 \text{NON_EMPTY} \\
 \text{decls2} := \text{choice}(\text{def_decl}(d) \in \text{comp}, [d], []) \\
 \text{decls_of_comp}(\text{comp}, \text{decls1}) \xrightarrow{\text{type}} \text{decls3} \\
 \hline
 \text{decls_of_comp}(\text{comp}, \overbrace{[d] + \text{decls1}}^{\text{decls}}) \xrightarrow{\text{type}} \overbrace{\text{decls2} + \text{decls3}}^{\text{comp.decls}}
 \end{array}$$

22.3 TypingRule.AnnotateDeclComps

The function

$$\text{annotate_decl_comps}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{decl}^*)^*}^{\text{comps}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{decl}^*}^{\text{new_decls}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of declaration components `comps` (a list of lists) in the static environment `tenv`, yielding the annotated list of declarations `new_decls` and modified environment `new_tenv`. Otherwise, the result is a type error.

We note that a strongly-connected component containing just a single declaration may contain any kind of global declaration — a type declaration, a global storage declaration, or a subprogram declaration — whereas a strongly-connected component containing multiple declarations must be checked to contain only subprograms. This is because the

only type of mutually-recursive declarations allowed in ASL are between subprograms. The rules below handle these cases separately (SINGLE for single declarations and MUTUALLY_RECURSIVE for more than one declaration).

22.3.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `comps` is the empty list;
 - * define `new_tenv` as `tenv`;
 - * define `new_decls` as the empty list.
- All of the following apply (SINGLE):
 - * `comps` is a list with `head` `comp` and `tail` `comps1`;
 - * `comp` is a single declaration `d`;
 - * applying `typecheck_decl` to `d` in `tenv` yields `(d1, tenv1)`^{#TE};
 - * applying `annotate_decl_comps` to `comps1` in `tenv1` yields `(new_tenv, decls1)`^{#TE};
 - * define `new_decls` as the list with `head` `d1` and `tail` `decls1`.
- All of the following apply (MUTUALLY_RECURSIVE):
 - * `comps` is a list with `head` `comp` and `tail` `comps1`;
 - * `comp` is a list with more than one declaration (which together represent a mutually-recursive list of declarations);
 - * applying `type_check_mutually_rec` to `comp` in `tenv` yields `(decls1, tenv1)`^{#TE};
 - * applying `annotate_decl_comps` to `comps1` in `tenv1` yields `(new_tenv, decls2)`^{#TE};
 - * define `new_decls` as the concatenation of `decls1` and `decls2`.

22.3.2 Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{annotate_decl_comps}(\text{tenv}, \overbrace{[]^{\text{comps}}}) \longrightarrow (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{[]^{\text{new_decls}}}) \\
\\
\text{SINGLE} \\
\begin{array}{c}
\text{comp} = [\text{d}] \quad \text{typecheck_decl}(\text{tenv}, \text{d}) \xrightarrow{\text{type}} (\text{d1}, \text{tenv1}) \text{ // \#TE} \\
\text{annotate_decl_comps}(\text{tenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{decls1}) \text{ // \#TE} \\
\hline
\text{annotate_decl_comps}(\text{tenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_tenv}, \overbrace{[\text{d1}] + \text{decls1}}^{\text{new_decls}})
\end{array} \\
\\
\text{MUTUALLY_RECURSIVE} \\
\begin{array}{c}
|\text{comp}| > 1 \quad \text{type_check_mutually_rec}(\text{tenv}, \text{comp}) \xrightarrow{\text{type}} (\text{decls1}, \text{tenv1}) \text{ // \#TE} \\
\text{annotate_decl_comps}(\text{tenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{decls2}) \text{ // \#TE} \\
\hline
\text{annotate_decl_comps}(\text{tenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_tenv}, \overbrace{\text{decls1} + \text{decls2}}^{\text{new_decls}})
\end{array}
\end{array}$$

22.4 TypingRule.BuildDependencies

The function

$$\text{build_dependencies}(\overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow (\overbrace{\text{identifier}^*}^{\text{defs}}, \overbrace{(\text{identifier} \times \text{identifier})^*}^{\text{depends}})$$

takes a set of declarations **decls** and returns a graph whose set of nodes are the identifiers that are used to name declarations and whose set of edges **depends** consists of pairs (a, b) where the declaration of a uses an identifier defined by b . We refer to this graph as the *dependency graph* (of **decls**).

22.4.1 Prose

All of the following apply:

- define **defs** as the union of two sets:
 1. the set of identifiers obtained by applying *def_decl* to each declaration in **decls**;
 2. the union of applying *def_enum_labels* to each declaration in **decls**.
- define **depends** as the union of applying *decl_dependencies* to each declaration in **decls**.

22.4.2 Formally

$$\begin{aligned}
 \text{defs} &:= \{ \text{def_decl}(d) \mid d \in \text{decls} \} \cup \bigcup_{d \in \text{decls}} \text{def_enum_labels}(d) \\
 \text{depends} &:= \bigcup_{d \in \text{decls}} \text{decl_dependencies}(d) \\
 \hline
 &\text{build_dependencies}(\text{decls}) \xrightarrow{\text{type}} (\text{ids}, \text{depends})
 \end{aligned}$$

22.5 TypingRule.DeclDependencies

The function

$$\text{decl_dependencies}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{(\text{identifier} \times \text{identifier})^*}^{\text{depends}}$$

returns the set of dependent pairs of identifiers **depends** induced by the declaration **d**.

22.5.1 Prose

Define **depends** as the union of the following two sets:

1. a pair (**id1**, **id2**) where **id1** is the result of applying *def_decl* to **d** and **id2** included in the result of applying *def_enum_labels* to **d**; and
2. a pair (**id1**, **id2**) where **id1** is the result of applying *def_decl* to **d** and **id2** included in the result of applying *use_decl* to **d**; and

22.5.2 Formally

$$\begin{aligned}
 \text{depends} &:= \{ (\text{id1}, \text{id2}) \mid \text{id1} = \text{def_decl}(d) \wedge \text{id2} \in \text{def_enum_labels}(d) \} \cup \\
 &\quad \{ (\text{id1}, \text{id2}) \mid \text{id1} = \text{def_decl}(d) \wedge \text{id2} \in \text{use_decl}(d) \} \\
 \hline
 &\text{decl_dependencies}(d) \xrightarrow{\text{type}} \text{depends}
 \end{aligned}$$

22.6 TypingRule.TypeCheckMutuallyRec

The function

$$\text{type_check_mutually_rec}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow (\overbrace{\text{decl}^*}^{\text{new_decls}} \times \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of mutually recursive declarations **decls** in the static environment **tenv**, yielding the annotated list of subprogram declarations **new_decls** and modified environment **new_tenv**.

One of the requirements from an ASL specification is that each setter has a corresponding getter. To facilitate checking this requirement, the type-system annotates the declarations of all subprograms that are not setters before annotating the declarations of setters. This way, when annotating a setter, the corresponding getter should have already been annotated and added to the environment, making it easy to check this requirement.

22.6.1 Prose

All of the following apply:

- checking that each declaration in d is a subprogram declaration yields $\text{TRUE} \# \text{TE}$;
- applying annotate_func_sig to each node f in tenv , where $D_Func(f)$ is a declaration in decls , yields $(\text{tenv}_f, d_f) \# \text{TE}$;
- define env_and_fs as the list of pairs, each consisting of the local environment component of tenv_f and the annotated subprogram d_f , for each subprogram declaration $D_Func(f)$ in decls ;
- splitting env_and_fs into two sublists by testing each pair to check whether the subprogram declaration component is that of a setter (or an empty setter) yields setters and others , respectively;
- define env_and_fs1 as the concatenation of others and setters ;
- applying fold_env_and_fs to the global component of tenv and env_and_fs1 yields $(\text{genv}, \text{env_and_fs2}) \# \text{TE}$;
- for each pair consisting of a local static environment and subprogram declaration $(\text{lenv}, D_Func(f))$, applying $\text{annotate_subprogram}()$ to the static environment $(\text{genv}, \text{lenv})$ and f yields $\text{new_d}_f \# \text{TE}$;
- define new_decls as the list of subprogram declarations $D_Func(\text{new_d}_f)$, for each pair $(_, D_Func(f))$ in env_and_fs2 ;
- define new_tenv as the static environment $(\text{genv}, L^{\text{tenv}})$.

22.6.2 Formally

$$\begin{array}{l}
 d \in \text{decls} : \text{check}(\text{ast_label}(d) = D_Func, \text{TE_BRA}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\
 D_Func(f) \in \text{decls} : \text{annotate_func_sig}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{tenv}_f, d_f) \# \text{TE} \\
 \text{env_and_fs} := [D_Func(f) \in \text{decls} : (L^{\text{tenv}_f}, d_f)] \\
 \text{setters} := \left[\begin{array}{l} (\text{lenv}, d) \mid (\text{lenv}, D_Func(f)) \in \text{env_and_fs} \wedge \\ \text{ast_label}(f) \in \{\text{ST_Setter}, \text{ST_EmptySetter}\} \end{array} \right] \\
 \text{others} := \left[\begin{array}{l} (\text{lenv}, d) \mid (\text{lenv}, D_Func(f)) \in \text{env_and_fs} \wedge \\ \text{ast_label}(f) \notin \{\text{ST_Setter}, \text{ST_EmptySetter}\} \end{array} \right] \\
 \text{env_and_fs1} := \text{others} + \text{setters} \\
 \text{fold_env_and_fs}(G^{\text{tenv}}, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{genv}, \text{env_and_fs2}) \# \text{TE} \\
 (\text{lenv}, D_Func(f)) \in \text{env_and_fs2} : \text{annotate_subprogram}((\text{genv}, \text{lenv}), f) \xrightarrow{\text{type}} \\
 \text{new_d}_f \# \text{TE} \\
 \text{new_decls} := [(_, D_Func(f)) \in \text{env_and_fs2} : D_Func(\text{new_d}_f)] \\
 \hline
 \text{type_check_mutually_rec}(\text{tenv}, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \overbrace{(\text{genv}, L^{\text{tenv}})}^{\text{new_tenv}})
 \end{array}$$

22.7 TypingRule.FoldEnvAndFs

The function

$$\text{fold_env_and_fs}(\overbrace{\mathbb{G}}^{\text{genv}}, \overbrace{(\mathbb{L} \times \text{func})^*}^{\text{env_and_fs}}) \longrightarrow \overbrace{\mathbb{G}}^{\text{new_genv}} \times \overbrace{(\mathbb{L} \times \text{func})^*}^{\text{new_env_and_fs}} \cup \overbrace{\text{TypeError}}^{\text{\#TE}}$$

processes a list of pairs, each consisting of a local static environment and a subprogram declaration, `env_and_fs`, in the context of a global static environment `genv`. The result is a modified global static environment `new_genv` and list of pairs `new_env_and_fs`.

22.7.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `env_and_fs` is the empty list;
 - * define `new_genv` as `genv`;
 - * define `new_env_and_fs` as the empty list.
- All of the following apply (NON_EMPTY):
 - * `env_and_fs` is the list with `head` (`lenv, f`) and `tail` `env_and_fs1`;
 - * define `tenv` as the environment where the global environment component is `genv` and the local environment component is `lenv`;
 - * applying `declare_one_func` to `f` in `tenv` yields (`tenv1, f1`) `// \#TE`;
 - * applying `fold_env_and_fs` to the global environment of `tenv1` and `env_and_fs1` yields (`new_genv, env_and_fs2`) `// \#TE`;
 - * define `new_env_and_fs` as the list with `head` (`lenv, f1`) and `tail` `env_and_fs2`.

22.7.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{fold_env_and_fs}(\overbrace{\mathbb{G}}^{\text{genv}}, \overbrace{[]^{\text{env_and_fs}}}^{\text{env_and_fs}}) \xrightarrow{\text{type}} (\overbrace{\mathbb{G}}^{\text{new_genv}}, \overbrace{[]^{\text{new_env_and_fs}}}^{\text{new_env_and_fs}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{tenv} := (\text{genv}, \text{lenv}) \quad \text{declare_one_func}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{tenv1}, f1) \quad // \quad \text{\#TE} \\
 \text{fold_env_and_fs}(G^{\text{tenv1}}, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{new_genv}, \text{env_and_fs2}) \quad // \quad \text{\#TE} \\
 \text{new_env_and_fs} := [(\text{lenv}, f1)] + \text{env_and_fs2}
 \end{array} \\
 \hline
 \text{fold_env_and_fs}(\overbrace{\mathbb{G}}^{\text{genv}}, \overbrace{[(\text{lenv}, f)] + \text{env_and_fs1}}^{\text{env_and_fs}}) \xrightarrow{\text{type}} (\overbrace{\mathbb{G}}^{\text{new_genv}}, \overbrace{\text{new_env_and_fs}}^{\text{new_env_and_fs}})
 \end{array}$$

22.8 TypingRule.DefDecl

The function

$$\text{def_decl}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\text{identifier}}^{\text{name}}$$

returns the identifier **name** being defined by the declaration **d**.

22.8.1 Prose

One of the following applies:

- All of the following apply (D_FUNC):
 - * **d** declares a subprogram for the identifier **name**.
- All of the following apply (D_GLOBALSTORAGE):
 - * **d** declares a global storage element for the identifier **name**.
- All of the following apply (D_TYPEDECL):
 - * **d** declares a type for the identifier **name**.

D_FUNC

$$\text{def_decl}(\overbrace{\text{D_Func}(\text{name} : \text{name}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \text{name}$$

D_GLOBALSTORAGE

$$\text{def_decl}(\overbrace{\text{D_GlobalStorage}(\text{name} : \text{name}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \text{name}$$

D_TYPEDECL

$$\text{def_decl}(\overbrace{\text{D_TypeDecl}(\text{name}, _, _)}^{\text{d}}) \xrightarrow{\text{type}} \text{name}$$

22.9 TypingRule.DefEnumLabels

The function

$$\text{def_enum_labels}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{labels}}$$

takes a declaration **d** and returns the set of enumeration labels it defines — **labels** — if it defines any.

22.9.1 Prose

One of the following applies:

- All of the following apply (DECL_ENUM):
 - * `d` is a declaration of an enumeration type with labels `labels`;
 - * the result is `labels` as a set (rather than a list).
- All of the following apply (OTHER):
 - * `d` is not a declaration of an enumeration type;
 - * define `labels` as the empty set.

$$\begin{array}{c}
 \text{DECL_ENUM} \\
 \hline
 d = \text{D_TypeDecl}(\text{name}, \text{T_Enum}(\text{labels}, _)) \\
 \hline
 \text{def_enum_labels}(d) \xrightarrow{\text{type}} \overbrace{\{\text{labels}\}}^{\text{labels}} \\
 \\
 \text{OTHER} \\
 \hline
 d \neq \text{D_TypeDecl}(\text{name}, \text{T_Enum}(\text{labels}, _)) \\
 \hline
 \text{def_enum_labels}(d) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{labels}}
 \end{array}$$

22.10 TypingRule.UseDecl

The function

$$\text{use_decl}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers `ids` which the declaration `d` depends on.

22.10.1 Prose

One of the following applies:

- All of the following apply (D_TYPEDECL):
 - * `d` declares a type `ty` and fields `fields`, that is, `D_TypeDecl(_, ty, fields)` (the first component is the name, which is being defined);
 - * define `ids` as the union of applying `use_ty` to `ty` and applying `use_subtypes` to `fields`.
- All of the following apply (D_GLOBALSTORAGE):
 - * `d` declares a global storage element with initial value `initial_value` and type `ty`;

- * define `ids` as the union of applying `use_e` to `initial_value` and applying `use_ty` to `ty`.
- All of the following apply (D_FUNC):
 - * `d` declares a subprogram with arguments `args`, `optional` return type `ret_ty_opt`, parameters `params`, and body statement `body`;
 - * define `ids` as the union of applying `use_ty` to each type of an argument in `args`, applying `use_ty` to `ret_ty_opt`, applying `use_ty` to each type of a parameter in `params`, and applying `use_e` to `body`.

22.10.2 Formally

$$\begin{array}{c}
 \text{D_TYPEDECL} \\
 \text{use_decl}(\overbrace{\text{D_TypeDecl}(_, \text{ty}, \text{fields})}^d) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \text{use_subtypes}(\text{fields})}^{\text{ids}} \\
 \\
 \text{D_GLOBALSTORAGE} \\
 \frac{\text{ids} := \text{use_e}(\text{initial_value}) \cup \text{use_ty}(\text{ty})}{\text{use_decl}(\overbrace{\text{D_GlobalStorage}(\{\text{initial_value} : \text{initial_value}, \text{ty} : \text{ty} \dots\})}^d) \xrightarrow{\text{type}} \text{ids}} \\
 \\
 \text{D_FUNC} \\
 \text{ids} := \begin{array}{l} \{(_, t) \in \text{use_ty}(t) : \text{id}\} \cup \\ \text{use_ty}(\text{ret_ty_opt}) \cup \\ \{(_, t) \in \text{params} : \text{use_ty}(t)\} \cup \\ \text{use_e}(\text{body}) \end{array} \\
 \hline
 \text{use_decl} \left(\text{D_Func} \left(\overbrace{\left(\begin{array}{l} \text{body} : \text{body}, \\ \text{args} : \text{args}, \\ \text{return_type} : \text{ret_ty_opt}, \\ \text{parameters} : \text{params}, \\ \dots \end{array} \right)}^d \right) \right) \xrightarrow{\text{type}} \text{ids}
 \end{array}$$

22.11 TypingRule.UseTy

The function

$$\text{use_ty}(\overbrace{\text{ty} \cup \langle \text{ty} \rangle}^t) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers `ids` which the type or `optional` type `t` depends on.

22.11.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * \mathbf{t} is `None`;
 - * define `ids` as \emptyset .
- All of the following apply (SOME):
 - * \mathbf{t} is $\langle \mathbf{ty} \rangle$;
 - * applying `use_ty` to \mathbf{ty} yields `ids`.
- All of the following apply (SIMPLE):
 - * \mathbf{t} is one of the following types: enumeration, Boolean, real, or string;
 - * define `ids` as the empty set.
- All of the following apply (T_NAMED):
 - * \mathbf{t} is the named type for \mathbf{s} ;
 - * define `ids` as the singleton set for \mathbf{s} .
- All of the following apply (INT_NO_CONSTRAINTS):
 - * \mathbf{t} is either the unconstrained integer type or a `parameterized integer type`;
 - * define `ids` as the empty set.
- All of the following apply (INT_WELL_CONSTRAINED):
 - * \mathbf{t} is the well-constrained integer type with constraints `vcs`;
 - * define `ids` as the union of applying `use_constraint` to each constraint in `vcs`.
- All of the following apply (T_TUPLE):
 - * \mathbf{t} is the tuple type with list of types `li`;
 - * define `ids` as the union of applying `use_constraint` to each constraint in `vcs`.
- All of the following apply (STRUCTURED):
 - * \mathbf{t} is a `structured type` with fields `fields`;
 - * define `ids` as the union of applying `use_ty` to each field type in `fields`.
- All of the following apply (ARRAY_EXPR):
 - * \mathbf{t} is an array expression with length expression `e` and element type \mathbf{t}' ;
 - * define `ids` as the union of applying `use_e` to `e` and applying `use_ty` to \mathbf{t}' .
- All of the following apply (ARRAY_ENUM):
 - * \mathbf{t} is an array expression with enumeration type \mathbf{s} and element type \mathbf{t}' ;

- * define ids as the union of the singleton set for \mathbf{s} and applying use_ty to \mathbf{t}' .
- All of the following apply (T_BITS):
 - * \mathbf{t} is a bitvector type with width expression \mathbf{e} and bitfields bitfields ;
 - * define ids as the union of applying use_e to \mathbf{e} and applying use_bitfield to each field in bitfields .

22.11.2 Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{use_ty}(\overbrace{\text{None}}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SOME} \\
 \frac{\text{use_ty}(\text{ty}) \xrightarrow{\text{type}} \text{ids}}{\text{use_ty}(\overbrace{\langle \text{ty} \rangle}^{\mathbf{t}}) \xrightarrow{\text{type}} \text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{SIMPLE} \\
 \frac{\text{ast_label}(\mathbf{t}) \in \{\text{T_Enum}, \text{T_Bool}, \text{T_Real}, \text{T_String}\}}{\text{use_ty}(\mathbf{t}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{T_NAMED} \\
 \text{use_ty}(\overbrace{\text{T_Named}(\mathbf{s})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\{\mathbf{s}\}}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{INT_NO_CONSTRAINTS} \\
 \frac{\text{ast_label}(\mathbf{c}) \in \{\text{Unconstrained}, \text{Parameterized}\}}{\text{use_ty}(\overbrace{\text{T_Int}(\mathbf{c})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
 \end{array}$$

$$\begin{array}{c}
 \text{INT_WELL_CONSTRAINED} \\
 \text{use_ty}(\overbrace{\text{T_Int}(\text{WellConstrained}(\mathbf{vcs}))}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\mathbf{c} \in \mathbf{vcs}} \text{use_constraint}(\mathbf{c})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{T_TUPLE} \\
 \text{use_ty}(\overbrace{\text{T_Tuple}(\mathbf{li})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\mathbf{t} \in \mathbf{li}} \text{use_ty}(\mathbf{t})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{STRUCTURED} \\
 \frac{L \in \{\text{T_Record}, \text{T_Exception}\}}{\text{use_ty}(\overbrace{L(\text{fields})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{(_, \mathbf{t}) \in \text{fields}} \text{use_ty}(\mathbf{t})}^{\text{ids}}}
 \end{array}$$

$$\begin{array}{c}
 \text{ARRAY_EXPR} \\
 \text{use_ty}(\overbrace{\text{T_Array}(\text{ArrayLength.Expr}(\mathbf{e}), \mathbf{t}')}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\mathbf{e}) \cup \text{use_ty}(\mathbf{t}')}^{\text{ids}}
 \end{array}$$

$$\text{ARRAY_ENUM} \quad \text{use_ty}(\overbrace{\text{T_Array}(\text{ArrayLength_Enum}(s, _), t')}^t) \xrightarrow{\text{type}} \overbrace{\{s\} \cup \text{use_ty}(t')}^{\text{ids}}$$

$$\text{T_BITS} \quad \text{use_ty}(\overbrace{\text{T_Bits}(e, \text{bitfields})}^t) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \bigcup_{f \in \text{bitfields}} \text{use_bitfield}(f)}^{\text{ids}}$$

22.12 TypingRule.UseSubtypes

The function

$$\text{use_subtypes}(\overbrace{\langle \langle \overbrace{\text{identifier} \times \text{field}^*}^x, \text{subfields} \rangle \rangle}^{\text{fields}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the **optional** pair consisting of identifier **x** (the type being subtyped) and fields **subfields** depends on.

22.12.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * **fields** is **None**;
 - * define **ids** as the empty set.
- All of the following apply (SOME):
 - * **fields** is $\langle \langle x, \text{subfields} \rangle \rangle$;
 - * define **ids** as the union of the singleton set for **x** and the union of applying **use_ty** to each field type in **subfields**.

22.12.2 Formally

$$\begin{array}{c} \text{NONE} \\ \text{use_subtypes}(\text{None}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \end{array} \quad \begin{array}{c} \text{SOME} \\ \text{ids} := \{x\} \cup \bigcup_{(_, t) \text{ use_ty}(t)} \\ \hline \text{use_subtypes}(\langle \langle x, \text{subfields} \rangle \rangle) \xrightarrow{\text{type}} \text{ids} \end{array}$$

22.13 TypingRule.UseExpr

The function

$$\text{use_e}(\overbrace{\langle \text{expr} \rangle}^{\mathbf{e}} \cup \overbrace{\langle \text{expr} \rangle}^{\mathbf{e}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\mathbf{ids}}$$

returns the set of identifiers \mathbf{ids} which the expression or **optional** expression \mathbf{e} depends on.

22.13.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * \mathbf{e} is **None**;
 - * define \mathbf{ids} as the empty set.
- All of the following apply (SOME):
 - * \mathbf{e} is $\langle \mathbf{e1} \rangle$;
 - * applying *use_e* to $\mathbf{e1}$ yields \mathbf{ids} .
- All of the following apply (E_LITERAL):
 - * \mathbf{e} is a literal expression;
 - * define \mathbf{ids} as the empty set.
- All of the following apply (E_ATC):
 - * \mathbf{e} is the typing assertion for expression \mathbf{e} and type \mathbf{ty} ;
 - * define \mathbf{ids} as the union of applying *use_e* to $\mathbf{e1}$ and applying *use_ty* to \mathbf{ty} .
- All of the following apply (E_VAR):
 - * \mathbf{e} is the variable expression for identifier \mathbf{x} ;
 - * define \mathbf{ids} as the singleton set for \mathbf{x} .
- All of the following apply (E_GETARRAY):
 - * \mathbf{e} is the **array access** expression for base expression $\mathbf{e1}$ and index expression $\mathbf{e2}$;
 - * define \mathbf{ids} as the union of applying *use_e* to $\mathbf{e1}$ and applying *use_e* to $\mathbf{e2}$.
- All of the following apply (E_BINOP):
 - * \mathbf{e} is the binary operation expression over expressions $\mathbf{e1}$ and $\mathbf{e2}$;
 - * define \mathbf{ids} as the union of applying *use_e* to $\mathbf{e1}$ and applying *use_e* to $\mathbf{e2}$.

- All of the following apply (E_UNOP):
 - * **e** is the unary operation expression over any unary operation and an expression **e1**;
 - * define **ids** as the union of applying *use_e* to **e1**.
- All of the following apply (E_CALL):
 - * **e** is the call expression of the subprogram named **x** with argument expressions **args** and parameter expressions **named_args**;
 - * define **ids** as the union of the singleton set for **x**, and the set obtained by applying *use_e* to each expression in **args** and each expression in **named_args**.
- All of the following apply (E_SLICE):
 - * **e** is the slicing expression over expression **e1** and slices **slices**;
 - * define **ids** as the union of applying *use_e* to **e1** and applying *use_slice* to each slice in **slices**.
- All of the following apply (E_COND):
 - * **e** is the conditional expression over expressions **e1**, **e2**, and **e3**;
 - * define **ids** as the union of applying *use_e* to each of **e1**, **e2**, and **e3**.
- All of the following apply (E_GETITEM):
 - * **e** is the tuple access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_GETFIELD):
 - * **e** is the field access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_GETFIELDS):
 - * **e** is the multiple field access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_RECORD):
 - * **e** is the record construction expression of type **ty** and field initializations **li**;
 - * define **ids** as the union of applying of *use_ty* to **ty** and applying *use_ty* to each field type in **li**.
- All of the following apply (E_CONCAT):
 - * **e** is the concatenation of expression **e.s**;

- * define **ids** as the union of applying of *use_e* to each expression in **e.s**.
- All of the following apply (E_TUPLE):
 - * **e** is the tuple construction expression for the expressions **e.s**;
 - * define **ids** as the union of applying of *use_e* to each expression in **e.s**.
- All of the following apply (E_UNKNOWN):
 - * **e** is the unknown expression with type **t**;
 - * define **ids** as the application of *use_ty* to **t**.
- All of the following apply (E_PATTERN):
 - * **e** is the pattern testing expression for subexpression **e1** and pattern **p**;
 - * define **ids** as the union of applying *use_e* to **e1** and applying *use_pattern* to **p**.

22.13.2 Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{use_e}(\overbrace{\text{None}}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
 \\
 \text{SOME} \\
 \frac{\text{use_e}(\text{e1}) \xrightarrow{\text{type}} \text{ids}}{\text{use_e}(\overbrace{\text{e1}}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}} \\
 \\
 \text{E_LITERAL} \\
 \text{use_e}(\overbrace{\text{E_Literal}(_)}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
 \\
 \text{E_ATC} \\
 \text{use_e}(\overbrace{\text{E_ATC}(\text{e1}, \text{ty})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_ty}(\text{ty})}^{\text{ids}} \\
 \\
 \text{E_VAR} \\
 \text{use_e}(\overbrace{\text{E_Var}(\text{x})}^e) \xrightarrow{\text{type}} \overbrace{\{\text{x}\}}^{\text{ids}} \\
 \\
 \text{E_GETARRAY} \\
 \text{use_e}(\overbrace{\text{E_GetArray}(\text{e1}, \text{e2})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}} \\
 \\
 \text{E_BINOP} \\
 \text{use_e}(\overbrace{\text{E_Binop}(_, \text{e1}, \text{e2})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}} \\
 \\
 \text{E_UNOP} \\
 \text{use_e}(\overbrace{\text{E_Unop}(_, \text{e1})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}} \\
 \\
 \text{E_CALL} \\
 \frac{\text{ids} := \{\text{x}\} \cup \bigcup_{\text{e1} \in \text{args}} \text{use_e}(\text{e1}) \cup \bigcup_{(_, \text{t}) \in \text{named_args}} \text{use_ty}(\text{t})}{\text{use_e}(\overbrace{\text{E_Call}(\text{x}, \text{args}, \text{named_args})}^e) \xrightarrow{\text{type}} \text{ids}}
 \end{array}$$

$$\begin{array}{c}
\text{E_SLICE} \\
\text{use_e}(\overbrace{\text{E_Slice}(\text{e1}, \text{slices})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \bigcup_{\text{s} \in \text{slices}} \text{use_slice}(\text{s})}^{\text{ids}} \\
\\
\text{E_COND} \\
\text{use_e}(\overbrace{\text{E_Cond}(\text{e1}, \text{e2}, \text{e3})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2}) \cup \text{use_e}(\text{e3})}^{\text{ids}} \\
\\
\begin{array}{cc}
\text{E_GETITEM} & \text{E_GETFIELD} \\
\text{use_e}(\overbrace{\text{E_GetItem}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}} & \text{use_e}(\overbrace{\text{E_GetField}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}}
\end{array} \\
\\
\text{E_GETFIELDS} \\
\text{use_e}(\overbrace{\text{E_GetFields}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}} \\
\\
\text{E_RECORD} \\
\text{use_e}(\overbrace{\text{E_Record}(\text{ty}, \text{li})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \bigcup_{(_, \text{t}) \in \text{li}} \text{use_ty}(\text{t})}^{\text{ids}} \\
\\
\text{E_CONCAT} \\
\text{use_e}(\overbrace{\text{E_Concat}(\text{e.s})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{e1} \in \text{e.s}} \text{use_e}(\text{e1})}^{\text{ids}} \\
\\
\begin{array}{cc}
\text{E_TUPLE} & \text{E_UNKNOWN} \\
\text{use_e}(\overbrace{\text{E_Concat}(\text{e.s})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{e1} \in \text{e.s}} \text{use_e}(\text{e1})}^{\text{ids}} & \text{use_e}(\overbrace{\text{E_Unknown}(\text{t})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{t})}^{\text{ids}}
\end{array} \\
\\
\text{E_PATTERN} \\
\text{use_e}(\overbrace{\text{E_Pattern}(\text{e1}, \text{p})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_pattern}(\text{p})}^{\text{ids}}
\end{array}$$

22.14 TypingRule.UseLexpr

The function

$$\text{use_le}(\overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the left-hand-side expression *le* depends on.

22.14.1 Prose

One of the following applies:

- All of the following apply (LE_VAR):
 - * `le` is a left-hand-side variable expression for `x`;
 - * define `ids` as the singleton set for `x`.
- All of the following apply (LE_DESTRUCTURING):
 - * `le` is a left-hand-side expression for assigning to a list of expressions `les`, that is `LE_Destructuring(les)`;
 - * define `ids` as the union of applying *use_le* to each expression in `les`.
- All of the following apply (LE_CONCAT):
 - * `le` is a left-hand-side concatenation of the list of expressions `les`;
 - * define `ids` as the union of applying *use_le* to each expression in `les`.
- All of the following apply (LE_DISCARD):
 - * `le` is a left-hand-side discard expression;
 - * define `ids` as the empty set.
- All of the following apply (LE_SETARRAY):
 - * `le` is a left-hand-side array update of the array given by the expression `e1` and index expression `e2`;
 - * define `ids` as the union of applying *use_le* to `e1` and applying *use_e* to `e2`.
- All of the following apply (LE_SETFIELD):
 - * `le` is a left-hand-side field update of the record given by the expression `e1`;
 - * define `ids` as the application of *use_le* to `e1`.
- All of the following apply (LE_SETFIELDS):
 - * `le` is a left-hand-side multiple field updates of the record given by the expression `e1`;
 - * define `ids` as the application of *use_le* to `e1`.
- All of the following apply (LE_SLICE):
 - * `le` is a left-hand-side slicing of the expression `e1` by slices `slices`;
 - * define `ids` as the union of applying *use_le* to `e1` and applying *use_slice* to each slice in `slices`.

22.14.2 Formally

$$\begin{array}{l}
\text{LE_VAR} \quad \overbrace{\text{use_le}(\text{LE_Var}(x))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{x}^{\text{ids}} \\
\text{LE_DESTRUCTURING} \quad \overbrace{\text{use_le}(\text{LE_Destructuring}(les))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\bigcup_{e \in les} \text{use_le}(e)}^{\text{ids}} \\
\text{LE_CONCAT} \quad \overbrace{\text{use_le}(\text{LE_Concat}(les))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\bigcup_{e \in les} \text{use_le}(e)}^{\text{ids}} \\
\text{LE_DISCARD} \quad \overbrace{\text{use_le}(\text{LE_Discard})}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
\text{LE_SETARRAY} \quad \overbrace{\text{use_le}(\text{LE_SetArray}(e1, e2))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1) \cup \text{use_e}(e2)}^{\text{ids}} \\
\text{LE_SETFIELD} \quad \overbrace{\text{use_le}(\text{LE_SetField}(e1, _))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1)}^{\text{ids}} \\
\text{LE_SETFIELDS} \quad \overbrace{\text{use_le}(\text{LE_SetFields}(e1, _))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1)}^{\text{ids}} \\
\text{LE_SLICE} \quad \overbrace{\text{use_le}(\text{LE_Slice}(e1, slices))}^{\text{le}} \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1) \cup \bigcup_{s \in slices} \text{use_slice}(s)}^{\text{ids}}
\end{array}$$

22.15 TypingRule.UsePattern

The function

$$\text{use_pattern}(\overbrace{\text{pattern}}^{\text{p}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the declaration **d** depends on.

22.15.1 Prose

One of the following applies:

- All of the following apply (MASK_ALL):
 - * **p** is either a mask pattern (Pattern.Mask) or a match-all pattern (Pattern.All);
 - * define **ids** as the empty set.

- All of the following apply (TUPLE):
 - * p is a tuple pattern list of patterns li ;
 - * define ids as the union of the application of *use_pattern* for each pattern in li .
- All of the following apply (ANY):
 - * p is a pattern for matching any of the patterns in the list of patterns li ;
 - * define ids as the union of the application of *use_pattern* for each pattern in li .
- All of the following apply (SINGLE):
 - * p is a pattern for matching the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (GEQ):
 - * p is a pattern for testing greater-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (LEQ):
 - * p is a pattern for testing less-than-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (NOT):
 - * p is a pattern negating the pattern $p1$;
 - * define ids as the application of *use_pattern* to $p1$.
- All of the following apply (RANGE):
 - * p is a pattern for testing the range of expressions from $e1$ to $e2$;
 - * define ids as the union of the application of *use_e* to both $e1$ and $e2$.

22.15.2 Formally

$$\begin{array}{c}
 \text{MASK_ALL} \\
 \hline
 \text{ast_label}(p) \in \{\text{Pattern_Mask}, \text{Pattern_All}\} \\
 \hline
 \text{use_pattern}(p) \xrightarrow{\text{type}} \overbrace{\emptyset}^{ids}
 \end{array}$$

$$\begin{array}{c}
 \text{TUPLE} \\
 \hline
 \text{use_pattern}(\overbrace{\text{Pattern_Tuple}(li)}^p) \xrightarrow{\text{type}} \overbrace{\bigcup_{p1 \in li} \text{use_pattern}(p1)}^{ids}
 \end{array}$$

$$\begin{array}{l}
\text{ANY} \\
\text{use_pattern}(\overbrace{\text{Pattern_Any}(\text{li})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{p1} \in \text{li}} \text{use_pattern}(\text{p1})}^{\text{ids}} \\
\\
\text{SINGLE} \\
\text{use_pattern}(\overbrace{\text{Pattern_Single}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{GEQ} \\
\text{use_pattern}(\overbrace{\text{Pattern_Geq}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{LEQ} \\
\text{use_pattern}(\overbrace{\text{Pattern_Leq}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{NOT} \\
\text{use_pattern}(\overbrace{\text{Pattern_Not}(\text{p1})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_pattern}(\text{p1})}^{\text{ids}} \\
\\
\text{RANGE} \\
\text{use_pattern}(\overbrace{\text{Pattern_Range}(\text{e1}, \text{e2})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}}
\end{array}$$

22.16 TypingRule.UseSlice

The function

$$\text{use_slice}(\overbrace{\text{slice}}^{\text{s}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the slice *s* depends on.

22.16.1 Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * *s* is the slice at the position given by the expression *e*;
 - * define *ids* as the application of *use_e* to *e*.
- All of the following apply (START_LENGTH_RANG):
 - * *s* is a slice given by the pair of expressions *e1* and *e2*;
 - * define *ids* as the union of applying *use_e* to both *e1* and *e2*.

22.16.2 Formally

$$\begin{array}{c}
 \text{SINGLE} \\
 \text{use_slice}(\overbrace{\text{Slice_Single}(e)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}}
 \end{array}
 \quad
 \frac{\text{STAR_LENGTH_RANGE} \quad L \in \{\text{Slice_Star}, \text{Slice_Length}, \text{Slice_Range}\}}{\text{use_slice}(\overbrace{L(e1, e2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e1) \cup \text{use_e}(e2)}^{\text{ids}}}$$

22.17 TypingRule.UseBitfield

The function

$$\text{use_bitfield}(\overbrace{\text{decl}}^{\text{bf}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the bitfield **bf** depends on.

22.17.1 Prose

One of the following applies:

- All of the following apply (SIMPLE):
 - * **bf** is the single field with slices **slices**;
 - * define **ids** as the union of applying *use_slice* to each slice in **slices**.
- All of the following apply (NESTED):
 - * **bf** is the nested bitfield with slices **slices** and bitfields **bitfields**;
 - * define **ids** as the union of applying *use_slice* to each slice in **slices** and applying *use_bitfield* to each bitfield in **bitfields**.
- All of the following apply (TYPE):
 - * **bf** is the typed bitfield with slices **slices** and type **ty**;
 - * define **ids** as the union of applying *use_slice* to each slice in **slices** and applying *use_ty* to **ty**.

22.17.2 Formally

$$\begin{array}{c}
\text{SIMPLE} \\
\text{use_bitfield}(\overbrace{\text{BitField_Simple}(_, \text{slices})}^{\text{bf}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{s \in \text{slices}} \text{use_slice}(s)}^{\text{ids}} \\
\\
\text{NESTED} \\
\frac{\text{ids} := \bigcup_{\text{bf1} \in \text{bitfields}} \text{use_bitfield}(s) \cup \bigcup_{s \in \text{slices}} \text{use_slice}(s)}{\text{use_bitfield}(\overbrace{\text{BitField_Nested}(_, \text{slices}, \text{bitfields})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{TYPE} \\
\frac{\text{ids} := \bigcup_{s \in \text{slices}} \text{use_slice}(s) \cup \text{use_ty}(\text{ty})}{\text{use_bitfield}(\overbrace{\text{BitField_Type}(_, \text{slices}, \text{ty})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}}
\end{array}$$

22.18 TypingRule.UseConstraint

The function

$$\text{use_constraint}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the integer constraint *c* depends on.

22.18.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * *c* is the single-value expression constraint with expression *e*;
 - * define *ids* as the application of *use_e* to *e*.
- All of the following apply (RANGE):
 - * *c* is the range constraint with expressions *e1* and *e2*;
 - * define *ids* as the union of applying *use_e* to both *e1* and *e2*.

22.18.2 Formally

$$\begin{array}{c}
\text{EXACT} \\
\text{use_constraint}(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}} \\
\\
\text{RANGE} \\
\text{use_constraint}(\overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e1) \cup \text{use_e}(e2)}^{\text{ids}}
\end{array}$$

22.19 TypingRule.UseStmt

The function

$$\text{use_s}(\overbrace{\text{stmt}}^{\mathbf{s}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\mathbf{ids}}$$

returns the set of identifiers \mathbf{ids} which the statement \mathbf{s} depends on.

22.19.1 Prose

One of the following applies:

- All of the following apply (PASS_RETURN_NONE_THROW_NONE):
 - * \mathbf{s} is either a pass statement `S.Pass`, a return-nothing statement `S.Return(None)`, or a throw-nothing statement (`S.Throw(None)`);
 - * define \mathbf{ids} as the empty set.
- All of the following apply (S_SEQ):
 - * \mathbf{s} is a sequencing statement for $\mathbf{s1}$ and $\mathbf{s2}$;
 - * define \mathbf{ids} as the union of applying `use_s` to both $\mathbf{s1}$ and $\mathbf{s2}$.
- All of the following apply (ASSERT_RETURN_SOME):
 - * \mathbf{s} is either an assertion with expression \mathbf{e} or a return statement with expression \mathbf{e} ;
 - * define \mathbf{ids} as the application of `use_e` to \mathbf{e} .
- All of the following apply (S_ASSIGN):
 - * \mathbf{s} is an assignment statement with left-hand-side \mathbf{le} and right-hand-side \mathbf{e} ;
 - * define \mathbf{ids} as the union of applying `use_le` to \mathbf{le} and `use_e` to \mathbf{e} .
- All of the following apply (S_CALL):
 - * \mathbf{s} is a call statement for the subprogram with name \mathbf{x} , arguments \mathbf{args} , and list of pairs consisting of a parameter identifier and associated expression `named_args`;
 - * define \mathbf{ids} as the union of the singleton set for \mathbf{x} , applying `use_e` to every expression in \mathbf{args} and applying `use_e` to every expression associated with a parameter in `named_args`.
- All of the following apply (S_COND):
 - * \mathbf{s} is the conditional statement with expression \mathbf{e} and statements $\mathbf{s1}$ and $\mathbf{s2}$;
 - * define \mathbf{ids} as the union of applying `use_e` to \mathbf{e} and `use_s` to both of $\mathbf{s1}$ and $\mathbf{s2}$.

- All of the following apply (S_CASE):

- * **s** is the case statement with expression **e** and case list **cases**;
- * define **ids** as the union of applying *use_e* to **e** and *use_case* to every case in **cases**.

- All of the following apply (S_FOR):

$$* \text{ s is the for statement } S_For \left\{ \begin{array}{ll} \text{index_name} & : \text{ } _ \\ \text{start_e} & : \text{start_e} \\ \text{for_direction} & : \text{direction} \\ \text{end_e} & : \text{end_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\};$$

- * define **ids** as the union of applying *use_e* to **limit**, **start_e**, and **end_e** and applying *use_s* to **s1**.

- All of the following apply (WHILE_REPEAT):

- * **s** is either a while statement or repeat statement, each with expression **e**, body statement **s1**, and optional limit expression **limit**;
- * define **ids** as the union of applying *use_e* to **limit** and to **e**, and applying *use_s* to **s1**.

- All of the following apply (S_DECL):

- * **s** is a declaration statement with local declaration item **ldi** and *optional* initialization expression **e**;
- * define **ids** as the union of applying *use_e* to **e** and *use_ldi* to **ldi**.

- All of the following apply (S_TRY):

- * **s** is a try statement with statement **s1**, catcher list **catchers**, and otherwise statement **s2**;
- * define **ids** as the union of applying *use_s* to both **s1** and **s2** and *use_catcher* to every catcher in **catchers**.

- All of the following apply (S_PRINT):

- * **s** is a print statement with list of expressions **args**;
- * define **ids** as the union of applying *use_e* to each expression in **args**.

22.19.2 Formally

$$\begin{array}{c}
\text{PASS_RETURN_NONE_THROW_NONE} \\
\frac{s = \text{S_Pass} \vee s = \text{S_Return}(\text{None}) \vee s = \text{S_Throw}(\text{None})}{\text{use_s}(s) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}} \\
\\
\text{S_SEQ} \qquad \text{ASSERT_RETURN_SOME} \\
\frac{\text{use_s}(\overbrace{\text{S_Seq}(s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_s}(s1) \cup \text{use_s}(s2)}^{\text{ids}} \quad \text{use_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}}}{\text{use_s}(\overbrace{\text{S_Seq}(s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_s}(s1) \cup \text{use_s}(s2)}^{\text{ids}}} \\
\\
\text{S_ASSIGN} \\
\frac{\text{use_s}(\overbrace{\text{S_Assign}(le, e)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_le}(le) \cup \text{use_e}(e)}^{\text{ids}}}{\text{use_s}(\overbrace{\text{S_Assign}(le, e)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_le}(le) \cup \text{use_e}(e)}^{\text{ids}}} \\
\\
\text{S_CALL} \\
\frac{\text{ids} := \{x\} \cup \bigcup_{e \in \text{args}} \text{use_e}(e) \cup \bigcup_{(_, e) \in \text{named_args}} \text{use_e}(e)}{\text{use_s}(\overbrace{\text{S_Call}(x, \text{args}, \text{named_args})}^s) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{S_COND} \\
\frac{\text{use_s}(\overbrace{\text{S_Cond}(e, s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \text{use_s}(s1) \cup \text{use_s}(s2)}^{\text{ids}}}{\text{use_s}(\overbrace{\text{S_Cond}(e, s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \text{use_s}(s1) \cup \text{use_s}(s2)}^{\text{ids}}} \\
\\
\text{S_CASE} \\
\frac{\text{use_s}(\overbrace{\text{S_Case}(e, \text{cases})}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \bigcup_{c \in \text{cases}} \text{use_case}(c)}^{\text{ids}}}{\text{use_s}(\overbrace{\text{S_Case}(e, \text{cases})}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \bigcup_{c \in \text{cases}} \text{use_case}(c)}^{\text{ids}}} \\
\\
\text{S_FOR} \\
\frac{\text{ids} := \text{use_e}(\text{limit}) \cup \text{use_e}(\text{start_e}) \cup \text{use_e}(\text{end_e}) \cup \text{use_s}(\text{body})}{\text{use_s} \left(\text{S_For} \left\{ \begin{array}{lcl} \text{index_name} & : & \text{—} \\ \text{start_e} & : & \text{start_e} \\ \text{for_direction} & : & \text{direction} \\ \text{end_e} & : & \text{end_e} \\ \text{body} & : & \text{body} \\ \text{limit} & : & \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{WHILE_REPEAT} \\
\frac{s = \text{S_While}(e, \text{limit}, s) \vee s = \text{S_Repeat}(s, e, \text{limit})}{\text{use_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{limit}) \cup \text{use_e}(e) \cup \text{use_s}(s1)}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{S_DECL} \\
\text{use_s}(\overbrace{\text{S_Decl}(_, \text{ldi}, e)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \text{use_ldi}(\text{ldi})}^{\text{ids}} \\
\\
\text{S_THROW_SOME} \\
\text{use_s}(\overbrace{\text{S_Throw}(\langle e \rangle)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}} \\
\\
\text{S_TRY} \\
\text{ids} := \text{use_s}(s1) \cup \bigcup_{c \in \text{catchers}} \text{use_catcher}(c) \cup \text{use_s}(s2) \\
\hline
\text{use_s}(\overbrace{\text{S_Try}(s1, \text{catchers}, s2)}^s) \xrightarrow{\text{type}} \text{ids} \\
\\
\text{S_PRINT} \\
\text{use_s}(\overbrace{\text{S_Print}(\text{args})}^s) \xrightarrow{\text{type}} \bigcup_{e \in \text{args}} \overbrace{\text{use_e}(e)}^{\text{ids}}
\end{array}$$

22.20 TypingRule.UseLDI

The function

$$\text{use_ldi}(\overbrace{\text{local_decl_item}}^{\text{ldi}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the local declaration item **ldi** depends on.

22.20.1 Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * **ldi** is a discarding declaration;
 - * define **ids** as the empty set.
- All of the following apply (TYPED):
 - * **ldi** is a typed declaration for the local declaration item **ldi1** and type **t**;
 - * define **ids** as the union of applying *use_ldi* to **ldi1** and *use_ty* to **t**.
- All of the following apply (TUPLE):
 - * **ldi** is a multi-variable declaration for the list of local declarations **ldis**;
 - * define **ids** as the union of applying *use_ldi* to each local declaration item in **ldis**.

22.20.2 Formally

$$\begin{array}{c}
\text{DISCARD} \\
\text{use_ldi}(\overbrace{\text{LDI_Discard}}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
\\
\text{TYPED} \\
\text{use_ldi}(\overbrace{\text{LDI_Typed}(\text{l di1}, \text{t})}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\text{use_ldi}(\text{l di1}) \cup \text{use_ty}(\text{t})}^{\text{ids}} \\
\\
\text{TUPLE} \\
\text{use_ldi}(\overbrace{\text{LDI_Tuple}(\text{l dis})}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{l di1} \in \text{l dis}} \text{use_ldi}(\text{l di1})}^{\text{ids}}
\end{array}$$

22.21 TypingRule.UseCase

The function

$$\text{use_case}(\overbrace{\text{case_alt}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the case alternative **c** depends on.

22.21.1 Prose

All of the following apply:

- **c** is the case alternative for the pattern **pattern**, **optional** where expression **e_opt** and **otherwise** statement **s**;
- define **ids** as the union of applying *use_pattern* to **pattern**, applying *use_e* to **e_opt**, and applying *use_s* to **s**.

22.21.2 Formally

$$\frac{\text{ids} := \text{use_pattern}(\text{pattern}) \cup \text{use_e}(\text{e_opt}) \cup \text{use_s}(\text{s})}{\text{use_case}(\overbrace{\{\text{pattern} : \text{pattern}, \text{where} : \text{e_opt}, \text{stmt} : \text{s}\}}^{\text{c}}) \xrightarrow{\text{type}} \text{ids}}$$

22.22 TypingRule.UseCatcher

The function

$$\text{use_catcher}(\overbrace{\text{catcher}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the try statement catcher **c** depends on.

22.22.1 Prose

All of the following apply:

- c is a case alternative with type \mathbf{ty} and statement \mathbf{s} ;
- define \mathbf{ids} as the union of applying use_ty to \mathbf{ty} and applying use_s to \mathbf{s} .

22.22.2 Formally

$$\text{use_catcher}(\overbrace{(_, \mathbf{ty}, \mathbf{s})}^c) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\mathbf{ty}) \cup \text{use_s}(\mathbf{s})}^{\mathbf{ids}}$$

Chapter 23

Static Evaluation

In this chapter, we define how to statically evaluate a subset of expressions and how to apply basic operators to literals via the following rules:

- `TypingRule.StaticEval` (see Section 23.1)
- `TypingRule.UnopLiterals` (see Section 23.2)
- `TypingRule.BinopLiterals` (see Section 23.3)

We also define the following helper rules:

- `TypingRule.SlicesToPositions` (see Section 23.4)
- `TypingRule.SliceToPositions` (see Section 23.5)
- `TypingRule.EvalToInt` (see Section 23.6)
- `TypingRule.ExtractSlice` (see Section 23.7)

In this chapter and the following ones, we use the special value `T` to represent a failure in transforming an expression into a desired form (the specific desired form varies according to the functions utilizing this value).

23.1 `TypingRule.StaticEval`

The function

$$\text{static.eval}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\mathcal{L}}^{\text{v}} \cup \{\text{T}\} \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

evaluates an expression `e`, from a restricted subset of all expressions, in the static environment `tenv`, returning a literal `v`. If `e` is not in the restricted set of expressions or cannot be statically evaluated to a compile-time constant, the result is `T`. Otherwise, the result is a type error.

23.1.1 Prose

One of the following applies:

- All of the following apply (E_LITERAL):
 - * e is the literal expression for the literal v , that is, $E_Literal(v)$.
- All of the following apply (E_VAR_CONSTANT):
 - * e is a variable expression with the identifier x , that is, $E_Var(x)$;
 - * determining whether x is bound to a constant in $tenv$ via *lookup_constant* yields the literal v .
- All of the following apply (E_VAR_NON_CONSTANT):
 - * e is a variable expression with the identifier x , that is, $E_Var(x)$;
 - * determining whether x is bound to a constant in $tenv$ via *lookup_constant* yields \perp (that is, x is not bound to a constant);
 - * checking whether x is defined in $tenv$ yields $TRUE\#TE$;
 - * the result is \top .
- All of the following apply (E_BINOP):
 - * e is a binary operation expression with operator op and operand expressions $e1$ and $e2$, that is, $E_Binop(op, e1, e2)$;
 - * applying *static_eval* to $e1$ in $tenv$ yields the literal $v1\#T, \#TE$;
 - * applying *static_eval* to $e2$ in $tenv$ yields the literal $v2\#T, \#TE$;
 - * applying op to $v1$ and $v2$ via *binop_literals* yields $v\#TE$.
- All of the following apply (E_UNOP):
 - * e is a unary operation expression with operator op and operand expression $e1$, that is, $E_Unop(op, e1)$;
 - * applying *static_eval* to $e1$ in $tenv$ yields the literal $v1\#T, \#TE$;
 - * applying op to $v1$ via *unop_literals* yields $v\#TE$.
- All of the following apply (E_SLICE_INT):
 - * e is a slicing expression of the integer literal for i and slice list $slices$, that is, $E_Slice(L_Int(i), slices)$;
 - * obtaining the indices of the slice list $slices$ in $tenv$ via *slices_to_positions* yields $positions\#TE$;
 - * pos_max is the maximum index in $positions$;
 - * converting the first $pos_max + 1$ digits of the binary representation of i into a bitvector via *int_to_bits* yields $bv1$;

- * extracting the slice of `bv1` given by `positions` yields `bv2`//^{#TE};
 - * `v` is the bitvector literal for `bv2`.
- All of the following apply (`E_SLICE_BITVECTOR`):
 - * `e` is a slicing expression of the bitvector literal for `bv` and slice list `slices`, that is, `E_Slice(L.Bitvector(bv), slices)`;
 - * obtaining the indices of the slice list `slices` in `tenv` via *slices_to_positions* yields `positions`//^{#TE};
 - * `pos_max` is the maximum index in `positions`;
 - * checking that the length of `bv` is greater than `pos_max` (which is 0-based) yields `TRUE`//^{#TE};
 - * extracting the slice of `bv` given by `positions` yields `bv2`//^{#TE};
 - * `v` is the bitvector literal for `bv2`.
 - All of the following apply (`E_SLICE_TYPE_ERROR`):
 - * `e` is a slicing expression of subexpression `e1` and slice list `slices`, that is, `E_Slice(e1, slices)`;
 - * `e1` is neither an integer literal nor a bitvector literal;
 - * the result is a type error indicating that either an integer literal or a bitvector literal were expected.
 - All of the following apply (`E_COND`):
 - * `e` is a conditional expression with condition subexpression `e_cond` and subexpressions `e1` (true case) and `e2` (false case), that is, `E_Cond(e_cond, e1, e2)`;
 - * evaluating `e_cond` in `tenv` either yields a Boolean literal `b` or a type error or `T`, either of which short-circuits the rule;
 - * `e'` is `e1` if `b` is `TRUE` and `e2` otherwise;
 - * the result is given by applying *static_eval* to `e'` in `tenv`.
 - All of the following apply (`UNSUPPORTED`):
 - * `e` is an expression that is not one of the following: a literal, a variable, a binary operation expression, a unary operation expression, a slice expression, and a conditional expression;
 - * the result is a type error indicating that `e` is not an expression that is supported for static evaluation.

23.1.2 Formally

$$\begin{array}{c}
\text{E_LITERAL} \\
\frac{}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_VAR_CONSTANT} \\
\frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} v}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_VAR_NON_CONSTANT} \\
\frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} \perp \quad \text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b \quad \text{check}(\neg b, \text{TE_UI}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{E_BINOP} \\
\frac{\text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \text{ // } \# \text{TE}, \top \quad \text{static_eval}(\text{tenv}, e2) \xrightarrow{\text{type}} v2 \text{ // } \# \text{TE}, \top \quad \text{binop_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} v \text{ // } \# \text{TE}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_UNOP} \\
\frac{\text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \text{ // } \# \text{TE}, \top \quad \text{unop_literals}(\text{op}, v1) \xrightarrow{\text{type}} v \text{ // } \# \text{TE}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, e1)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_SLICE_INT} \\
\frac{\text{slices_to_positions}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \text{ // } \# \text{TE} \quad \text{pos_max} := \max(\text{positions}) \quad \text{bv1} := \text{int_to_bits}(i, \text{pos_max} + 1) \quad \text{extract_slice}(\text{bv1}, \text{positions}) \xrightarrow{\text{type}} \text{bv2} \text{ // } \# \text{TE}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Slice}(\text{L_Int}(i), \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(\text{bv2})}^v)} \\
\\
\text{E_SLICE_BITVECTOR} \\
\frac{\text{slices_to_positions}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \text{ // } \# \text{TE} \quad \text{pos_max} := \max(\text{positions}) \quad \text{check}(|\text{bv}| > \text{pos_max}, \text{SliceOutOfRange}) \rightarrow \text{TRUE} \text{ // } \# \text{TE} \quad \text{extract_slice}(\text{bv}, \text{positions}) \xrightarrow{\text{type}} \text{bv2} \text{ // } \# \text{TE}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Slice}(\text{L_Bitvector}(\text{bv}), \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(\text{bv2})}^v)}
\end{array}$$

$$\begin{array}{c}
\text{E_SLICE_TYPE_ERROR} \\
\hline
\text{ast_label}(\mathbf{e1}) \notin \{\mathbf{L_Int}, \mathbf{L_Bitvector}\} \\
\hline
\text{static_eval}(\text{tenv}, \overbrace{\mathbf{E_Slice}(\mathbf{e1}, \mathbf{slices})}^{\mathbf{e}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch})
\end{array}$$

$$\begin{array}{c}
\text{E_COND} \\
\hline
\begin{array}{c}
\text{static_eval}(\text{tenv}, \mathbf{e_cond}) \xrightarrow{\text{type}} \mathbf{v_cond} \parallel \# \mathbf{TE}, \top \\
\mathbf{v_cond} \stackrel{\text{is}}{=} \mathbf{L_Bool}(\mathbf{b}) \quad \mathbf{e'} := \text{choice}(\mathbf{b}, \mathbf{e1}, \mathbf{e2}) \quad \text{static_eval}(\text{tenv}, \mathbf{e'}) \xrightarrow{\text{type}} \mathbf{v} \parallel \# \mathbf{TE}, \top
\end{array} \\
\hline
\text{static_eval}(\text{tenv}, \overbrace{\mathbf{E_Cond}(\mathbf{e_cond}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}) \xrightarrow{\text{type}} \mathbf{v}
\end{array}$$

$$\begin{array}{c}
\text{UNSUPPORTED} \\
\hline
\text{ast_label}(\mathbf{e}) \notin \{\mathbf{E_Literal}, \mathbf{E_Var}, \mathbf{E_Binop}, \mathbf{E_Unop}, \mathbf{E_Slice}, \mathbf{E_Cond}\} \\
\hline
\text{static_eval}(\text{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \text{TypeError}(\text{UnsupportedExpression})
\end{array}$$

23.2 TypingRule.UnopLiterals

The function

$$\text{unop_literals}(\overbrace{\mathbf{unop}}^{\mathbf{op}}, \overbrace{\mathcal{L}}^{\mathbf{l}}) \longrightarrow \overbrace{\mathcal{L}}^{\mathbf{r}} \cup \top \text{TypeError}$$

statically evaluates a unary operator \mathbf{op} (a terminal derived from the AST non-terminal for unary operators) over a literal \mathbf{l} and returns the resulting literal \mathbf{r} . Otherwise, the result is a type error.

The following set of unary operator types and argument types defines the correct argument type for a given unary operator:

$$\text{unop_signatures} \triangleq \left\{ \begin{array}{lll} (\text{NEG} & , & \mathbf{L_Int}) \\ (\text{NEG} & , & \mathbf{L_Real}) \\ (\text{BNOT} & , & \mathbf{L_Bool}) \\ (\text{NOT} & , & \mathbf{L_Bitvector}) \end{array} \right\}$$

23.2.1 Prose

One of the following applies:

- All of the following apply (ERROR):
 - * $(\mathbf{op}, \text{ast_label}(\mathbf{l}))$ is not in *unop_signatures*;
 - * the result is a type error indicating that the combination of \mathbf{op} and $\text{ast_label}(\mathbf{l})$ is not legal.
- All of the following apply (NEGATE_INT):

- * op is `NEG` and l is an integer literal for n ;
- * define r as the integer literal for $-n$.
- All of the following apply (`NEGATE_REAL`):
 - * op is `NEG` and l is a real literal for q ;
 - * define r as the real literal for $-q$.
- All of the following apply (`NOT_BOOL`):
 - * op is `BNOT` and l is a Boolean literal for b ;
 - * define r as the Boolean literal for $\neg b$.
- All of the following apply (`NOT_BITS_EMPTY`, `NOT_BITS_NOT_EMPTY`):
 - * op is `NOT` and l is a bitvector literal for the sequence of bits bits ;
 - * c is the sequence of bits of the same length as bits where in each position the bit in r is defined as the negation of the bit of bits in the same position;
 - * define r as the bitvector literal for c .

23.2.2 Formally

ERROR

$(\text{op}, l) \notin \text{unop_signatures}$

$\text{unop_literals}(\text{op}, \text{ast_label}(l)) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch})$

NEGATE_INT

$\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Int}(n)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(-n)}^r$

NEGATE_REAL

$\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Real}(q)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(-q)}^r$

NOT_BOOL

$\text{unop_literals}(\overbrace{\text{BNOT}}^{\text{op}}, \overbrace{\text{L_Bool}(b)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\neg b)}^r$

NOT_BITS_EMPTY

$\text{bits} \stackrel{\text{is}}{=} [] \quad c := []$

$\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r$

NOT_BITS_NOT_EMPTY

$\text{bits} \stackrel{\text{is}}{=} b_{1..k} \quad c := [i = 1..k : (1 - b_i)]$

$\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r$

23.3 TypingRule.BinopLiterals

The function

$$\textit{binop_literals}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\mathcal{L}}^{\text{v1}}, \overbrace{\mathcal{L}}^{\text{v2}}) \longrightarrow \overbrace{\mathcal{L}}^{\text{r}} \cup \text{TypeError}$$

statically evaluates a binary operator `op` (a terminal derived from the AST non-terminal for binary operators) over a pair of literals `l1` and `l2` and returns the resulting literal `r`. The result is a type error, if it is illegal to apply the operator to the given values, or a different kind of type error is detected.

The following set of binary operator types and argument types defines the correct

argument types for a given binary operator:

$$\text{binop_signatures} \triangleq \left\{ \begin{array}{l} \text{(PLUS , L_Int , L_Int) ,} \\ \text{(MINUS , L_Int , L_Int) ,} \\ \text{(MUL , L_Int , L_Int) ,} \\ \text{(DIV , L_Int , L_Int) ,} \\ \text{(DIVRM , L_Int , L_Int) ,} \\ \text{(MOD , L_Int , L_Int) ,} \\ \text{(POW , L_Int , L_Int) ,} \\ \text{(SHL , L_Int , L_Int) ,} \\ \text{(SHR , L_Int , L_Int) ,} \\ \text{(EQ_OP , L_Int , L_Int) ,} \\ \text{(NEQ , L_Int , L_Int) ,} \\ \text{(LEQ , L_Int , L_Int) ,} \\ \text{(LT , L_Int , L_Int) ,} \\ \text{(GEQ , L_Int , L_Int) ,} \\ \text{(GT , L_Int , L_Int) ,} \\ \text{(BAND , L_Bool , L_Bool) ,} \\ \text{(BOR , L_Bool , L_Bool) ,} \\ \text{(IMPL , L_Bool , L_Bool) ,} \\ \text{(EQ_OP , L_Bool , L_Bool) ,} \\ \text{(NEQ , L_Bool , L_Bool) ,} \\ \text{(PLUS , L_Real , L_Real) ,} \\ \text{(MINUS , L_Real , L_Real) ,} \\ \text{(MUL , L_Real , L_Real) ,} \\ \text{(RDIV , L_Real , L_Real) ,} \\ \text{(POW , L_Real , L_Int) ,} \\ \text{(EQ_OP , L_Real , L_Real) ,} \\ \text{(NEQ , L_Real , L_Real) ,} \\ \text{(LEQ , L_Real , L_Real) ,} \\ \text{(LT , L_Real , L_Real) ,} \\ \text{(GEQ , L_Real , L_Real) ,} \\ \text{(GT , L_Real , L_Real) ,} \\ \text{(EQ_OP , L_Bitvector , L_Bitvector) ,} \\ \text{(NEQ , L_Bitvector , L_Bitvector) ,} \\ \text{(OR , L_Bitvector , L_Bitvector) ,} \\ \text{(AND , L_Bitvector , L_Bitvector) ,} \\ \text{(XOR , L_Bitvector , L_Bitvector) ,} \\ \text{(MINUS , L_Bitvector , L_Bitvector) ,} \\ \text{(PLUS , L_Bitvector , L_Bitvector) ,} \\ \text{(MINUS , L_Bitvector , L_Int) ,} \\ \text{(PLUS , L_Bitvector , L_Int) } \end{array} \right\}$$

23.3.1 Prose

One of the following applies:

- All of the following apply (ERROR):
 - * (op, *ast_label*(11), *ast_label*(12)) is not included in *binop_signatures*;
 - * the result is a type error indicating the op cannot be applied to the arguments with the types given by *ast_label*(11) and *ast_label*(12).
- All of the following apply (ADD_INT):
 - * op is PLUS, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a + b$.
- All of the following apply (SUB_INT):
 - * op is MINUS, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a - b$.
- All of the following apply (MUL_INT):
 - * op is MUL, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a \times b$.
- All of the following apply (DIV_INT):
 - * op is DIV, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is positive yields *TRUE*//*#TE*;
 - * define n as a divided by b (note that n is potentially a fraction);
 - * checking that n is an integer yields *TRUE*//*#TE*;
 - * define r as the literal integer for $a \div b$.
- All of the following apply (FDIV_INT):
 - * op is DIVRM, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is positive yields *TRUE*//*#TE*;
 - * define n as a divided by b , rounded down (if a is negative, n is rounded down towards infinity);
 - * define r as the literal integer for n .
- All of the following apply (FREM_INT):
 - * op is MOD, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * applying *binop_literals* to DIVRM with 11 and 12 yields c //*#TE*;
 - * define n as $a - c$;
 - * define r as the literal integer for n .
- All of the following apply (EXP_INT):

- * `op` is `POW`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
- * checking that b is non-negative yields `TRUE//#TE`;
- * define n as a^b ;
- * define r as the literal integer for n .
- All of the following apply (`SHL`):
 - * `op` is `SHL`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * checking that b is non-negative yields `TRUE//#TE`;
 - * applying *binop.literals* to `POW` with `l2` and `l1` yields the literal integer for e ;
 - * applying *binop.literals* to `MUL` with `l2` and the literal integer for e yields r .
- All of the following apply (`SHR`):
 - * `op` is `SHR`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * checking that b is non-negative yields `TRUE//#TE`;
 - * applying *binop.literals* to `POW` with `l2` and `l1` yields the literal integer for e ;
 - * applying *binop.literals* to `DIVRM` with `l2` and the literal integer for e yields r .
- All of the following apply (`EQ_INT`):
 - * `op` is `EQ_OP`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (`NE_INT`):
 - * `op` is `NEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is different from b holds.
- All of the following apply (`LE_INT`):
 - * `op` is `LEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is less than or equal to bs .
- All of the following apply (`LT_INT`):
 - * `op` is `LT`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is less than bs .
- All of the following apply (`GE_INT`):
 - * `op` is `GEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is greater or equal than bs .

- All of the following apply (GT_INT):
 - * `op` is `GT`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than b .
- All of the following apply (AND_BOOL):
 - * `op` is `BAND`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if both a and b are `TRUE`.
- All of the following apply (OR_BOOL):
 - * `op` is `BOR`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if at least one of a and b is `TRUE`.
- All of the following apply (IMPLIES_BOOL):
 - * `op` is `IMPL`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is `FALSE` or b is `TRUE`.
- All of the following apply (EQ_BOOL):
 - * `op` is `EQ_OP`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (NE_BOOL):
 - * `op` is `NEQ`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is different from b .
- All of the following apply (ADD_REAL):
 - * `op` is `PLUS`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a + b$.
- All of the following apply (SUB_REAL):
 - * `op` is `MINUS`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a - b$.
- All of the following apply (MUL_REAL):
 - * `op` is `MUL`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a \times b$.
- All of the following apply (DIV_REAL):

- * `op` is `RDIV`, `l1` is the literal real for a , and `l2` is the literal real for b ;
- * checking whether b is different from 0 yields `TRUE`^{#TE};
- * define `r` as the real literal for $a \div b$.
- All of the following apply (`EXP_REAL`):
 - * `op` is `POW`, `l1` is the literal real for a , and `l2` is the literal integer for b ;
 - * define `r` as the real literal for a^b .
- All of the following apply (`EQ_REAL`):
 - * `op` is `EQ_OP`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (`NE_REAL`):
 - * `op` is `NEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is different from b .
- All of the following apply (`LE_REAL`):
 - * `op` is `LEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is less than or equal to b .
- All of the following apply (`LT_REAL`):
 - * `op` is `LT`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is less than b .
- All of the following apply (`GE_REAL`):
 - * `op` is `GEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than or equal to b .
- All of the following apply (`GT_REAL`):
 - * `op` is `GT`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than b .
- All of the following apply (`BITWISE_DIFFERENT_BITWIDTHS`):
 - * `v1` is a bitvector literal for a ;
 - * `v2` is a bitvector literal for b ;
 - * the lengths of a and b are different;

- * the result is a type error indicating that the bitvectors must be of the same width.
- All of the following apply (BITWISE_EMPTY):
 - * v1 is the empty bitvector literal;
 - * v2 is the empty bitvector literal;
 - * op is one of OR, AND, XOR, PLUS, or MINUS;
 - * define r as the empty bitvector literal.
- All of the following apply (EQ_BITS_EMPTY):
 - * v1 is the empty bitvector literal;
 - * v2 is the empty bitvector literal;
 - * op is EQ_OP;
 - * define r as the Boolean literal for TRUE.
- All of the following apply (EQ_BITS_NOT_EMPTY):
 - * v1 is a bitvector literal for $a_{1..k}$;
 - * v2 is a bitvector literal for $b_{1..k}$;
 - * op is EQ_OP;
 - * define b as TRUE if and only if a_i is equal to b_i , for $i = 1..k$;
 - * define r as the Boolean literal for b.
- All of the following apply (NE_BITS):
 - * v1 is a bitvector literal for a ;
 - * v2 is a bitvector literal for b ;
 - * op is NEQ;
 - * applying *binop_literals* to NEQ for v1 and v2 yields the Boolean literal for $\neg \text{b}$ ^{#TE};
 - * define r as the Boolean literal for $\neg \text{b}$.
- All of the following apply (OR_BITS):
 - * v1 is a bitvector literal for $a_{1..k}$;
 - * v2 is a bitvector literal for $b_{1..k}$;
 - * op is OR;
 - * define c_i as the maximum of a_i and b_i for $i = 1..k$;
 - * define r as the bitvector literal for $c_{1..k}$.
- All of the following apply (AND_BITS):

- * **v1** is a bitvector literal for $a_{1..k}$;
- * **v2** is a bitvector literal for $b_{1..k}$;
- * **op** is AND;
- * define c_i as the minimum of a_i and b_i for $i = 1..k$;
- * define **r** as the bitvector literal for $c_{1..k}$.
- All of the following apply (XOR_BITS):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is XOR;
 - * define c_i as 1 if a_i is different from b_i and 0 otherwise, for $i = 1..k$;
 - * define **r** as the bitvector literal for $c_{1..k}$.
- All of the following apply (ADD_BITS):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is PLUS;
 - * define a as the natural number represented by $a_{1..k}$;
 - * define b as the natural number represented by $b_{1..k}$;
 - * define c as the two's complement little endian representation of $a + b$ in k bits;
 - * define **r** as the bitvector literal for c .
- All of the following apply (SUB_BITS):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is MINUS;
 - * define a as the natural number represented by $a_{1..k}$;
 - * define b as the natural number represented by $b_{1..k}$;
 - * define c as the two's complement little endian representation of $a - b$ in k bits;
 - * define **r** as the bitvector literal for c .
- All of the following apply (ADD_BITS_INT):
 - * **v1** is a bitvector literal for a ;
 - * **v2** is an integer literal for b ;
 - * **op** is PLUS;
 - * define y as the natural number represented by a ;

- * define c as the two's complement little endian representation of $y + b$ in $|a|$ bits;
- * define r as the bitvector literal for c .
- All of the following apply (SUB_BITS_INT):
 - * $v1$ is a bitvector literal for a ;
 - * $v2$ is an integer literal for b ;
 - * op is MINUS;
 - * define y as the natural number represented by a ;
 - * define c as the two's complement little endian representation of $y - b$ in $|a|$ bits;
 - * define r as the bitvector literal for c .

23.3.2 Formally

$$\frac{\text{ERROR} \quad (op, \text{ast_label}(l1), \text{ast_label}(l2)) \notin \text{binop_signatures}}{\text{binop_literals}(op, \overbrace{l1}^{v1}, \overbrace{l2}^{v2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch})}$$

Arithmetic Operators Over Integer Values

$$\begin{array}{c} \text{ADD_INT} \\ \text{binop_literals}(\overbrace{\text{PLUS}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a + b)}^r \\ \\ \text{SUB_INT} \\ \text{binop_literals}(\overbrace{\text{MINUS}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a - b)}^r \\ \\ \text{MUL_INT} \\ \text{binop_literals}(\overbrace{\text{MUL}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a \times b)}^r \\ \\ \text{DIV_INT} \\ \text{check}(b > 0, \text{DIV_DenominatorNegative}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\ n := a \div b \\ \text{check}(n \in \mathbb{Z}, \text{TE_DII}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\ \hline \text{binop_literals}(\overbrace{\text{DIV}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r \\ \\ \text{FDIV_INT} \\ \text{check}(b > 0, \text{FDIV_DenominatorNegative}) \longrightarrow \text{TRUE} \parallel \text{\#TE} \\ n := \text{choice}(a \geq 0, \lfloor a \div b \rfloor, -(\lceil (-a) \div b \rceil)) \\ \hline \text{binop_literals}(\overbrace{\text{DIVRM}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r \end{array}$$

FREM_INT

$$\frac{\text{binop_literals}(\text{DIVRM}, \text{L_Int}(a), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(c) \quad // \quad \#TE}{\text{binop_literals}(\overbrace{\text{MOD}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a - (c \times b))}^{\text{r}})}$$

EXP_INT

$$\frac{\text{check}(b \geq 0, \text{ExponentNegative}) \longrightarrow \text{TRUE} \quad // \quad \#TE}{\text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a^b)}^{\text{r}})}$$

SHL

$$\frac{\text{check}(b \geq 0, \text{ShifterNegative}) \longrightarrow \text{TRUE} \quad // \quad \#TE \quad \text{binop_literals}(\text{POW}, \text{L_Int}(2), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(e)}{\text{binop_literals}(\overbrace{\text{MUL}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(e)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}}$$

SHR

$$\frac{\text{check}(b \geq 0, \text{ShifterNegative}) \longrightarrow \text{TRUE} \quad // \quad \#TE \quad \text{binop_literals}(\text{POW}, \text{L_Int}(2), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(e)}{\text{binop_literals}(\overbrace{\text{DIVRM}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(e)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}}$$

Relational Operators Over Integer Values

EQ_INT

$$\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^{\text{r}}$$

NE_INT

$$\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^{\text{r}}$$

LE_INT

$$\text{binop_literals}(\overbrace{\text{LEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \leq b)}^{\text{r}}$$

LT_INT

$$\text{binop_literals}(\overbrace{\text{LT}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a < b)}^{\text{r}}$$

$$\text{GE_INT} \\ \text{binop_literals}(\overbrace{\text{GEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \geq b)}^r$$

$$\text{GT_INT} \\ \text{binop_literals}(\overbrace{\text{GT}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a > b)}^r$$

Boolean Operators Over Boolean Values

$$\text{AND_BOOL} \\ \text{binop_literals}(\overbrace{\text{BAND}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \wedge b)}^r$$

$$\text{OR_BOOL} \\ \text{binop_literals}(\overbrace{\text{BOR}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \vee b)}^r$$

$$\text{IMPLIES_BOOL} \\ \text{binop_literals}(\overbrace{\text{IMPL}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\neg a \vee b)}^r$$

$$\text{EQ_BOOL} \\ \text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^r$$

$$\text{NE_BOOL} \\ \text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^r$$

Arithmetic Operators Over Real Values

$$\text{ADD_REAL} \\ \text{binop_literals}(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a + b)}^r$$

$$\text{SUB_REAL} \\ \text{binop_literals}(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a - b)}^r$$

$$\text{MUL_REAL} \\ \text{binop_literals}(\overbrace{\text{MUL}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a \times b)}^r$$

$$\begin{array}{c}
\text{DIV_REAL} \\
\hline
\text{check}(b \neq 0, \text{RDIV_DenominatorZero}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\hline
\text{binop_literals}(\overbrace{\text{RDIV}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a \div b)}^r
\end{array}$$

$$\begin{array}{c}
\text{EXP_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a^b)}^r
\end{array}$$

Relational Operators Over Real Values

$$\begin{array}{c}
\text{EQ_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^r
\end{array}$$

$$\begin{array}{c}
\text{NE_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{LE_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{LEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \leq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{LT_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{LT}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a < b)}^r
\end{array}$$

$$\begin{array}{c}
\text{GE_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{GEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \geq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{GT_REAL} \\
\hline
\text{binop_literals}(\overbrace{\text{GT}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a > b)}^r
\end{array}$$

23.3.3 Operators Over Bitvectors

The function `binary_to_unsigned` : $\{0, 1\}^* \rightarrow \mathbb{N}$ converts a non-empty sequence of bits into a natural number:

$$\text{binary_to_unsigned}(a_{n..1}) \triangleq \sum_{i=1}^n a_i \cdot 2^{a_i}$$

and an empty sequence of bits into 0:

$$\text{binary_to_unsigned}([\]) \triangleq 0 \text{ .}$$

The function $\text{int_to_bits} : \overbrace{\mathbb{Z}}^{\text{val}} \times \overbrace{\mathbb{Z}}^{\text{width}} \rightarrow \{0,1\}^*$ converts an integer val to its two's complement little endian representation of width bits.

BITWISE_DIFFERENT_BITWIDTHS

$$\frac{|a| \neq |b|}{\text{binop_literals}(\overbrace{\text{op}}^{\text{v1}}, \overbrace{\text{L_Bitvector}(a)}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_RSB})}$$

BITWISE_EMPTY

$$\frac{\text{op} \in \{\text{OR}, \text{AND}, \text{XOR}, \text{PLUS}, \text{MINUS}\}}{\text{binop_literals}(\overbrace{\text{op}}^{\text{v1}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}([\])}^{\text{r}}}$$

EQ_BITS_EMPTY

$$\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v1}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\text{TRUE})}^{\text{r}}$$

EQ_BITS_NOT_EMPTY

$$\frac{\mathbf{b} := \bigwedge_{i=1}^k a_i = b_i}{\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\mathbf{b})}^{\text{r}}}$$

NE_BITS

$$\frac{\text{binop_literals}(\text{EQ_OP}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{L_Bool}(\mathbf{b}) \quad \text{// \#TE}}{\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bool}(\neg \mathbf{b})}$$

OR_BITS

$$\frac{i = 1..k : c_i = \max(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{OR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})}$$

AND_BITS

$$\frac{i = 1..k : c_i = \min(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{AND}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})}$$

XOR_BITS

$$\frac{\text{xor_bit} = \lambda a, b \in \{0, 1\}. \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \quad i = 1..k : c_i = \text{xor_bit}(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{XOR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})}$$

ADD_BITS

$$\frac{\begin{array}{l} a := \text{binary_to_unsigned}(a_{1..k}) \\ b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a + b, k) \end{array}}{\text{binop_literals}(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r}$$

SUB_BITS

$$\frac{\begin{array}{l} a := \text{binary_to_unsigned}(a_{1..k}) \\ b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a - b, k) \end{array}}{\text{binop_literals}(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r}$$

ADD_BITS_INT

$$\frac{\begin{array}{l} y := \text{binary_to_unsigned}(a) \quad c := \text{int_to_bits}(y + b, |a|) \end{array}}{\text{binop_literals}(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r}$$

SUB_BITS_INT

$$\frac{\begin{array}{l} y := \text{binary_to_unsigned}(a) \quad c := \text{int_to_bits}(y - b, |a|) \end{array}}{\text{binop_literals}(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r}$$

23.4 TypingRule.SlicesToPositions

The function

$$\text{slices_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\mathbb{Z}^*}^{\text{positions}} \cup \{\top\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

transforms the list of slices `slices` in `tenv` into a list of indices `positions`. The result is `⊤` if `slices` cannot be statically evaluated to a list of positions. Otherwise, the result is a type error.

23.4.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is the empty list;
 - * `positions` is the empty list.
- All of the following apply (NON_EMPTY):
 - * `view` `slices` as the list with `s` as its `head` and `slices1` as its `tail`;
 - * applying *slice_to_positions* to `s` in `tenv` yields the list of positions `positions1` // $\top, \#TE$;
 - * transforming `slices1` to a list of positions in `tenv` via *slice_to_positions* yields `positions2` // $\top, \#TE$;
 - * `positions` is the concatenation of `positions1` and `positions2`.

23.4.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{slice_to_positions}(\text{tenv}, \overbrace{[]^{\text{slices}}} \xrightarrow{\text{type}} \overbrace{[]^{\text{positions}}} \\
 \\
 \text{NON_EMPTY} \\
 \frac{\begin{array}{c} \text{slice_to_positions}(\text{tenv}, s) \xrightarrow{\text{type}} \text{positions1} \text{ // } \top, \#TE \\ \text{slice_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions2} \text{ // } \top, \#TE \end{array}}{\text{slice_to_positions}(\text{tenv}, \overbrace{[s] + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\text{positions1} + \text{positions2}}^{\text{positions}}}
 \end{array}$$

23.5 TypingRule.SliceToPositions

The function

$$\text{slice_to_positions}(\overbrace{SE}^{\text{tenv}}, \overbrace{\text{slice}}^s) \longrightarrow \overbrace{\mathbb{Z}^+}^{\text{positions}} \cup \{\top\} \cup \overbrace{T\text{TypeError}}^{\#TE}$$

transforms a slice `s` in `tenv` into a list of indices `positions`. The result is \top if `slices` cannot be statically evaluated to a list of positions. Otherwise, the result is a type error.

23.5.1 Prose

One of the following applies:

- All of the following apply (SINGLE):

- * **s** is a slice for a single position given by the expression **e**, that is, `Slice_Single(e)`;
 - * applying *eval.to.int* to **e** in **tenv** yields the integer n //**#TE**;
 - * checking that n is non-negative yields **TRUE**//**#TE**;
 - * **positions** is the list containing the single element n .
- All of the following apply (**RANGE**):
 - * **s** is a slice for a range given by the expression **e_top** for the top position and **e_bot** for the bottom position, that is, `Slice_Range(e_top, e_bot)`;
 - * applying *eval.to.int* to **e_bot** in **tenv** yields the integer b //**#TE**;
 - * applying *eval.to.int* to **e_top** in **tenv** yields the integer t //**#TE**;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields **TRUE**//**#TE**;
 - * **positions** is the list of integers from t down to b , inclusive.
 - All of the following apply (**LENGTH**):
 - * **s** is a slice for a length slice given by the expression **e_bot** for the bottom position and **length** for the length of the slice, that is, `Slice_Length(e_bot, length)`;
 - * applying *eval.to.int* to **e_bot** in **tenv** to an integer yields the integer b //**#TE**;
 - * applying *eval.to.int* to **length** in **tenv** to an integer yields the integer l //**#TE**;
 - * t is $b + l - 1$;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields **TRUE**//**#TE**;
 - * **positions** is the list of integers from t down to b , inclusive.
 - All of the following apply (**STAR**):
 - * **s** is a slice for a scaled slice given by the expression **factor** for the factor and **length** for the length of the slice (**length*factor**), that is, `Slice_Star(factor, length)`;
 - * applying *eval.to.int* to **factor** in **tenv** to an integer yields the integer f //**#TE**;
 - * applying *eval.to.int* to **length** in **tenv** to an integer yields the integer l //**#TE**;
 - * t is $(f \times l) + l - 1$;
 - * b is $t - l + 1$;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields **TRUE**//**#TE**;
 - * **positions** is the list of integers from t down to b , inclusive.

23.5.2 Formally

$$\begin{array}{c}
\text{SINGLE} \\
\frac{\begin{array}{c} \text{eval_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n \text{ // } \top, \#TE \\ \text{check}(n \geq 0, \text{BadSlice}) \longrightarrow \text{TRUE // \#TE} \end{array}}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Single}(e)}^s) \xrightarrow{\text{type}} [n]} \\
\\
\text{RANGE} \\
\frac{\begin{array}{c} \text{eval_to_int}(\text{tenv}, e_{\text{bot}}) \xrightarrow{\text{type}} b \text{ // } \top, \#TE \\ \text{eval_to_int}(\text{tenv}, e_{\text{top}}) \xrightarrow{\text{type}} t \text{ // } \top, \#TE \\ \text{check}(t \geq b \geq 0, \text{BadSlice}) \longrightarrow \text{TRUE // \#TE} \end{array}}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Range}(e_{\text{top}}, e_{\text{bot}})}^s) \xrightarrow{\text{type}} [t..b]} \\
\\
\text{LENGTH} \\
\frac{\begin{array}{c} \text{eval_to_int}(\text{tenv}, e_{\text{bot}}) \xrightarrow{\text{type}} b \text{ // } \top, \#TE \\ \text{eval_to_int}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} l \text{ // } \top, \#TE \\ t := b + l - 1 \quad \text{check}(t \geq b \geq 0, \text{BadSlice}) \longrightarrow \text{TRUE // \#TE} \end{array}}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Length}(e_{\text{bot}}, \text{length})}^s) \xrightarrow{\text{type}} [t..b]} \\
\\
\text{STAR} \\
\frac{\begin{array}{c} \text{eval_to_int}(\text{tenv}, \text{factor}) \xrightarrow{\text{type}} f \text{ // } \top, \#TE \\ \text{eval_to_int}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} l \text{ // } \top, \#TE \\ t := (f \times l) + l - 1 \quad b := t - l \quad \text{check}(t \geq b \geq 0, \text{BadSlice}) \longrightarrow \text{TRUE // \#TE} \end{array}}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Star}(\text{factor}, \text{length})}^s) \xrightarrow{\text{type}} [t..b]}
\end{array}$$

23.6 TypingRule.EvalToInt

The function

$$\text{eval_to_int}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathbb{Z}}^n \cup \{\top\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

statically evaluates the expression e to the integer n . The result is \top if e cannot be statically evaluated to an integer. Otherwise, the result is a type error.

23.6.1 Prose

All of the following apply:

- applying *static_eval* to e in tenv has one of three outcomes:
 - * a literal 1 , which satisfies the premise;

- * \top (which means the expression could not be evaluated to a literal), which short-circuits the entire rule; or
- * $\#TE$ (indicating a type error was detected), which short-circuits the entire rule;
- checking that 1 is an integer literal yields $TRUE//\#TE$;
- define 1 as the literal integer for n .

23.6.2 Formally

$$\frac{\begin{array}{c} static_eval(tenv, e) \xrightarrow{type} 1 // \top, \#TE \\ check(ast_label(1) = L_Int, ExpectedIntegerType) \longrightarrow TRUE // \#TE \\ 1 \stackrel{is}{=} L_Int(n) \end{array}}{eval_to_int(tenv, e) \xrightarrow{type} n}$$

23.7 TypingRule.ExtractSlice

The function

$$extract_slice(\overbrace{\{0, 1\}^*}^{bits}, \overbrace{\mathbb{Z}^*}^{positions}) \longrightarrow \overbrace{\{0, 1\}^*}^r \cup TTypeError$$

extracts from the list of bits **bits** the sublist **r** of bits appearing at the positions given by **positions**. Otherwise, the result is a type error.

23.7.1 Prose

Define **r** the sublist of **bits** given by taking **bits**[**i**], for every value given in **positions** (in the order they appear).

23.7.2 Formally

$$extract_slice(tenv, bits, positions) \xrightarrow{type} \overbrace{[i \in positions : bits[i]]}^r$$

Chapter 24

Symbolic Subsumption Testing

This chapter is concerned with implementing a [sound subsumption test](#), as defined in Section 6.3 and employed by `TypingRule.DomainSubtypeSatisfaction` (see Section 8.3).

The symbolic reasoning operates by first transforming types into expressions in a *symbolic domain* AST (defined next, reusing `int_constraint` from the untyped AST) over which it then operates:

```
sym_dom ::= D.Bool
          | D.String
          | D.Real
          | D.Symbols(identifier+)
          | D.Int(int_set)
          | D.Bits(int_set)
          | D.Unknown
int_set  ::= Finite( $\mathcal{P}_{\text{fin}}(\mathbb{Z}) \setminus \emptyset$ )
          | Top
          | FromSyntax(syntax)
syntax   ::= int_constraint*
```

- `D.Bool`, `D.String`, and `D.Real` denote the symbolic domains of the Boolean type, the string type, and the real type, respectively;
- `D.Symbols` denotes the symbolic domain of an enumeration type with a specific set of labels;
- `D.Int` denotes the symbolic domain of integer types. More specifically:
 - * We refer to an element of the form `D.Int(Finite(S))` as a *symbolic finite set integer domain*, which represents the set of integers S ;
 - * We refer to an element of the form `D.Int(FromSyntax(vcs))` as a *symbolic constrained integer domain*, which represents the set of integers given by the list of constraints vcs ; and

- * We refer to an element of the form `D.Int(Top)` as a *symbolic unconstrained integer domain*, which represents the set of all integers.
- `D.Bits` denotes the symbolic domain of bitvector types. More specifically:
 - * We refer to an element of the form `D.Bits(Finite(S))` as a *symbolic finite set width bitvector domain*, which represents all bitvectors whose width is given by one of the integers in S ;
 - * We refer to an element of the form `D.Bits(FromSyntax(vcs))` as a *symbolic constrained width bitvector domain*, which represents all bitvectors whose width is given by the list of constraints `vcs` ; and
 - * We refer to an element of the form `D.Bits(Top)` as a *symbolic unconstrained width bitvector domain*, which represents all bitvectors whose width is unknown.
- `D.Unknown` is assigned to types that are not treated precisely by the symbolic subsumption testing, such as array types, record types, and tuple types.

The main rule of this chapter is `TypingRule.SymSubsumes` (see Section 24.1), which defines the function *sym_subsumes*.

Other helper rules are as follows:

- `TypingRule.SymDomOfType` (see Section 24.2)
- `TypingRule.SymDomOfExpr` (see Section 24.3)
- `TypingRule.IntSetOp` (see Section 24.4)
- `TypingRule.IntSetToIntConstraints` (see Section 24.5)
- `TypingRule.SymDomOfLiteral` (see Section 24.6)
- `TypingRule.SymIntSetOfConstraints` (see Section 24.7)
- `TypingRule.ConstraintToIntSet` (see Section 24.8)
- `TypingRule.NormalizeToInt` (see Section 24.9)
- `TypingRule.SymDomIsSubset` (see Section 24.10)
- `TypingRule.SymIntSetSubset` (see Section 24.11)
- `TypingRule.ConstraintBinop` (see Section 24.12)
- `TypingRule.IsRightIncreasing` (see Section 24.14)
- `TypingRule.IsRightDecreasing` (see Section 24.15)
- `TypingRule.IsLeftIncreasing` (see Section 24.16)

24.1 TypingRule.SymSubsumes

The predicate

$$\text{sym_subsumes}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{ty}}^{\text{s}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

soundly approximates *subsumes*(tenv, t, s). Otherwise, the result is a type error.

We assume that both *t* and *s* have been successfully annotated as per Chapter 9 (otherwise a typing error prevents us from applying this function).

24.1.1 Prose

All of the following apply:

- applying *symdom_of_type* to *t* in *tenv* yields *dt*;
- applying *symdom_of_type* to *s* in *tenv* yields *ds*;
- applying *symdom_is_subset* to *dt* and *ds* in *tenv* yields *b*.

24.1.2 Formally

$$\frac{\text{symdom_of_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{dt} \quad \text{symdom_of_type}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{ds} \quad \text{symdom_is_subset}(\text{tenv}, \text{dt}, \text{ds}) \xrightarrow{\text{type}} \text{b}}{\text{sym_subsumes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{b}}$$

24.2 TypingRule.SymDomOfType

The function

$$\text{symdom_of_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{sym.dom}}^{\text{d}}$$

transforms a type *t* in a static environment *tenv* into a symbolic domain *d*.

24.2.1 Prose

One of the following applies:

- All of the following apply (T_BOOL):
 - * *t* is the Boolean type;
 - * define *d* as *D.Bool*.
- All of the following apply (T_STRING):
 - * *t* is the string type;
 - * define *d* as *D.String*.

- All of the following apply (T_REAL):
 - * `t` is the real type;
 - * define `d` as `D_Real`.
- All of the following apply (T_ENUM):
 - * `t` is the enumeration type with list of labels `li`;
 - * define `d` as the symbolic list of symbols `li`, that is, `D_Symbols(li)`.
- All of the following apply (INT_UNCONSTRAINED):
 - * `t` is the unconstrained integer type;
 - * define `d` as `D_Int(Top)`, which intuitively represents the entire set of integers.
- All of the following apply (INT_PARAMETERIZED):
 - * `t` is the `parameterized integer type` for the identifier `id`;
 - * define `d` as the symbolic constrained integer domain with a single constraint for the variable expression for `id`, that is, `D_Int(FromSyntax([Constraint.Exact(E_Var(id))]))`.
- All of the following apply (INT_WELL_CONSTRAINED_FINITE):
 - * `t` is the well-constrained integer type for the list of constraints `vcs`;
 - * applying `intset_of_intconstraints` to `vcs` in `tenv` yields `vis`;
 - * `vis` is a set of integers, that is, `ast_label(vis)` is `Finite`;
 - * define `d` as the symbolic finite set integer domain for `vis`.
- All of the following apply (INT_WELL_CONSTRAINED_SYMBOLIC):
 - * `t` is the well-constrained integer type for the list of constraints `vcs`;
 - * applying `intset_of_intconstraints` to `vcs` in `tenv` yields `vis`;
 - * `vis` is not a set of integers, that is, `ast_label(vis)` is not `Finite`;
 - * define `d` as the symbolic constrained integer domain for the list of constraints `vcs`, that is, `D_Int(FromSyntax(vcs))`.
- All of the following apply (BITS_FINITE_SINGLE):
 - * `t` is a bitvector type with width expression `width`;
 - * applying `syndom_of_expr` to `width` in `tenv` yields a symbolic integer domain for the integer `n`;
 - * define `d` as the symbolic finite set width bitvector domain for the integer `n`, that is, `D_Bits(Finite({n}))`.

- All of the following apply (BITS.FINITE_SET):
 - * t is a bitvector type with width expression $width$;
 - * applying *symdom_of_expr* to $width$ in $tenv$ yields a list of at least two integers;
 - * define d as the symbolic constrained width bitvector domain for the expression $width$, that is, $D_Bits(FromSyntax([Constraint_Exact(width)]))$.
- All of the following apply (BITS.CONSTRAINT_EXACT):
 - * t is a bitvector type with width expression $width$;
 - * applying *symdom_of_expr* to $width$ in $tenv$ yields a symbolic integer domain for the single expression constraint v , that is $D_Int(FromSyntax([Constraint_Exact(v)]))$;
 - * define d as the symbolic constrained width bitvector domain for the single-value constraint v , that is, $D_Bits(FromSyntax([Constraint_Exact(v)]))$.
- All of the following apply (BITS.SYMBOLIC):
 - * t is a bitvector type with width expression $width$;
 - * applying *symdom_of_expr* to $width$ in $tenv$ yields a symbolic integer domain for the symbolic constraint c , that is, $D_Int(FromSyntax(c))$, and c is not a single expression constraint;
 - * define d as the symbolic constrained width bitvector domain for the expression $width$, that is, $D_Bits(FromSyntax([Constraint_Exact(width)]))$.
- All of the following apply (BITS.TOP):
 - * t is a bitvector type with width expression $width$;
 - * applying *symdom_of_expr* to $width$ in $tenv$ yields a symbolic integer domain for the unconstrained integer, that is, $D_Int(Top)$;
 - * define d as the symbolic constrained width bitvector domain for the expression $width$, that is, $D_Bits(FromSyntax([Constraint_Exact(width)]))$.
- All of the following apply (UNSUPPORTED):
 - * t is one of the following types: an array type, an exception type, a record type, a tuple type;
 - * define s as *D_Unknown*, meaning that the symbolic reasoning does not currently reason about them precisely.
- All of the following apply (T.NAMED):
 - * t is the named type for identifier id ;
 - * applying *make_anonymous* to t in $tenv$ yields $t1$;
 - * applying *symdom_of_type* to $t1$ in $tenv$ yields d .

24.2.2 Formally

T_BOOL

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bool}}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bool}}^d$$

T_STRING

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_String}}^t) \xrightarrow{\text{type}} \overbrace{\text{D_String}}^d$$

T_REAL

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Real}}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Real}}^d$$

T_ENUM

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Enum}(\text{li})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Symbols}(\text{li})}^d$$

INT_UNCONSTRAINED

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{Top})}^d$$

INT_PARAMETERIZED

$$\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{id}))}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{FromSyntax}([\text{Constraint_Exact}(\text{E_Var}(\text{id}))])}^d$$

INT_WELL_CONSTRAINED_FINITE

$$\frac{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \text{vis} \quad \text{ast_label}(\text{vis}) = \text{Finite}}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{vis})}^d}$$

INT_WELL_CONSTRAINED_SYMBOLIC

$$\frac{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \text{vis} \quad \text{ast_label}(\text{vis}) \neq \text{Finite}}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{FromSyntax}(\text{vcs}))}^d}$$

Notice that a bitvector type given by $\text{T_Bits}(\text{width})$ guarantees that width is an integer type via TypingRule.TBits (see Section 9.6), which is why $\text{symdom_of_expr}(\text{tenv}, \text{width})$ is guaranteed to return a symbolic domain of the form $\text{D_Int}(i)$.

BITS_FINITE_SINGLE

$$\frac{\text{symdom_of_expr}(\text{tenv}, \text{width}) \xrightarrow{\text{type}} \text{D_Int}(\text{Finite}(\{n\}))}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bits}(\text{width})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{Finite}(\{n\}))}^d)}$$

BITS_FINITE_SET

$$\frac{\text{symdom_of_expr}(\text{tenv}, \text{width}) \xrightarrow{\text{type}} \text{D_Int}(\text{Finite}(N)) \quad |N| > 1}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bits}(\text{width})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{FromSyntax}([\text{Constraint_Exact}(\text{width})]))}^d}}$$

BITS_CONSTRAINT_EXACT

$$\frac{\text{symdom_of_expr}(\text{tenv}, \text{width}) \xrightarrow{\text{type}} \text{D_Int}(\text{FromSyntax}([\text{Constraint_Exact}(v)]))}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bits}(\text{width})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{FromSyntax}([\text{Constraint_Exact}(v)]))}^d}}$$

BITS_SYMBOLIC

$$\frac{\text{symdom_of_expr}(\text{tenv}, \text{width}) \xrightarrow{\text{type}} \text{D_Int}(\text{FromSyntax}(c)) \quad c \neq [\text{Constraint_Exact}(v)]}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bits}(\text{width})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{FromSyntax}([\text{Constraint_Exact}(\text{width})]))}^d}}$$

BITS_TOP

$$\frac{\text{symdom_of_expr}(\text{tenv}, \text{width}) \xrightarrow{\text{type}} \text{D_Int}(\text{Top})}{\text{symdom_of_type}(\text{tenv}, \overbrace{\text{T_Bits}(\text{width})}^t) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{FromSyntax}([\text{Constraint_Exact}(\text{width})]))}^d}}$$

UNSUPPORTED

$$\frac{\text{ast_label}(t) \in \{\text{T_Array}, \text{T_Exception}, \text{T_Record}, \text{T_Tuple}\}}{\text{symdom_of_type}(\text{tenv}, t) \xrightarrow{\text{type}} \text{D_Unknown}}$$

T_NAMED

$$\frac{t = \text{T_Named}(\text{id}) \quad \text{make_anonymous}(t) \xrightarrow{\text{type}} t1 \quad \text{symdom_of_type}(\text{tenv}, t1) \xrightarrow{\text{type}} d}{\text{symdom_of_type}(\text{tenv}, t) \xrightarrow{\text{type}} d}$$

24.3 TypingRule.SymDomOfExpr

The function

$$\text{symdom_of_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{sym_dom}}^{\text{d}}$$

assigns a symbolic domain d to an integer typed expression e in the static environment tenv .

24.3.1 Prose

One of the following applies:

- All of the following apply (E_LITERAL):
 - * e is a literal expression for the literal v ;
 - * applying *symdom_of_literal* to v yields d .
- All of the following apply (E_VAR_CONSTANT):
 - * e is a variable expression for the identifier x ;
 - * applying *lookup_constant* to x in tenv yields the literal v ;
 - * applying *symdom_of_literal* to v yields d .
- All of the following apply (E_VAR_TYPE):
 - * e is a variable expression for the identifier x ;
 - * applying *lookup_constant* to x in tenv yields \perp ;
 - * applying *type_of* to x in tenv yields t1 ;
 - * applying *symdom_of_type* to t1 yields d .
- All of the following apply (E_UNOP_MINUS):
 - * e is a unary operation expression for the operation MINUS and subexpression e1 ;
 - * applying *symdom_of_expr* to the binary operation expression with the operation MINUS and the literal expression for 0 and e1 in tenv yields d .
- All of the following apply (E_UNOP_UNKNOWN):
 - * e is a unary operation expression for an operation that is not MINUS;
 - * define d as *D_Unknown*.
- All of the following apply (E_BINOP_SUPPORTED):
 - * e is a binary operation expression for an operation that is one of PLUS, MINUS, or MUL and subexpressions e1 and e2 ;

- * applying *syndom_of_expr* to *e1* in *tenv* yields a symbolic constrained integer domain with constraint *is1*, that is, *D.Int(is1)*;
 - * applying *syndom_of_expr* to *e2* in *tenv* yields a symbolic constrained integer domain with constraint *is2*, that is, *D.Int(is2)*;
 - * applying *intset_op* to *op* and *is1* and *is2* yields *vis*;
 - * define *s* as the symbolic finite set integer domain over *vis*, that is, *D.Int(vis)*.
- All of the following apply (*E_BINOP_SUNUPPORTED*):
 - * *e* is a binary operation expression for an operation that is not one of PLUS, MINUS, or MUL;
 - * define *s* as *D.Unknown*.
 - All of the following apply (*UNSUPPORTED*):
 - * *e* is not one of the following expression types a literal expression, a variable expression, a unary operation expression, or a binary operation expression;
 - * define *s* as *D.Unknown*.

24.3.2 Formally

$$\begin{array}{c}
 \text{E_LITERAL} \\
 \frac{\text{syndom_of_literal}(v) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} d} \\
 \\
 \text{E_VAR_CONSTANT} \\
 \frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} v \quad \text{syndom_of_literal}(v) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} d} \\
 \\
 \text{E_VAR_TYPE} \\
 \frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} \perp \quad \text{type_of}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \quad \text{syndom_of_type}(t1) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} d} \\
 \\
 \text{E_UNOP_MINUS} \\
 \frac{\text{syndom_of_expr}(\text{E_Binop}(\text{MINUS}, \text{E_Literal}(\text{L_Int}(0)), e1)) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Unop}(\text{MINUS}, e1)}^e) \xrightarrow{\text{type}} d}
 \end{array}$$

$$\begin{array}{c}
\text{E_UNOP_UNKNOWN} \\
\hline
\text{op} \neq \text{MINUS} \\
\hline
\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, \text{e1})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{D_Unknown}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{E_BINOP_SUPPORTED} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}, \text{MUL}\} \quad \text{syndom_of_expr}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{D_Int}(\text{is1}) \\
\text{syndom_of_expr}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{D_Int}(\text{is2}) \quad \text{intset_op}(\text{op}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \text{vis} \\
\hline
\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, \text{e1}, \text{e2})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{vis})}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{E_BINOP_UNSUPPORTED} \\
\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}\} \\
\hline
\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, _, _)}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{D_Unknown}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{UNSUPPORTED} \\
\text{ast_label}(\text{e}) \notin \{\text{E_Literal}, \text{E_Var}, \text{E_Unop}, \text{E_Binop}\} \\
\hline
\text{syndom_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{D_Unknown}}^{\text{d}}
\end{array}$$

24.4 TypingRule.IntSetOp

The function

$$\text{intset_op}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_set}}^{\text{is1}}, \overbrace{\text{int_set}}^{\text{is2}}) \longrightarrow \overbrace{\text{int_set}}^{\text{vis}}$$

applies the binary operation `op` to the symbolic integer sets `is1` and `is2`, yielding the symbolic integer set `vis`.

24.4.1 Prose

One of the following applies:

- All of the following apply (TOP):
 - * at least one of `is1` and `is2` is `Top`;
 - * define `vis` as `Top`.
- All of the following apply (FINITE_FINITE):
 - * `is1` is the symbolic finite set integer domain for `s1`;
 - * `is2` is the symbolic finite set integer domain for `s2`;

- * define **vis** as the symbolic finite set domain for the set obtained by applying **op** to each element of **s1** and each element of **s2**.
- All of the following apply (**FINITE_SYNTAX**):
 - * **is1** is the symbolic finite set integer domain for **s1**;
 - * **is2** is the symbolic constrained integer domain for **s2**;
 - * applying *int_set_to_int_constraints* to **s1** yields the list of constraints **cs1**;
 - * applying *intset_op* to **op**, the symbolic constrained integer domain for **cs1**, and **is2** yields **vis**.
- All of the following apply (**SYNTAX_FINITE**):
 - * **is1** is the symbolic constrained integer domain for **cs1**;
 - * **is2** is the symbolic finite set integer domain for **s2**;
 - * applying *int_set_to_int_constraints* to **s2** yields the list of constraints **cs2**;
 - * applying *intset_op* to **op**, **is1**, and the symbolic constrained integer domain for **cs2**, yields **vis**.
- All of the following apply (**SYNTAX_SYNTAX_WELL_CONSTRAINED**):
 - * **is1** is the symbolic constrained integer domain for **cs1**;
 - * **is2** is the symbolic constrained integer domain for **cs2**;
 - * applying *constraint_binop* to **op**, **cs1**, and **cs2** yields a list of constraints **vcs**;
 - * define **vis** as the symbolic constrained integer domain for **vcs**.
- All of the following apply (**SYNTAX_SYNTAX_TOP**):
 - * **is1** is the symbolic constrained integer domain for **cs1**;
 - * **is2** is the symbolic constrained integer domain for **cs2**;
 - * applying *constraint_binop* to **op**, **cs1**, and **cs2** yields \top ;
 - * define **vis** as **Top**.

24.4.2 Formally

$$\begin{array}{c}
 \text{TOP} \\
 \hline
 \text{is1} = \text{Top} \vee \text{is2} = \text{Top} \\
 \hline
 \text{intset_op}(\text{op}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{Top}}^{\text{vis}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{FINITE_FINITE} \\
 \hline
 \text{vis} := \text{Finite}(\{\text{op}(a, b) \mid a \in \text{s1}, b \in \text{s2}\}) \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

$$\begin{array}{c}
 \text{FINITE_SYNTAX} \\
 \hline
 \text{int_set_to_int_constraints}(\text{s1}) \xrightarrow{\text{type}} \text{cs1} \\
 \text{intset_op}(\text{op}, \text{FromSyntax}(\text{cs1}), \text{FromSyntax}(\text{cs2})) \xrightarrow{\text{type}} \text{vis} \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

$$\begin{array}{c}
\text{SYNTAX_FINITE} \\
\frac{\text{int_set_to_int_constraints}(\text{s2}) \xrightarrow{\text{type}} \text{cs2} \quad \text{intset_op}(\text{op}, \text{FromSyntax}(\text{cs1}), \text{FromSyntax}(\text{cs2})) \xrightarrow{\text{type}} \text{vis}}{\text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}} \\
\\
\text{SYNTAX_SYNTAX_WELL_CONSTRAINED} \\
\frac{\text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{WellConstrained}(\text{vcs})}{\text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\text{vcs})}^{\text{vis}}} \\
\\
\text{SYNTAX_SYNTAX_TOP} \\
\frac{\text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \top}{\text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{Top}}^{\text{vis}}}
\end{array}$$

24.5 TypingRule.IntSetToIntConstraints

The function

$$\text{int_set_to_int_constraints}(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\text{s}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{cs}}$$

transforms a finite set of integers into the equivalent list of integer constraints.

24.5.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * s is the empty set;
 - * define cs as the empty list.
- All of the following apply (SINGLETON):
 - * s is the singleton set for a ;
 - * define cs as the list containing the single range constraint for the interval starting from a and ending at a , that is, $\text{Constraint_Range}(\overbrace{\text{E_Literal}(\text{L_Int})}^{\text{a}}, \overbrace{\text{E_Literal}(\text{L_Int})}^{\text{a}})$.
- All of the following apply (NEW_INTERVAL):
 - * define a as the minimal element of s ;

- * define **s1** as the set **s** with a removed from it;
 - * applying *int_set_to_int_constraints* to **s1** yields the list of constraints **cs1**;
 - * **cs1** is a list where its **head** is a range constraint for the interval starting from b and ending at c and **tail** **cs2**;
 - * b is greater than $a + 1$;
 - * define **cs** as the list with first element a range constraint for the interval from a to a , second element a range constraint for the interval from b to c , and remaining elements given by **cs2**.
- All of the following apply (MERGE_INTERVAL):
 - * define a as the minimal element of **s**;
 - * define **s1** as the set **s** with a removed from it;
 - * applying *int_set_to_int_constraints* to **s1** yields the list of constraints **cs1**;
 - * **cs1** is a list where its **head** is a range constraint for the interval starting from b and ending at c and **tail** **cs2**;
 - * b is equal to $a + 1$;
 - * define **cs** as the list with **head** a range constraint for the interval from a to c and **tail** **cs2**.

24.5.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{int_set_to_int_constraints}(\overbrace{\{\emptyset\}}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{[]}^{\mathbf{cs}} \\
 \\
 \text{SINGLETON} \\
 \text{int_set_to_int_constraints}(\overbrace{\{a\}}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{[\text{Constraint_Range}(\overbrace{a}^{\text{E_Literal(L.Int)}}, \overbrace{a}^{\text{E_Literal(L.Int)})}]}^{\mathbf{cs}} \\
 \\
 \text{NEW_INTERVAL} \\
 \begin{array}{l}
 a = \min(\mathbf{s}) \quad \mathbf{s1} := \mathbf{s} \setminus \{a\} \quad \text{int_set_to_int_constraints}(\mathbf{s1}) \xrightarrow{\text{type}} \mathbf{cs1} \\
 \mathbf{cs1} = [\text{Constraint_Range}(\overbrace{b}^{\text{E_Literal(L.Int)}}, \overbrace{c}^{\text{E_Literal(L.Int)})}] + \mathbf{cs2} \quad b > a + 1 \\
 \mathbf{cs} := [\text{Constraint_Range}(\overbrace{a}^{\text{E_Literal(L.Int)}}, \overbrace{a}^{\text{E_Literal(L.Int)})}] + \\
 \quad [\text{Constraint_Range}(\overbrace{b}^{\text{E_Literal(L.Int)}}, \overbrace{c}^{\text{E_Literal(L.Int)})}] + \\
 \quad \mathbf{cs2} \\
 \hline
 \text{int_set_to_int_constraints}(\mathbf{s}) \xrightarrow{\text{type}} \mathbf{cs}
 \end{array}
 \end{array}$$

$$\begin{array}{c}
\text{MERGE_INTERVAL} \\
a = \min(s) \quad s1 := s \setminus \{a\} \quad \frac{\text{int_set_to_int_constraints}(s1) \xrightarrow{\text{type}} cs1}{\text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}} \\
cs1 = [\text{Constraint_Range}(\frac{\text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}}{b}, \frac{\text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}}{c})] + cs2 \\
b = a + 1 \quad cs := [\text{Constraint_Range}(\frac{\text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}}{a}, \frac{\text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}}{c})] + cs2 \\
\hline
\text{int_set_to_int_constraints}(s) \xrightarrow{\text{type}} cs
\end{array}$$

24.6 TypingRule.SymDomOfLiteral

The function

$$\text{symdom_of_literal}(\overbrace{\text{literal}}^v) \xrightarrow{\text{type}} \overbrace{\text{sym_dom}}^d$$

returns the symbolic domain d that corresponds to the literal v .

24.6.1 Prose

One of the following applies:

- All of the following apply (L_INT):
 - * v is an integer literal for n ;
 - * define d as the symbolic finite set integer domain for the singleton set for n , that is, $\text{D_Int}(\text{Finite}(\{n\}))$.
- All of the following apply (L_BOOL):
 - * v is a Boolean literal;
 - * define d as the symbolic Boolean domain.
- All of the following apply (L_REAL):
 - * v is a real literal;
 - * define d as the symbolic real domain.
- All of the following apply (L_STRING):
 - * v is a string literal;
 - * define d as the symbolic string domain.
- All of the following apply (L_BITVECTOR):
 - * v is a bitvector literal for the sequence of bits bv ;
 - * the length of bv is n ;
 - * define d as the symbolic finite set width bitvector domain for the singleton set for n .

24.6.2 Formally

$$\begin{array}{c}
\text{L_INT} \\
\text{syndom_of_literal}(\overbrace{\text{L_Int}(n)}^v) \xrightarrow{\text{type}} \overbrace{\text{D_Int}(\text{Finite}(\{n\}))}^d \\
\\
\text{L_BOOL} \\
\text{syndom_of_literal}(\overbrace{\text{L_Bool}(_)}^v) \xrightarrow{\text{type}} \overbrace{\text{D_Bool}}^d \\
\\
\text{L_REAL} \\
\text{syndom_of_literal}(\overbrace{\text{L_Real}(_)}^v) \xrightarrow{\text{type}} \overbrace{\text{D_Real}}^d \\
\\
\text{L_STRING} \\
\text{syndom_of_literal}(\overbrace{\text{L_String}(_)}^v) \xrightarrow{\text{type}} \overbrace{\text{D_String}}^d \\
\\
\text{L_BITVECTOR} \\
\frac{n := |\text{bv}|}{\text{syndom_of_literal}(\overbrace{\text{L_Bitvector}(\text{bv})}^v) \xrightarrow{\text{type}} \overbrace{\text{D_Bits}(\text{Finite}(\{n\}))}^d}
\end{array}$$

24.7 TypingRule.SymIntSetOfConstraints

The function

$$\text{intset_of_intconstraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{constraints}}^{\text{vcs}}) \longrightarrow \overbrace{\text{int.set}}^{\text{vis}}$$

returns the symbolic set of integers **vis** for the list of constraints **vcs** in the static environment **tenv**.

24.7.1 Prose

One of the following applies:

- All of the following apply (FINITE):
 - * applying *constraint_to_intset* to every constraint **vcs**[**i**] in **tenv**, for **i** in **indices**(**vcs**), yields a finite set of integers C_i , that is, **Finite**(C_i);
 - * define **vis** as the union of C_i for all **i** in **indices**(**vcs**).
- All of the following apply (SYMBOLIC):
 - * there exists a constraint **c** in **vcs** such that applying *constraint_to_intset* to **c** in **tenv** does not yield a finite set of integers;
 - * define **vis** as the symbolic constrained integer domain for **vcs**, that is, **FromSyntax**(**vcs**).

24.7.2 Formally

$$\begin{array}{c}
 \text{FINITE} \\
 \frac{\begin{array}{c} i \in \text{indices}(\text{vcs}) : \text{constraint_to_intset}(\text{tenv}, \text{vcs}[i]) \xrightarrow{\text{type}} \text{Finite}(C_i) \\ C := \bigcup_{i \in \text{indices}(\text{vcs})} C_i \end{array}}{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \overbrace{\text{Finite}(C)}^{\text{vis}}} \\
 \\
 \text{SYMBOLIC} \\
 \frac{\begin{array}{c} \exists i \in \text{indices}(\text{vcs}) : \text{constraint_to_intset}(\text{tenv}, \text{vcs}[i]) \xrightarrow{\text{type}} \\ \text{is1} \wedge \text{ast_label}(\text{is1}) \neq \text{Finite} \end{array}}{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\text{vcs})}^{\text{vis}}}
 \end{array}$$

24.8 TypingRule.ConstraintToIntSet

The function

$$\text{constraint_to_intset}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^{\text{c}}) \longrightarrow \overbrace{\text{int_set} \cup \{\text{Top}\}}^{\text{vis}}$$

transforms an integer constraint c into a set of integers vis or Top if the expressions involved in the integer constraints cannot be simplified to integers.

24.8.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is a single expression constraint for e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *normalize.to.int* to e in tenv yields the integer $n \text{ // } \text{Top}$;
 - * define vis as the singleton set for n , that is, $\text{Finite}(\{n\})$.
- All of the following apply (RANGE):
 - * c is a range constraint for $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *normalize.to.int* to $e1$ in tenv yields the integer $b \text{ // } \text{Top}$;
 - * applying *normalize.to.int* to $e2$ in tenv yields the integer $t \text{ // } \text{Top}$;
 - * define vis as the set integers that are both greater or equal to b and less than or equal to t .

24.8.2 Formally

$$\begin{array}{c}
\text{EXACT} \\
\hline
\text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n \text{ // } \text{Top} \\
\hline
\text{constraint_to_intset}(\text{tenv}, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Finite}(\{n\})}^{\text{vis}} \\
\\
\text{RANGE} \\
\begin{array}{c}
\text{normalize_to_int}(\text{tenv}, e1) \xrightarrow{\text{type}} b \text{ // } \text{Top} \\
\text{normalize_to_int}(\text{tenv}, e2) \xrightarrow{\text{type}} t \text{ // } \text{Top} \\
\text{vis} := \text{Finite}(\{n \mid b \leq n \leq t\})
\end{array} \\
\hline
\text{constraint_to_intset}(\text{tenv}, \overbrace{\text{Constraint.Range}(e1, e2)}^c) \xrightarrow{\text{type}} \text{vis}
\end{array}$$

24.9 TypingRule.NormalizeToInt

The function

$$\text{normalize_to_int}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathbb{Z}}^n \cup \{\text{Top}\}$$

symbolically simplifies the integer-typed expression e and returns the resulting integer or Top if the result of the simplification is not an integer.

We assume that e has been annotated as it is part of the constraint for an integer type, and therefore applying *normalize* to it does not yield a type error.

24.9.1 Prose

One of the following applies:

- All of the following apply (INTEGER):
 - * applying *normalize* to e in tenv yields the expression $e1$;
 - * applying *static_eval* to $e1$ in tenv yields the integer literal for n .
- All of the following apply (TOP):
 - * applying *normalize* to e in tenv yields the expression $e1$;
 - * applying *static_eval* to $e1$ in tenv yields \top .
 - * the result is Top .

24.9.2 Formally

$$\begin{array}{c}
 \text{INTEGER} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{L_Int}(n) \\
 \hline
 \text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n \\
 \\
 \text{TOP} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \top \\
 \hline
 \text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} \text{Top}
 \end{array}$$

24.10 TypingRule.SymDomIsSubset

The function

$$\text{symdom_is_subset}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{sym_dom}}^{\text{d1}}, \overbrace{\text{sym_dom}}^{\text{d2}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

conservatively tests whether the symbolic domain **d1** is subsumed by the symbolic domain **d2**, yielding the result **b**.

24.10.1 Prose

One of the following applies:

- All of the following apply (**DIFFERENT_LABELS**):
 - * the AST labels of **d1** and **d2** are different, which means they are symbolic domains of unrelated types;
 - * define **b** as **FALSE**.
- All of the following apply (**UNKNOWN**):
 - * one of **d1** and **d2** is **D_Unknown**;
 - * define **b** as **FALSE**.
- All of the following apply (**BOOL_STRING_REAL**):
 - * the AST labels of **d1** and **d2** are equal and they are one of **D_Bool**, **D_String**, and **D_Real**;
 - * define **b** as **TRUE**.
- All of the following apply (**SYMBOLS**):
 - * **d1** is a symbolic enumeration domain for the list of identifiers **is1**;
 - * **d2** is a symbolic enumeration domain for the list of identifiers **is2**;
 - * **b** is **TRUE** if and only if the set for **is1** is a subset of the set for **is2** (including if both sets are equal).

- All of the following apply (BITS):
 - * d1 is a symbolic bitvector domain for is1;
 - * d2 is a symbolic bitvector domain for is2;
 - * applying *sym_intset_subset* to is1 and is2 in tenv yields b.
- All of the following apply (INT):
 - * d1 is a symbolic integer domain for is1;
 - * d2 is a symbolic integer domain for is2;
 - * applying *sym_intset_subset* to is1 and is2 in tenv yields b.

24.10.2 Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABELS} \\
 \frac{ast_label(d1) \neq ast_label(d2)}{symdom_is_subset(tenv, d1, d2) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b} \\
 \\
 \text{UNKNOWN} \\
 \frac{ast_label(d1) = D_Unknown \vee ast_label(d2) = D_Unknown}{symdom_is_subset(tenv, d1, d2) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b} \\
 \\
 \text{BOOL_STRING_REAL} \\
 \frac{ast_label(d1) = ast_label(d2) \quad ast_label(d1) \in \{D_Bool, D_String, D_Real\}}{symdom_is_subset(tenv, d1, d2) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b} \\
 \\
 \text{SYMBOLS} \\
 \frac{b := \{s1\} \subseteq \{s2\}}{symdom_is_subset(tenv, \overbrace{D_Symbols(s1)}^{d1}, \overbrace{D_Symbols(s2)}^{d2}) \xrightarrow{\text{type}} b} \\
 \\
 \text{BITS} \\
 \frac{sym_intset_subset(tenv, is1, is2) \xrightarrow{\text{type}} b}{symdom_is_subset(tenv, \overbrace{D_Bits(is1)}^{d1}, \overbrace{D_Bits(is2)}^{d2}) \xrightarrow{\text{type}} b} \\
 \\
 \text{INT} \\
 \frac{sym_intset_subset(tenv, is1, is2) \xrightarrow{\text{type}} b}{symdom_is_subset(tenv, \overbrace{D_Int(is1)}^{d1}, \overbrace{D_Int(is2)}^{d2}) \xrightarrow{\text{type}} b}
 \end{array}$$

24.11 TypingRule.SymIntSetSubset

The function

$$\text{sym_intset_subset}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_set}}^{\text{is1}}, \overbrace{\text{int_set}}^{\text{is2}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

24.11.1 Prose

One of the following applies:

- All of the following apply (RIGHT_TOP):
 - * `is2` is `Top`;
 - * define `b` as `TRUE`.
- All of the following apply (LEFT_TOP_RIGHT_NOT_TOP):
 - * `is1` is `Top`;
 - * `is2` is not `Top`;
 - * define `b` as `FALSE`.
- All of the following apply (FINITE):
 - * `is1` is a finite set of integers for `s1`, that is, `Finite(s1)`;
 - * `is2` is a finite set of integers for `s2`, that is, `Finite(s2)`;
 - * define `b` as `TRUE` if and only if `s1` is a subset of `s2` or both sets are equal.
- All of the following apply (SYNTAX):
 - * `is1` is a set of integers given by the list of constraints `cs1`, that is, `FromSyntax(s1)`;
 - * `is2` is a set of integers given by the list of constraints `cs2`, that is, `FromSyntax(s2)`;
 - * applying `constraints.equal` to `cs1` and `cs2` in `tenv` yields `b`.
- All of the following apply (OTHER):
 - * both `is1` and `is2` are not `Top`;
 - * the AST labels of `is1` and `is2` are different;
 - * define `b` as `FALSE`.

24.11.2 Formally

$$\begin{array}{c}
\text{RIGHT_TOP} \\
\text{sym_intset_subset}(\text{tenv}, \text{is1}, \overbrace{\text{Top}}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}} \\
\\
\text{LEFT_TOP_RIGHT_NOT_TOP} \\
\frac{\text{is2} \neq \text{Top}}{\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{Top}}^{\text{is1}}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}} \\
\\
\text{FINITE} \\
\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}} \\
\\
\text{SYNTAX} \\
\frac{\text{constraints_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{b}}{\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{OTHER} \\
\frac{\text{is1} \neq \text{Top} \quad \text{is2} \neq \text{Top} \quad \text{ast_label}(\text{is1}) \neq \text{ast_label}(\text{is2})}{\text{sym_intset_subset}(\text{tenv}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}}
\end{array}$$

24.12 TypingRule.ConstraintBinop

The function

$$\text{constraint_binop}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\text{int_constraints}}^{\text{ics}}$$

symbolically applies the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding the integer constraints `ics`.

24.12.1 Prose

One of the following applies:

- All of the following apply (`WELL_CONSTRAINED`):
 - * for every `i` in `indices(cs1)` and every `j` in `indices(cs2)`, applying `constraint_binop_pair` to `op`, `cs1[i]`, and `cs2[j]`, yields the constraint $c_{(i,j)}$;
 - * define `cs` as the list consisting of constraints $c_{(i,j)}$, for every `i` in `indices(cs1)` and every `j` in `indices(cs2)`;

- * `ics` is the well-constrained list of constraints `cs`.
- All of the following apply (UNCONSTRAINED):
 - * there exist `i` in `indices(cs1)` and `j` in `indices(cs2)`, such that applying *constraint_binop_pair* to `op`, `cs1[i]`, and `cs2[j]`, yields \top ;
 - * `ics` is Unconstrained.

24.12.2 Formally

$$\begin{array}{c}
 \text{WELL_CONSTRAINED} \\
 i \in \text{indices}(cs1), j \in \text{indices}(cs2) : \\
 \text{constraint_binop_pair}(op, cs1[i], cs2[j]) \xrightarrow{\text{type}} c_{(i,j)} \\
 cs := [i \in \text{indices}(cs1), j \in \text{indices}(cs2) : c_{(i,j)}] \\
 \hline
 \text{constraint_binop}(op, cs1, cs2) \xrightarrow{\text{type}} \overbrace{\text{WellConstrained}(cs)}^{ics}
 \end{array}$$

$$\begin{array}{c}
 \text{UNCONSTRAINED} \\
 \exists i \in \text{indices}(cs1). \exists j \in \text{indices}(cs2). \\
 \text{constraint_binop_pair}(op, cs1[i], cs2[j]) \xrightarrow{\text{type}} \top \\
 \hline
 \text{constraint_binop}(op, cs1, cs2) \xrightarrow{\text{type}} \overbrace{\text{Unconstrained}}^{ics}
 \end{array}$$

24.13 TypingRule.ConstraintBinopPair

The function

$$\text{constraint_binop_pair}(\overbrace{\text{binop}}^{op}, \overbrace{\text{int_constraint}}^{c1}, \overbrace{\text{int_constraint}}^{c2}) \longrightarrow \overbrace{\text{int_constraint} \cup \{\top\}}^{\text{res}}$$

symbolically applies the binary operation `op` to the pair of integer constraints `c1` and `c2`, yielding `res` which is an integer constraint `c` or \top , which indicates a failure in representing the result as an integer constraint.

24.13.1 Prose

One of the following applies:

- All of the following apply (EXACT_EXACT):
 - * `c1` is an exact constraint for the expression `e1`, that is, `Constraint.Exact(e1)`;
 - * `c2` is an exact constraint for the expression `e2`, that is, `Constraint.Exact(e2)`;
 - * define `c` as the exact constraint for the binary operation expression with `op`, `e1`, and `e2`, that is, `Constraint.Exact(E.Binop(op, e1, e2))`;

- All of the following apply (EXACT_RANGE):
 - * c_1 is an exact constraint for the expression e_1 , that is, `Constraint.Exact(e_1)`;
 - * c_2 is a range constraint for the expressions $e_{2.1}$ and $e_{2.2}$, that is, `Constraint.Range($e_{2.1}$, $e_{2.2}$)`;
 - * One of the following applies:
 - All of the following apply:
 - ▷ applying *is_right_increasing* to op yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for op e_1 and $e_{2.1}$, and the binary operation expression for op e_1 and $e_{2.2}$;
 - All of the following apply:
 - ▷ applying *is_right_decreasing* to op yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for op e_1 and $e_{2.2}$, and the binary operation expression for op e_1 and $e_{2.1}$;
 - Otherwise, the result is `Top`;
- All of the following apply (RANGE_EXACT):
 - * c_1 is a range constraint for the expressions $e_{1.1}$ and $e_{1.2}$, that is, `Constraint.Range($e_{1.1}$, $e_{1.2}$)`;
 - * c_2 is an exact constraint for the expression e_2 , that is, `Constraint.Exact(e_2)`;
 - * One of the following applies:
 - All of the following apply:
 - ▷ applying *is_left_increasing* to op yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for op $e_{1.1}$ and e_2 , and the binary operation expression for op $e_{1.2}$ and e_2 ;
 - Otherwise, the result is `Top`.
- All of the following apply (RANGE_RANGE):
 - * c_1 is a range constraint for the expressions $e_{1.1}$ and $e_{1.2}$, that is, `Constraint.Range($e_{1.1}$, $e_{1.2}$)`;
 - * c_2 is a range constraint for the expressions $e_{2.1}$ and $e_{2.2}$, that is, `Constraint.Range($e_{2.1}$, $e_{2.2}$)`;
 - * One of the following applies:
 - All of the following apply:
 - ▷ applying *is_left_increasing* to op yields `TRUE`;
 - ▷ applying *is_right_increasing* to op yields `TRUE`;

- ▷ define c as the range constraint for the following two expressions: the binary operation expression for $\text{op } e1_1$ and $e2_1$, and the binary operation expression for $\text{op } e1_2$ and $e2_2$;
- All of the following apply:
 - ▷ applying *is_left_increasing* to op yields **TRUE**;
 - ▷ applying *is_right_decreasing* to op yields **TRUE**;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for $\text{op } e1_1$ and $e2_2$, and the binary operation expression for $\text{op } e1_2$ and $e2_1$;
- Otherwise, the result is **Top**.

24.13.2 Formally

EXACT_EXACT

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1}}^{\substack{c1 \\ \text{Constraint_Exact}}}, \overbrace{\text{e2}}^{\substack{c2 \\ \text{Constraint_Exact}}}) \xrightarrow{\text{type}} \overbrace{\text{e1 op e2}}^{\substack{c \\ \text{Constraint_Exact} \\ \text{E_Binop}}}$$

EXACT_RANGE

$$\text{res} := \begin{cases} \overbrace{\text{e1 op e2_1} \dots \text{e1 op e2_2}}^{\substack{\text{Constraint_Range} \\ \text{E_Binop} \quad \text{E_Binop}}} & \text{if } (\text{is_right_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \overbrace{\text{e1 op e2_2} \dots \text{e1 op e2_1}}^{\substack{\text{Constraint_Range} \\ \text{E_Binop} \quad \text{E_Binop}}} & \text{if } (\text{is_right_decreasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \text{Top} & \text{else} \end{cases}$$

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1}}^{\substack{c1 \\ \text{Constraint_Exact}}}, \overbrace{\text{e2_1} \dots \text{e2_2}}^{\substack{c2 \\ \text{Constraint_Range}}}) \xrightarrow{\text{type}} \text{res}$$

RANGE_EXACT

$$\text{res} := \begin{cases} \overbrace{\text{e1_1 op e2} \dots \text{e1_2 op e2}}^{\substack{\text{Constraint_Range} \\ \text{E_Binop} \quad \text{E_Binop}}} & \text{if } (\text{is_left_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \text{Top} & \text{else} \end{cases}$$

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1_1} \dots \text{e1_2}}^{\substack{c1 \\ \text{Constraint_Range}}}, \overbrace{\text{e2}}^{\substack{c2 \\ \text{Constraint_Exact}}}) \xrightarrow{\text{type}} \text{res}$$

$$\begin{array}{c}
\text{RANGE_RANGE} \\
\text{res} := \left\{ \begin{array}{l}
\begin{array}{c} \text{Constraint_Range} \\ \hline \begin{array}{c} \text{E_Binop} \quad \text{E_Binop} \\ \hline e1_1 \text{ op } e2_1 \dots e1_2 \text{ op } e2_2 \end{array} \end{array} \quad \text{if } \left(\begin{array}{l} is_left_increasing(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \wedge \\ is_right_increasing(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \end{array} \right) \\
\begin{array}{c} \text{Constraint_Range} \\ \hline \begin{array}{c} \text{E_Binop} \quad \text{E_Binop} \\ \hline e1_1 \text{ op } e2_2 \dots e1_2 \text{ op } e2_1 \end{array} \end{array} \quad \text{if } \left(\begin{array}{l} is_left_increasing(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \wedge \\ is_right_decreasing(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \end{array} \right) \\
\text{Top} \quad \text{else}
\end{array} \right. \\
\hline
\text{constraint_binop_pair}(\text{op}, \overbrace{e1_1 \dots e1_2}^{c1}, \overbrace{e2_1 \dots e2_2}^{c2}) \xrightarrow{\text{type}} \text{res}
\end{array}$$

24.14 TypingRule.IsRightIncreasing

The function

$$is_right_increasing(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \{\top\}$$

tests whether the value of binary operation **op** increases along with its right-hand-side operand, yielding the result in **b**. Otherwise, the result is \top , which indicates that the answer is not always **TRUE** or **FALSE**.

24.14.1 Prose

One of the following applies:

- All of the following (**TRUE**):
 - * **op** is one of **MUL**, **SHL**, **SHR**, **POW**, **PLUS**;
 - * define **b** as **TRUE**.
- All of the following (**FALSE**):
 - * **op** is one of **DIV**, **DIVRM**, **MOD**, **MINUS**;
 - * define **b** as **FALSE**.
- All of the following (**TOP**):
 - * **op** is **RDIV**;
 - * define **b** as \top .

24.14.2 Formally

$$\begin{array}{c}
\text{TRUE} \\
\hline
\text{op} \in \{\text{MUL}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}\} \\
\text{is_right_increasing}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}
\end{array}
\quad
\begin{array}{c}
\text{FALSE} \\
\hline
\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MOD}, \text{MINUS}\} \\
\text{is_right_increasing}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

$$\begin{array}{c}
\text{TOP} \\
\hline
\text{is_right_increasing}(\overbrace{\text{RDIV}}^{\text{op}}) \xrightarrow{\text{type}} \top
\end{array}$$

24.15 TypingRule.IsRightDecreasing

The function

$$\text{is_right_decreasing}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \{\top\}$$

tests whether the value of binary operation **op** decreases along with its right-hand-side operand, yielding the result in **b**. Otherwise, the result is \top , which indicates that the answer is not always **TRUE** or **FALSE**.

24.15.1 Prose

One of the following applies:

- All of the following (**TRUE**):
 - * **op** is **MINUS**;
 - * define **b** as **TRUE**.
- All of the following (**FALSE**):
 - * **op** is one of **DIV**, **DIVRM**, **MUL**, **SHL**, **SHR**, **POW**, **PLUS**, **MOD**;
 - * define **b** as **FALSE**.
- All of the following (**TOP**):
 - * **op** is one of **AND**, **BAND**, **BEQ**, **BOR**, **XOR**, **EQ_OP**, **GT**, **GEQ**, **IMPL**, **LT**, **LEQ**, **NEQ**, **OR**, **RDIV**;
 - * define **b** as \top .

24.15.2 Formally

$$\begin{array}{c}
\text{TRUE} \\
\hline
\text{is_right_decreasing}(\overbrace{\text{MINUS}}^{\text{op}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}
\end{array}
\quad
\begin{array}{c}
\text{FALSE} \\
\hline
\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MUL}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MOD}\} \\
\text{is_right_decreasing}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

$$\begin{array}{c}
\text{TOP} \\
\hline
\text{op} \in \{\text{AND}, \text{BAND}, \text{BEQ}, \text{BOR}, \text{XOR}, \text{EQ_OP}, \text{GT}, \text{GEQ}, \text{IMPL}, \text{LT}, \text{LEQ}, \text{NEQ}, \text{OR}, \text{RDIV}\} \\
\text{is_right_decreasing}(\text{op}) \xrightarrow{\text{type}} \top
\end{array}$$

24.16 TypingRule.IsLeftIncreasing

The function

$$is_left_increasing(\overbrace{binop}^{op}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \{\top\}$$

tests whether the value of binary operation `op` increases along with its left-hand-side operand, yielding `TRUE` or \top , which indicates that the answer is not always `TRUE` or `FALSE`.

24.16.1 Prose

One of the following applies:

- All of the following (`TRUE`):
 - * `op` is one of `MUL`, `DIV`, `DIVRM`, `MOD`, `SHL`, `SHR`, `POW`, `PLUS`, `MINUS`;
 - * define `b` as `TRUE`.
- All of the following (`TOP`):
 - * `op` is one of `AND`, `BAND`, `BEQ`, `BOR`, `XOR`, `EQ_OP`, `GT`, `GEQ`, `IMPL`, `LT`, `LEQ`, `NEQ`, `OR`, `RDIV`;
 - * define `b` as \top .

24.16.2 Formally

$$\frac{\begin{array}{c} \text{TRUE} \\ op \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \end{array}}{is_left_increasing(op) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

$$\frac{\begin{array}{c} \text{TOP} \\ op \in \{\text{AND}, \text{BAND}, \text{BEQ}, \text{BOR}, \text{XOR}, \text{EQ_OP}, \text{GT}, \text{GEQ}, \text{IMPL}, \text{LT}, \text{LEQ}, \text{NEQ}, \text{OR}, \text{RDIV}\} \end{array}}{is_left_increasing(op) \xrightarrow{\text{type}} \top}$$

Chapter 25

Symbolic Reduction and Equivalence Testing

In this chapter, we define two forms of symbolic reasoning — *symbolic reduction* and *symbolic equivalence testing*. Symbolic reduction simplifies expressions into *equivalent* expressions that are simpler to reason about. In our context, equivalence means that we can substitute one expression for another without affecting the semantics of the overall specification. Symbolic equivalence is a *conservative* test. By conservative, we mean that if a test for equivalence returns **TRUE** then the expressions being compared are indeed equivalent, but if the test returns **FALSE** then there are two possibilities:

- the expressions are not equivalent;
- the expressions are equivalent, but the reasoning power of our rules is not enough to prove it, and so we conservatively answer negatively.

In proof-theoretic terms, we can say that our equivalence tests are *sound* but *incomplete*. Notice that for a conservative test, it is always correct to return **FALSE**.

We first define symbolic expressions and operations over symbolic expressions (Section 25.1) and then we define the following rules:

- `TypingRule.Normalize` (see Section 25.2)
- `TypingRule.ReduceConstants` (see Section 25.3)
- `TypingRule.ReduceConstraint` (see Section 25.4)
- `TypingRule.ReduceConstraints` (see Section 25.5)
- `TypingRule.ToIR` (see Section 25.6)
- `TypingRule.ToIRCase` (see Section 25.7)
- `TypingRule.ExprEqualNorm` (see Section 25.8)

- `TypingRule.ExprEqual` (see Section 25.9)
- `TypingRule.ExprEqualCase` (see Section 25.10)
- `TypingRule.TypeEqual` (see Section 25.11)
- `TypingRule.BitwidthEqual` (see Section 25.12)
- `TypingRule.BitFieldsEqual` (see Section 25.13)
- `TypingRule.BitFieldEqual` (see Section 25.14)
- `TypingRule.ConstraintsEqual` (see Section 25.15)
- `TypingRule.ConstraintEqual` (see Section 25.16)
- `TypingRule.SlicesEqual` (see Section 25.17)
- `TypingRule.SliceEqual` (see Section 25.18)
- `TypingRule.ArrayLengthEqual` (see Section 25.19)
- `TypingRule.LiteralEqual` (see Section 25.20)

25.1 Symbolic Expressions

Our symbolic reduction and equivalence testing rules use *symbolic expressions*, defined below:

$$\begin{aligned} \text{polynomial} &\triangleq \text{Sum}(\text{unitary_monomial} \rightarrow \mathbb{Q}) \\ \text{unitary_monomial} &\triangleq \text{Prod}(\mathbb{I} \rightarrow \mathbb{N}^+) \end{aligned}$$

We now explain each component of a symbolic expression and how it can be interpreted as a mathematical formula via the interpretation function α . We also define operations over symbolic expressions.

Definition 32 (Unitary Monomial) A Unitary Monomial is a partial function from identifiers to positive integers¹.

A non-empty unitary monomial, $\text{Prod}(m) \in \text{unitary_monomial}$ where $m \neq \emptyset_\lambda$, can be interpreted as follows:

$$\alpha(\text{Prod}(m)) \triangleq \prod_{x \in \text{dom}(m)} x^{m(x)} .$$

An empty unitary monomial is interpreted as the constant 1:

$$\alpha(\text{Prod}(\emptyset_\lambda)) \triangleq 1 .$$

¹A unitary monomial has a unit factor, for example x^3 , whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

For example,

$$\alpha(\text{Prod}(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\})) = x^3 \cdot y \cdot z^2 .$$

The function

$$\text{mul_monomials}(\overbrace{\text{unitary_monomial}}^{m1}, \overbrace{\text{unitary_monomial}}^{m2}) \rightarrow \overbrace{\text{unitary_monomial}}^m$$

multiplies two unitary monomials and returns a unitary monomial

$$\frac{f := \lambda x \in \text{identifier.} \begin{cases} f1(x) & \text{if } x \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(x) & \text{if } x \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(x) + f2(x) & \text{else } x \in \text{dom}(f1) \cap \text{dom}(f2) \end{cases}}{\text{mul_monomials}(\overbrace{\text{Prod}(f1)}^{m1}, \overbrace{\text{Prod}(f2)}^{m2}) \xrightarrow{\text{type}} \overbrace{\text{Prod}(f)}^m}$$

For example,

$$\text{mul_monomials}(\text{Prod}(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}), \text{Prod}(\{x \mapsto 1, w \mapsto 2\})) = \text{Prod}(\{x \mapsto 4, y \mapsto 1, z \mapsto 2, w \mapsto 2\})$$

Definition 33 (Polynomial) Polynomials are partial functions from monomials to rationals. Intuitively, each unitary monomial is mapped to its factor in the polynomial. A polynomial $\text{Sum}(p)$ can be interpreted as follows:

$$\alpha(\text{Sum}(p)) \triangleq \sum_{m \in \text{dom}(p)} p(m) \cdot \alpha(m)$$

For example,

$$\text{Sum}\left(\left\{\begin{array}{ll} \text{Prod}(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}) & \mapsto -1, \\ \text{Prod}(\{x \mapsto 2, y \mapsto 1\}) & \mapsto \frac{3}{4} \end{array}\right\}\right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot x^2 \cdot y .$$

The function

$$\text{add_polynomials} : \text{polynomial} \times \text{polynomial} \rightarrow \text{polynomial}$$

adds two polynomials:

$$\frac{f := \lambda m \in \text{unitary_monomial.} \begin{cases} f1(m) & \text{if } m \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(m) & \text{if } m \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(m) + f2(m) & \text{else } m \in \text{dom}(f1) \cap \text{dom}(f2) \end{cases}}{\text{add_polynomials}(\overbrace{\text{Sum}(f1)}^{p1}, \overbrace{\text{Sum}(f2)}^{p2}) \xrightarrow{\text{type}} \overbrace{\text{Sum}(f)}^p}$$

The overloaded function

$$\text{add_polynomials} : \text{polynomial}^* \rightarrow \text{polynomial}$$

adds a list of polynomials:

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{add_polynomials}([\] \xrightarrow{\text{type}} \text{Prod}(\emptyset_\lambda)
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ONE} \\
 \text{add_polynomials}([p]) \xrightarrow{\text{type}} p
 \end{array}$$

$$\frac{
 \begin{array}{c}
 \text{TWO_OR_MORE} \\
 \text{add_polynomials}(p_{2..k}) \xrightarrow{\text{type}} p', \quad \text{add_polynomials}(p_1, p') \xrightarrow{\text{type}} p
 \end{array}
 }{
 \text{add_polynomials}(p_{1..k}) \xrightarrow{\text{type}} p
 }$$

The function

$$\text{mul_polynomials} : \overbrace{\text{polynomial}}^{p1} \times \overbrace{\text{polynomial}}^{p2} \rightarrow \overbrace{\text{polynomial}}^p$$

multiplies two polynomials.

$$\begin{array}{c}
 \text{ps} := \left\{ \begin{array}{l} \text{Sum}(\{ \text{mul_monomials}(m1, m2) \mapsto f1(m1) \times f2(m2) \}) \mid \\ m1 \in \text{dom}(f1) \ \wedge \ m2 \in \text{dom}(f2) \end{array} \right\} \\
 \text{ordered_ps} := [i = 1..k : p_i] \text{ such that } \{p_i \mid i = 1..k\} = \text{ps} \\
 \text{add_polynomials}(i = 1..k : \text{ordered_ps}) \xrightarrow{\text{type}} p
 \end{array}$$

$$\frac{
 \text{mul_polynomials}(\overbrace{\text{Sum}(f1)}^{p1}, \overbrace{\text{Sum}(f2)}^{p2}) \xrightarrow{\text{type}} p
 }{
 }$$

25.2 TypingRule.Normalize

The function

$$\text{normalize}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr} \cup \text{TTypeError}}^{\text{new_e}}$$

symbolically simplifies an expression e in the static environment tenv , yielding an expression new_e . Otherwise, the result is a type error.

25.2.1 Prose

One of the following applies:

- All of the following apply (NORMALIZABLE)
 - * applying *to_ir* to e in tenv to obtain a symbolic expression yields a symbolic expression $p1 \# \text{TE}$;
 - * applying *reduce_ir* to $p1$ to symbolically simplify $p1$ yields $p2$;
 - * applying *polynomial_to_expr* to $p2$ to transform it into an expression yields new_e .
- All of the following apply (NOT_NORMALIZABLE)
 - * applying *to_ir* to e in tenv to obtain a symbolic expression yields \top , indicating it cannot be transformed to a corresponding symbolic expression;
 - * define new_e as e .

25.2.2 Formally

NORMALIZABLE

$$\frac{\text{p1} \neq \top \quad \text{reduce_ir}(\text{p1}) \xrightarrow{\text{type}} \text{p2} \quad \text{polynomial_to_expr}(\text{p2}) \xrightarrow{\text{type}} \text{new_e}}{\text{normalize}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{new_e}}$$

NOT_NORMALIZABLE

$$\frac{\text{to_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \top}{\text{normalize}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{e}}^{\text{new_e}}}$$

25.3 TypingRule.ReduceConstants

The function

$$\text{reduce_constants}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{literal}}^{\text{l}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

symbolically simplifies an expression e into the literal l . Otherwise, the result is a type error.

25.3.1 Prose

One of the following applies:

- All of the following apply (EVAL_LITERAL):
 - * applying *static_eval* to e in tenv yields the literal $\text{l} \# \text{\#TE}$.
- All of the following apply (EVAL_NORMALIZED):
 - * applying *static_eval* to e in tenv yields \top ;
 - * applying *normalize* to e in tenv yields $\text{e1} \# \text{\#TE}$;
 - * applying *static_eval* to e1 in tenv yields the literal $\text{l} \# \text{\#TE}$.
- All of the following apply (EVAL_FAILURE):
 - * applying *static_eval* to e in tenv yields \top ;
 - * applying *normalize* to e in tenv yields $\text{e1} \# \text{\#TE}$;
 - * applying *static_eval* to e1 in tenv yields \top ;
 - * the result is a type error indicating that e cannot be reduced to a constant in tenv .

25.3.2 Formally

$$\begin{array}{c}
\text{EVAL_LITERAL} \\
\frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} 1 \quad \# \text{TE} \quad 1 \neq \top}{\text{reduce_constants}(\text{tenv}, e) \xrightarrow{\text{type}} 1} \\
\\
\text{EVAL_NORAMALIZED} \\
\frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \top \quad \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \# \text{TE} \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} 1 \quad \# \text{TE}}{\text{reduce_constants}(\text{tenv}, e) \xrightarrow{\text{type}} 1} \\
\\
\text{EVAL_FAILURE} \\
\frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \top \quad \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \top \quad \# \text{TE}}{\text{reduce_constants}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{CannotBeReducedToAConstant})}
\end{array}$$

25.4 TypingRule.ReduceConstraint

The function

$$\text{reduce_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}}$$

symbolically simplifies an integer constraint c , yielding the integer constraint new_c

25.4.1 Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an exact integer constraint for e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *normalize* to e in tenv yields e' ;
 - * define new_c as the exact integer constraint for e' , that is, $\text{Constraint_Exact}(e')$.
- All of the following apply (RANGE):
 - * c is a range integer constraint for $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *normalize* to $e1$ in tenv yields $e1'$;
 - * applying *normalize* to $e2$ in tenv yields $e2'$;
 - * define new_c as the range integer constraint for $e1'$ and $e2'$, that is, $\text{Constraint_Range}(e1', e2')$.

25.4.2 Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e' \\
 \hline
 \text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE} \\
 \hline
 \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \quad \text{normalize}(\text{tenv}, e2) \xrightarrow{\text{type}} e2' \\
 \hline
 \text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}
 \end{array}$$

25.5 TypingRule.ReduceConstraints

The function

$$\text{reduce_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraints}}^c) \longrightarrow \overbrace{\text{int_constraints}}^{\text{new_c}}$$

symbolically simplifies an integer constraints AST node c , yielding the integer constraints AST node new_c

25.5.1 Prose

One of the following applies:

- All of the following apply (UNCONSTRAINED_PARAMETERIZED):
 - * the AST label of c is either Unconstrained or Parameterized;
 - * define new_c as c .
- All of the following apply (WELL_CONSTRAINED):
 - * c is a list of constraints, that is, WellConstrained(cs);
 - * applying *reduce_constraint* to every constraint $cs[i]$ in tenv for every i in $\text{indices}(cs)$ yields c_i ;
 - * define new_cs as the list containing c_i for every i in $\text{indices}(cs)$;
 - * new_c is WellConstrained(new_cs).

25.5.2 Formally

$$\begin{array}{c}
 \text{UNCONSTRAINED_PARAMETERIZED} \\
 \text{ast_label}(c) \in \{\text{Unconstrained}, \text{Parameterized}\} \\
 \hline
 \text{reduce_constraints}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{c}^{\text{new_c}}
 \end{array}$$

$$\begin{array}{c}
\text{WELL_CONSTRAINED} \\
\frac{i \in \text{indices}(\text{cs}) : \text{reduce_constraint}(\text{tenv}, \text{cs}[i]) \xrightarrow{\text{type}} c_i}{\text{new_cs} := [i \in \text{indices}(\text{cs}) : c_i]} \\
\hline
\text{reduce_constraints}(\text{tenv}, \overbrace{\text{WellConstrained}(\text{cs})}^c) \xrightarrow{\text{type}} \overbrace{\text{WellConstrained}(\text{new_cs})}^{\text{new_c}}
\end{array}$$

25.6 TypingRule.ToIR

The function

$$\text{to_ir}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{polynomial}}^p \cup \{\top\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

transforms a subset of ASL expressions into symbolic expressions. If an ASL expression cannot be represented by a symbolic expression (because, for example, it contains operations that are not available in symbolic expressions), the special value \top is returned.

25.6.1 Prose

Intuitively, *to_ir* first conducts a case analysis to determine whether the ASL expression corresponds to a polynomial. If that fails, it proceeds to check whether the expression is a compile time constant.

One of the following applies:

- All of the following apply (CASE_SUCCESS):
 - * applying *to_ir_case* to *e* in *tenv* yields a symbolic expression *p*.
- All of the following apply (STATIC_EVAL_LITERAL):
 - * applying *to_ir_case* to *e* in *tenv* yields \top ;
 - * applying *static_eval* to *e* yields an integer literal for $v \ll \top$;
 - * define *p* as the polynomial representing the value *v*.
- All of the following apply (STATIC_EVAL_NON_LITERAL):
 - * applying *to_ir_case* to *e* in *tenv* yields \top ;
 - * applying *static_eval* to *e* in *tenv* yields a literal $1 \ll \top$;
 - * 1 is not an integer value;
 - * define *p* as \top .

25.6.2 Formally

$$\begin{array}{c}
\text{CASE_SUCCESS} \\
\frac{\text{to_ir_case}(\text{tenv}, e) \xrightarrow{\text{type}} p \quad p \neq \top}{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p} \\
\\
\text{STATIC_EVAL_LITERAL} \\
\frac{\text{to_ir_case}(\text{tenv}, e) \xrightarrow{\text{type}} \top \quad \text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{L_Int}(v) \parallel \top \quad p := \text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto v\})}{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p} \\
\\
\text{STATIC_EVAL_NON_LITERAL} \\
\frac{\text{to_ir_case}(\text{tenv}, e) \xrightarrow{\text{type}} \top \quad \text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} 1 \parallel \top \quad \text{ast_label}(1) \neq \text{L_Int}}{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \top}
\end{array}$$

25.7 TypingRule.ToIRCase

The function

$$\text{to_ir_case}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{polynomial}}^p \cup \{\top\}$$

transforms a subset of ASL expressions into symbolic expressions. If an expression cannot be represented by a symbolic expression, the special value \top is returned.

25.7.1 Prose

Intuitively, *to_ir* first conducts a case analysis to determine whether the ASL expression corresponds to a polynomial. If that fails, it proceeds to check whether the expression is a compile time constant.

One of the following applies:

- All of the following apply (LITERAL_INT):
 - * e is an integer literal expression for i , that is, $\text{E_Literal}(\text{L_Int}(i))$;
 - * p is the symbolic expression for i .
- All of the following apply (LITERAL_OTHER):
 - * e is a variable expression other than an integer literal;
 - * p is \perp .
- All of the following apply (INT_CONSTANT):
 - * e is a variable expression with identifier s , that is, $\text{E_Var}(s)$;
 - * looking up the constant associated with s in tenv yields the literal expression for v , that is, $\text{E_Literal}(v)$;

- * checking whether v is an integer literal yields $\text{TRUE} \# \text{TE}$;
- * v is an integer literal for i ;
- * p is the symbolic expression for i , that is, $\text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto i\})$.
- All of the following apply ($\text{INT_EXACT_CONSTANT}$):
 - * e is a variable expression with identifier s , that is, $\text{E_Var}(s)$;
 - * looking up the constant associated with s in tenv yields \perp ;
 - * determining the type of s yields $t \# \text{TE}$;
 - * the *underlying type* of t is $\text{ty1} \# \text{TE}$;
 - * checking whether ty1 is an integer type yields $\text{TRUE} \# \text{TE}$;
 - * ty1 is a well-constrained integer with the exact constraint e , that is, $\text{T_Int}(\text{WellConstrained}([\text{Constraint_Exact}(e)]))$;
 - * converting e to a symbolic expression yields p (which may possibly be \perp).
- All of the following apply (INT_VAR):
 - * e is a variable expression with identifier s , that is, $\text{E_Var}(s)$;
 - * looking up the constant associated with s in tenv yields \perp ;
 - * determining the type of s yields $t \# \text{TE}$;
 - * the *underlying type* of t is $\text{ty1} \# \text{TE}$;
 - * checking whether ty1 is an integer type yields $\text{TRUE} \# \text{TE}$;
 - * ty1 is not a well-constrained integer with a single exact constraint;
 - * p is the symbolic expression for the variable s , that is, $\text{Sum}(\{\text{Prod}(\{s \mapsto 1\}) \mapsto 1\})$.
- All of the following apply (EBINOP_PLUS):
 - * e is a binary addition expression with operands $e1$ and $e2$, that is, $\text{E_Binop}(\text{PLUS}, e1, e2)$;
 - * converting $e1$ to a symbolic expression in tenv yields $\text{ir1} \# \text{T}, \# \text{TE}$;
 - * converting $e2$ to a symbolic expression in tenv yields $\text{ir2} \# \text{T}, \# \text{TE}$;
 - * p is the symbolic expression adding up ir1 and ir2 .
- All of the following apply (EBINOP_MINUS):
 - * e is a binary subtraction expression with operands $e1$ and $e2$, that is, $\text{E_Binop}(\text{MINUS}, e1, e2)$;
 - * e' is the addition expression with operands $e1$ and the negation of $e2$, that is, $\text{E_Binop}(\text{PLUS}, e1, \text{E_Unop}(\text{MINUS}, e2))$;
 - * converting e' into a symbolic expression in tenv yields $p \# \text{T}, \# \text{TE}$.

- All of the following apply (EBINOP_MUL):
 - * e is a binary multiplication expression with operands $e1$ and $e2$, that is, $E_Binop(MUL, e1, e2)$;
 - * converting $e1$ to a symbolic expression in $tenv$ yields $ir1 \llcorner^{\top, \#TE}$;
 - * converting $e2$ to a symbolic expression in $tenv$ yields $ir2 \llcorner^{\top, \#TE}$;
 - * p is the symbolic expression multiplying $ir1$ and $ir2$.
- All of the following apply (EBINOP_DIV_NON_INT_DENOMINATOR):
 - * e is a binary division expression with operands $e1$ and $e2$, that is, $E_Binop(DIV, e1, e2)$;
 - * $e2$ is not an integer literal expression;
 - * p is \perp .
- All of the following apply (EBINOP_DIV_INT_DENOMINATOR):
 - * e is a binary division expression with operands $e1$ and $e2$, that is, $E_Binop(DIV, e1, e2)$;
 - * $e2$ is an integer literal expression for $i2$;
 - * converting $e1$ to a symbolic expression in $tenv$ yields $ir1 \llcorner^{\top, \#TE}$;
 - * $f2$ is $\frac{1}{i2}$ (testing against $i2 = 0$ is done dynamically);
 - * p is the polynomial $ir1$ with each monomial multiplied by $f2$.
- All of the following apply (EBINOP_SHL_NON_LINT_EXPONENT):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is not an integer literal expression;
 - * p is \perp .
- All of the following apply (EBINOP_SHL_NON_NEG_SHIFT):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is an integer literal expression for $i2$;
 - * $i2$ is negative;
 - * p is \perp .
- All of the following apply (EBINOP_SHL_OKAY):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is an integer literal expression for $i2$;

- * converting $e1$ to a symbolic expression in $tenv$ yields $ir1 \text{ // } \top, \#TE$;
- * $i2$ is non-negative;
- * $f2$ is 2^{i2} ;
- * p is the polynomial $ir1$ with each monomial multiplied by $f2$.
- All of the following apply (EBINOP_OTHER_NON_LITERALS):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * at least one of $e1$ and $e2$ is not a literal expression;
 - * p is \perp .
- All of the following apply (EBINOP_OTHER_LITERALS_NON_INT_RESULT):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, DIV, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * $e1$ is the literal expression for literal $l1$;
 - * $e2$ is the literal expression for literal $l2$;
 - * statically applying op to $l1$ and $l2$ yields the literal l , which is not an integer literal;
 - * p is \perp .
- All of the following apply (EBINOP_OTHER_LITERALS_INT_RESULT):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * $e1$ is the literal expression for literal $l1$;
 - * $e2$ is the literal expression for literal $l2$;
 - * statically applying op to $l1$ and $l2$ yields the integer literal for k ;
 - * p is the symbolic expression for the integer k , that is, $\text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto k\})$.
- All of the following apply (EUNOP_NEG):
 - * e is a unary expression with the negation operator NEG and operand $e1$;
 - * converting the binary expression with operator MUL and left-hand-side operand for the integer literal -1 and right-hand-side operand $e1$ in $tenv$ yields $p \text{ // } \top, \#TE$.
- All of the following apply (EUNOP_OTHER):
 - * e is a unary expression with an operator other than NEG;
 - * p is \top .
- All of the following apply (OTHER):
 - * e is an expression with a label other than E_Literal, E_Var, E_Binop, and E_Unop;
 - * p is \top .

25.7.2 Formally

$$\begin{array}{c}
\text{LITERAL_INT} \\
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Literal}(\text{L_Int}(i))}^e) \xrightarrow{\text{type}} \overbrace{\text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto i\})}^p \\
\\
\text{LITERAL_OTHER} \\
\frac{\text{ast_label}(v) \neq \text{L_Int}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{INT_CONSTANT} \\
\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{E_Literal}(v) \\ \text{check}(\text{ast_label}(v) = \text{L_Int}, \text{ExpectedIntegerLiteral}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ v \stackrel{\text{is}}{=} \text{L_Int}(i) \end{array}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} \overbrace{\text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto i\})}^p} \\
\\
\text{INT_EXACT_CONSTRAINT} \\
\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \top \quad \text{type_of}(s) \xrightarrow{\text{type}} t \text{ // } \#TE \\ \text{make_anonymous}(t) \xrightarrow{\text{type}} \text{ty1} \text{ // } \#TE \\ \text{check}(\text{ast_label}(\text{ty1}) = \text{T_Int}, \text{ExpectedIntegerType}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{ty1} = \text{T_Int}(\text{WellConstrained}([\text{Constraint_Exact}(e)])) \quad \text{to_ir}(e) \xrightarrow{\text{type}} p \end{array}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} p} \\
\\
\text{INT_VAR} \\
\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \top \quad \text{type_of}(s) \xrightarrow{\text{type}} t \quad \text{make_anonymous}(t) \xrightarrow{\text{type}} \text{ty1} \\ \text{check}(\text{ast_label}(\text{ty1}) = \text{T_Int}, \text{ExpectedIntegerType}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{ty1} \neq \text{T_Int}(\text{WellConstrained}([\text{Constraint_Exact}(_)])) \end{array}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} \overbrace{\text{Sum}(\{\text{Prod}(\{s \mapsto 1\}) \mapsto 1\})}^p} \\
\\
\text{EBINOP_PLUS} \\
\frac{\begin{array}{l} \text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \text{ // } \#TE, \top \\ \text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \text{ // } \#TE, \top \\ p := \text{add_polynomials}(\text{ir1}, \text{ir2}) \end{array}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Binop}(\text{PLUS}, e1, e2)}^e) \xrightarrow{\text{type}} p} \\
\\
\text{EBINOP_MINUS} \\
\frac{\begin{array}{l} e' := \text{E_Binop}(\text{PLUS}, e1, \text{E_Unop}(\text{MINUS}, e2)) \quad \text{to_ir}(\text{tenv}, e') \xrightarrow{\text{type}} p \text{ // } \#TE, \top \end{array}}{\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MINUS}, e1, e2)}^e) \xrightarrow{\text{type}} p}
\end{array}$$

EBINOP_MUL

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
\text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \parallel \#TE, \top \\
p := \text{mul_polynomials}(\text{ir1}, \text{ir2}) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{MUL}, e1, e2)}^e) \xrightarrow{\text{type}} p
\end{array}$$

EBINOP_DIV_NON_INT_DENOMINATOR

$$\begin{array}{c}
e2 \neq E.\text{Literal}(L.\text{Int}(_)) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{DIV}, e1, e2)}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_DIV_INT_DENOMINATOR

$$\begin{array}{c}
\text{f2} := \frac{1}{i2} \quad \text{ir1} \stackrel{\text{is}}{=} \text{Sum}[i = 1..k : m_i \mapsto c_i] \quad p := \text{Sum}[i = 1..k : m_i \mapsto c_i \times \text{f2}] \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{DIV}, e1, \overbrace{E.\text{Literal}(L.\text{Int}(i2))}^{e2})}^e) \xrightarrow{\text{type}} p
\end{array}$$

EBINOP_SHL_NON_INT_EXPONENT

$$\begin{array}{c}
e2 \neq E.\text{Literal}(L.\text{Int}(_)) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{SHL}, _, e2)}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_SHL_NEG_SHIFT

$$\begin{array}{c}
i2 < 0 \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{SHL}, e1, E.\text{Literal}(L.\text{Int}(i2)))}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_SHL_OKAY

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
i2 \geq 0 \\
\text{f2} := 2^{i2} \quad \text{ir1} \stackrel{\text{is}}{=} \text{Sum}[i = 1..k : m_i \mapsto c_i] \quad p := \text{Sum}[i = 1..k : m_i \mapsto c_i \times \text{f2}] \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{E.\text{Binop}(\text{SHL}, e1, E.\text{Literal}(L.\text{Int}(i2)))}^e) \xrightarrow{\text{type}} p
\end{array}$$

$$\begin{array}{c}
\text{EBINOP_OTHER_NON_LITERALS} \\
\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{DIV}, \text{SHL}\} \quad (e1 \neq \text{E_Literal}(_) \vee e2 \neq \text{E_Literal}(_)) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} \top
\\[10pt]
\text{EBINOP_OTHER_LITERALS_NON_INT_RESULT} \\
\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{SHL}\} \quad \text{binop_literals}(\text{op}, 11, 12) \xrightarrow{\text{type}} 1 \quad 1 \neq \text{L_Int}(_) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, \text{E_Literal}(11), \text{E_Literal}(12))}^e) \xrightarrow{\text{type}} \top
\\[10pt]
\text{EBINOP_OTHER_LITERALS_INT_RESULT} \\
\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{SHL}\} \\
\text{binop_literals}(\text{op}, 11, 12) \xrightarrow{\text{type}} \text{L_Int}(k) \quad p := \text{Sum}(\{\text{Prod}(\emptyset_\lambda) \mapsto k\}) \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, \text{E_Literal}(11), \text{E_Literal}(12))}^e) \xrightarrow{\text{type}} p
\\[10pt]
\text{EUNOP_NEG} \\
\text{to_ir}(\text{tenv}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(-1)), e1)) \xrightarrow{\text{type}} p \text{ // } \#TE, \top \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Unop}(\text{NEG}, e1)}^e) \xrightarrow{\text{type}} p
\\[10pt]
\text{EUNOP_OTHER} \\
\text{op} \neq \text{NEG} \\
\hline
\text{to_ir_case}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, _)}^e) \xrightarrow{\text{type}} \top
\\[10pt]
\text{OTHER} \\
\text{ast_label}(e) \notin \{\text{E_Literal}, \text{E_Var}, \text{E_Binop}, \text{E_Unop}\} \\
\hline
\text{to_ir_case}(\text{tenv}, e) \xrightarrow{\text{type}} \top
\end{array}$$

25.8 TypingRule.ExprEqualNorm

The function

$$\text{expr_equal_norm}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{e1}, \overbrace{\text{expr}}^{e2}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b \cup \overbrace{\top \text{TypeError}}^{\#TE}$$

conservatively tests whether the expression **e1** is equivalent to the expression **e2** in environment **tenv** by attempting to transform both expressions to their symbolic expression form and, if successful, comparing the resulting normal forms for equality. The result is given in **b** or a type error, if one is detected.

25.8.1 Prose

One of the following applies:

- All of the following apply (ALL_SUPPORTED):
 - * transforming **e1** into a symbolic expression in **tenv** yields **ir1** *//* **#TE**;
 - * transforming **e2** into a symbolic expression in **tenv** yields **ir2** *//* **#TE**;
 - * **b** is the result of equating **ir1** and **ir2**.
- All of the following apply (UNSUPPORTED1):
 - * transforming **e1** into a symbolic expression in **tenv** yields **⊤**;
 - * **b** is **FALSE**;
- All of the following apply (UNSUPPORTED2):
 - * transforming **e1** into a symbolic expression in **tenv** yields **ir1**;
 - * transforming **e2** into a symbolic expression in **tenv** yields **⊤**;
 - * **b** is **FALSE**;

25.8.2 Formally

$$\begin{array}{c}
 \text{ALL_SUPPORTED} \\
 \frac{\begin{array}{c} \text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \mathbf{ir1} \text{ // } \mathbf{\#TE} \\ \text{to_ir}(\mathbf{e2}) \xrightarrow{\text{type}} \mathbf{ir2} \text{ // } \mathbf{\#TE} \end{array}}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{ir1} = \mathbf{ir2}}^{\mathbf{b}}} \\
 \\
 \begin{array}{cc}
 \text{UNSUPPORTED1} & \text{UNSUPPORTED2} \\
 \frac{\text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \mathbf{\top}}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{FALSE}}^{\mathbf{b}}} & \frac{\text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \mathbf{ir1} \quad \text{to_ir}(\mathbf{e2}) \xrightarrow{\text{type}} \mathbf{\top}}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{FALSE}}^{\mathbf{b}}}
 \end{array}
 \end{array}$$

25.9 TypingRule.ExprEqual

The function

$$\text{expr_equal}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\mathbf{e1}}, \overbrace{\text{expr}}^{\mathbf{e2}}) \longrightarrow \overbrace{\{\mathbf{TRUE}, \mathbf{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\mathbf{\top TypeError}}^{\mathbf{\#TE}}$$

conservatively checks whether the expression **e1** is equivalent to the expression **e2** in environment **tenv**. The result is given in **b** or a type error, if one is detected.

25.9.1 Prose

One of the following applies:

- All of the following apply (NORM_TRUE):
 - * comparing **e1** to **e2** in **tenv** via *expr_equal_norm* yields **TRUE**//**#TE**;
 - * **b** is **TRUE**.
- All of the following apply (NORM_FALSE):
 - * comparing **e1** to **e2** in **tenv** via *expr_equal_norm* yields **FALSE**;
 - * comparing **e1** to **e2** by case analysis via *expr_equal_case* yields **b**//**#TE**.

25.9.2 Formally

$$\begin{array}{c}
 \text{NORM_TRUE} \\
 \frac{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{TRUE} \ // \ \mathbf{\#TE}}{\text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{TRUE}} \\
 \\
 \text{NORM_FALSE} \\
 \frac{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{FALSE} \quad \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \ // \ \mathbf{\#TE}}{\text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}
 \end{array}$$

25.10 TypingRule.ExprEqualCase

The function

$$\text{expr_equal_case}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\mathbf{e1}}, \overbrace{\text{expr}}^{\mathbf{e2}}) \longrightarrow \overbrace{\{\mathbf{TRUE}, \mathbf{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\mathbf{TTypeError}}^{\mathbf{\#TE}}$$

specializes the equivalence test for expressions **e1** and **e2** in **tenv** for the different types of expressions. The result is given in **b** or a type error, if one is detected.

25.10.1 Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of **e1** and **e2** are different;
 - * **b** is **FALSE**.
- All of the following apply (E_BINOP):
 - * **e1** is a binary expression with operator **op1** and operands **e1_1** and **e1_2**, that is, **E_Binop**(**op1**, **e1_1**, **e1_2**);

- * **e2** is a binary expression with operator **op2** and operands **e2.1** and **e2.2**, that is, **E_Binop(op2, e2.1, e2.2)**;
 - * testing the equivalence of **e1.1** and **e2.1** in **tenv** yields **b1** *//TE*;
 - * testing the equivalence of **e1.2** and **e2.2** in **tenv** yields **b2** *//TE*;
 - * **b** is **TRUE** if and only if **op1** is equal to **op2** and both **b1** and **b2** are **TRUE**.
- All of the following apply (**E_CALL**):
 - * **e1** is a call expression with subprogram name **name1** and list of arguments **args1**, that is, **E_Call(name1, args1, _)**;
 - * **e2** is a call expression with subprogram name **name2** and list of arguments **args2**, that is, **E_Call(name2, args2, _)**;
 - * checking whether **name1** is equal to **name2** either yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * checking whether the lists of arguments **args1** and **args2** have the same length yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **args1**, testing whether **args1[i]** is equivalent to **args2[i]** in **tenv** yields **b_i** *//TE*;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **args1**.
 - All of the following apply (**E_CONCAT**):
 - * **e1** is a concatenation expression with **l1**, that is, **E_Concat(l1)**;
 - * **e2** is a concatenation expression with **l2**, that is, **E_Concat(l2)**;
 - * checking whether the lists of expressions **l1** and **l2** have the same length yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **l1**, testing whether **l1[i]** is equivalent to **l2[i]** in **tenv** yields **b_i** *//TE*;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **l1**.
 - All of the following apply (**E_COND**):
 - * **e1** is a conditional expression with expressions **e1.1**, **e1.2**, and **e1.3**, that is, **E_Cond(e1.1, e1.2, e1.3)**;
 - * **e2** is a conditional expression with expressions **e2.1**, **e2.2**, and **e2.3**, that is, **E_Cond(e2.1, e2.2, e2.3)**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields **b1** *//TE*;
 - * testing whether **e1.2** is equivalent to **e2.2** yields **b2** *//TE*;
 - * testing whether **e1.3** is equivalent to **e2.3** yields **b3** *//TE*;
 - * **b** is **TRUE** if and only if all of **b1**, **b2**, and **b3** are **TRUE**.

- All of the following apply (E_SLICE):
 - * **e1** is a slicing expression with expression **e1_1** and list of slices **slices1**, that is, `E_Slice(e1_1, slices1)`;
 - * **e1** is a slicing expression with expression **e2_1** and list of slices **slices2**, that is, `E_Slice(e2_1, slices2)`;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b1**//**#TE**;
 - * testing whether the lists of slices **slices1** and **slices2** are equivalent in **tenv** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (E_GETARRAY):
 - * **e1** is an **array access** expression with array expression **e1_1** and position expression **e1_2**, that is, `E_GetArray(e1_1, e1_2)`;
 - * **e2** is an **array access** expression with array expression **e2_1** and position expression **e2_2**, that is, `E_GetArray(e2_1, e2_2)`;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b1**//**#TE**;
 - * testing whether **e1_2** is equivalent to **e2_2** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (E_GETFIELD):
 - * **e1** is a field access expression with subexpression **e1_1** and field name **field1**, that is, `E_GetField(e1_1, field1)`;
 - * **e2** is a field access expression with subexpression **e2_1** and field name **field2**, that is, `E_GetField(e2_1, field2)`;
 - * **b1** is **TRUE** if and only if **field1** is equal to **field2**;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (E_GETFIELDS):
 - * **e1** is a fields access expression with subexpression **e1_1** and list of field names **fields1**, that is, `E_GetFields(e1_1, fields1)`;
 - * **e2** is a fields access expression with subexpression **e2_1** and list of field names **fields2**, that is, `E_GetFields(e2_1, fields2)`;
 - * **b1** is **TRUE** if and only if **fields1** is equal to **fields2**;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (E_GETITEM):

- * **e1** is a tuple access expression with subexpression **e1.1** and position **i1**, that is, `E_GetItem(e1.1, i1)`;
 - * **e2** is a tuple access expression with subexpression **e2.1** and position **i2**, that is, `E_GetItem(e2.1, i2)`;
 - * **b1** is **TRUE** if and only if **i1** is equal to **i2**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (**E_LITERAL**):
 - * **e1** is the literal expression with literal **v1**;
 - * **e2** is the literal expression with literal **v2**;
 - * **b** is **TRUE** if and only if **v1** is equivalent to **v2** in **tenv**.
 - All of the following apply (**E_PATTERN**):
 - * both **e1** and **e2** are pattern expressions;
 - * **b** is **FALSE**.
 - All of the following apply (**E_RECORD**):
 - * both **e1** and **e2** are record expressions;
 - * **b** is **FALSE**.
 - All of the following apply (**E_TUPLE**):
 - * **e1** is a tuple expression with subexpression list **l1**, that is, `E_Tuple(l1)`;
 - * **e2** is a tuple expression with subexpression list **l2**, that is, `E_Tuple(l2)`;
 - * checking whether the lengths of **l1** and **l2** are equal yields either **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **l1**, testing whether **l1[i]** is equivalent to **l2[i]** in **tenv** yields **b_i**//**#TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **l1**.
 - All of the following apply (**E_UNOP**):
 - * **e1** is a unary operator expression with operator **op1** and operand expressions **e1.1**, that is, `E_Unop(op1, e1.1)`;
 - * **e2** is a unary operator expression with operator **op2** and operand expressions **e2.1**, that is, `E_Unop(op2, e2.1)`;
 - * testing whether **e1.1** is equivalent to **e2.1** in **tenv** yields **b1**;
 - * **b** is **TRUE** if and only if **op1** is equal to **op2** and **b1** is **TRUE**.
 - All of the following apply (**E_UNKNOWN**):

- * both **e1** and **e2** are UNKNOWN expressions;
- * **b** is **FALSE**.
- All of the following apply (E_ATC):
 - * **e1** is a type assertion with subexpression with operator **e1_1** and type **t1**, that is, $E_ATC(e1_1, t1)$;
 - * **e2** is a type assertion with subexpression with operator **e2_1** and type **t2**, that is, $E_ATC(e2_1, t2)$;
 - * testing whether **e1_1** is equivalent to **e2_1** in **tenv** yields **b1**;
 - * testing whether **t1** is equivalent to **t2** in **tenv** yields **b2**;
 - * **b** is **TRUE** if and only if both **b1** and **b2** are **TRUE**.
- All of the following apply (E_VAR):
 - * **e1** is a variable expression with identifier **name1**, that is, $E_Var(name1)$;
 - * **e2** is a variable expression with identifier **name2**, that is, $E_Var(name2)$;
 - * **b** is **TRUE** if and only if both **name1** is equal to **name2**.

25.10.2 Formally

$$\frac{\text{DIFFERENT_LABELS} \quad ast_label(e1) \neq ast_label(e2)}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} \text{FALSE}}$$

$$\frac{\begin{array}{l} E_BINOP \\ e1 \stackrel{\text{is}}{=} E_Binop(op1, e1_1, e1_2) \\ e2 \stackrel{\text{is}}{=} E_Binop(op2, e2_1, e2_2) \quad \begin{array}{l} expr_equal(e1_1, e2_1) \xrightarrow{\text{type}} b1 \quad \#TE \\ expr_equal(e1_2, e2_2) \xrightarrow{\text{type}} b2 \quad \#TE \end{array} \\ b := (op1 = op2) \wedge b1 \wedge b2 \end{array}}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} b}$$

(Recall that a conjunction over an empty set equals **TRUE**.)

$$\frac{\begin{array}{l} E_CALL \\ e1 \stackrel{\text{is}}{=} E_Call(name1, args1, _) \quad e2 \stackrel{\text{is}}{=} E_Call(name2, args2, _) \\ bool_transition(name1 = name2) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ equal_length(args1, args2) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{FALSE} \\ i \in indices(args1) : \begin{array}{l} expr_equal(tenv, args1[i], args2[i]) \xrightarrow{\text{type}} b_i \quad \#TE \\ b := \bigwedge_{i \in indices(args1)} b_i \end{array} \end{array}}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} b}$$

$$\begin{array}{c}
\text{E_CONCAT} \\
\begin{array}{l}
e1 \stackrel{\text{is}}{=} \text{E_Concat}(l1) \quad e2 \stackrel{\text{is}}{=} \text{E_Concat}(l2) \\
\text{equal_length}(l1, l2) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\
i \in \text{indices}(l1) : \text{expr_equal}(\text{tenv}, l1[i], l2[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\
b := \bigwedge_{i \in \text{indices}(l1)} b_i
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{E_COND} \\
\begin{array}{l}
e1 \stackrel{\text{is}}{=} \text{E_Cond}(e1_1, e1_2, e1_3) \quad e2 \stackrel{\text{is}}{=} \text{E_Cond}(e2_1, e2_2, e2_3) \\
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // } \#TE \\
\text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\
\text{expr_equal}(\text{tenv}, e1_3, e2_3) \xrightarrow{\text{type}} b3 \text{ // } \#TE \\
b := b1 \wedge b2 \wedge b3
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

$$\begin{array}{c}
\text{E_SLICE} \\
\begin{array}{l}
e1 \stackrel{\text{is}}{=} \text{E_Slice}(e1_1, \text{slices1}) \quad e2 \stackrel{\text{is}}{=} \text{E_Slice}(e2_1, \text{slices2}) \\
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // } \#TE \\
\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\
b := b1 \wedge b2
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{E_GETARRAY} \\
\begin{array}{l}
e1 \stackrel{\text{is}}{=} \text{E_GetArray}(e1_1, e1_2) \quad e2 \stackrel{\text{is}}{=} \text{E_GetArray}(e2_1, e2_2) \\
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // } \#TE \\
\text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\
b := b1 \wedge b2
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{E_GETFIELD} \\
\begin{array}{l}
e1 \stackrel{\text{is}}{=} \text{E_GetField}(e1_1, \text{field1}) \quad e2 \stackrel{\text{is}}{=} \text{E_GetField}(e2_1, \text{field2}) \\
b1 := \text{field1} = \text{field2} \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\
b := b1 \wedge b2
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETFIELDS

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} E_GetFields(e1_1, fields1) \quad e2 \stackrel{\text{is}}{=} E_GetFields(e2_1, fields2) \\
b1 := fields1 = fields2 \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETITEM

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} E_GetItem(e1_1, i1) \quad e2 \stackrel{\text{is}}{=} E_GetItem(e2_1, i2) \\
b1 := i1 = i2 \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_LITERAL

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} E_Literal(v1) \quad e2 \stackrel{\text{is}}{=} E_Literal(v2) \\
\text{literal_equal}(v1, v2) \xrightarrow{\text{type}} b \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_PATTERN

$$\begin{array}{c}
\text{ast_label}(e1) = E_Pattern \wedge \text{ast_label}(e2) = E_Pattern \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{FALSE}
\end{array}$$

E_RECORD

$$\begin{array}{c}
\text{ast_label}(e1) = E_Record \wedge \text{ast_label}(e2) = E_Record \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{FALSE}
\end{array}$$

E_TUPLE

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} E_Tuple(l1) \quad e2 \stackrel{\text{is}}{=} E_Tuple(l2) \quad \text{equal_length}(l1, l2) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{FALSE} \\
i \in \text{indices}(l1) : \text{expr_equal}(\text{tenv}, l1[i], l2[i]) \xrightarrow{\text{type}} b_i \quad \#TE \\
b := \bigwedge_{i \in \text{indices}(l1)} b_i \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_UNOP

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} E_Unop(op1, e1_1) \quad e2 \stackrel{\text{is}}{=} E_Unop(op2, e2_1) \\
\text{expr_equal}(e1_1, e2_1) \xrightarrow{\text{type}} b1 \quad \#TE \\
b := (op1 = op2) \wedge b1 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{E_UNKNOWN} \\
\frac{(\text{ast_label}(\mathbf{e1}) = \text{E_Unknown} \wedge \text{ast_label}(\mathbf{e2}) = \text{E_Unknown})}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{E_ATC} \\
\frac{\begin{array}{c} \mathbf{e1} \stackrel{\text{is}}{=} \text{E_ATC}(\mathbf{e1_1}, \mathbf{t1}) \\ \mathbf{e2} \stackrel{\text{is}}{=} \text{E_ATC}(\mathbf{e2_1}, \mathbf{t2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{b2} \quad \# \text{TE} \\ \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}} \\
\\
\text{E_VAR} \\
\frac{\begin{array}{c} \mathbf{e1} \stackrel{\text{is}}{=} \text{E_Var}(\text{name1}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \text{E_Var}(\text{name2}) \\ \mathbf{b} := \text{name1} = \text{name2} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}
\end{array}$$

25.11 TypingRule.TypeEqual

The function

$$\text{type_equal}(\overbrace{\mathbf{ty}}^{\mathbf{t1}}, \overbrace{\mathbf{ty}}^{\mathbf{t2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\text{T_TypeError}}^{\# \text{TE}}$$

conservatively tests whether the type $\mathbf{t1}$ is equivalent to the type $\mathbf{t2}$ in environment tenv and yields the result in \mathbf{b} . Otherwise, the result is a type error.

25.11.1 Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of $\mathbf{t1}$ and $\mathbf{t2}$ are different;
 - * \mathbf{b} is **FALSE**.
- All of the following apply (TBOOL_TREAL_TSTRING):
 - * both $\mathbf{t1}$ and $\mathbf{t2}$ are both either `T_Bool`, `T_Real`, or `T_String`;
 - * \mathbf{b} is **TRUE**.
- All of the following apply (TINT_UNCONSTRAINED):
 - * both $\mathbf{t1}$ and $\mathbf{t2}$ are the unconstrained integer type `unconstrained_integer`;

- * `b` is **TRUE**.
- All of the following apply (`TINT_PARAMETERIZED`):
 - * `t1` is the **parameterized integer type** with identifier `i1`, that is, `T_Int(Parameterized(i1))`;
 - * `t2` is the **parameterized integer type** with identifier `i2`, that is, `T_Int(Parameterized(i2))`;
 - * `b` is **TRUE** if and only if `i1` is equal to `i2`.
- All of the following apply (`TINT_WELLCONSTRAINED`):
 - * `t1` is the well-constrained integer type with list of constraints `c1`, that is, `T_Int(WellConstrained(c1))`;
 - * `t2` is the well-constrained integer type with list of constraints `c2`, that is, `T_Int(WellConstrained(c2))`;
 - * testing whether `c1` and `c2` are equivalent in `tenv` yields `b` **//#TE**.
- All of the following apply (`TBITS`):
 - * `t1` is the bitvector type with width expression `w1` and list of bitfields `bf1`, that is, `T_Bits(w1, bf1)`;
 - * `t2` is the bitvector type with width expression `w2` and list of bitfields `bf2`, that is, `T_Bits(w2, bf2)`;
 - * testing whether `w1` and `w2` are equivalent bitwidths in `tenv` yields `b1` **//#TE**;
 - * testing whether `bf1` and `bf2` are equivalent lists of bitfields in `tenv` yields `b2` **//#TE**;
 - * `b` is **TRUE** if and only if both `b1` and `b2` are **TRUE**.
- All of the following apply (`TARRAY`):
 - * `t1` is an array type with index `l1` and element type `t1`, that is, `T_Array(l1, t1)`;
 - * `t2` is an array type with index `l2` and element type `t2`, that is, `T_Array(l2, t2)`;
 - * testing whether `l1` is equivalent to `l2` in `tenv` yields `b1` **//#TE**;
 - * testing whether `t1` is equivalent to `t2` in `tenv` yields `b2` **//#TE**;
 - * `b` is **TRUE** if and only if both `b1` and `b2` are **TRUE**.
- All of the following apply (`TNAMED`):
 - * `t1` is a named type with identifier `s1`, that is `T_Named(s1)`;
 - * `t2` is a named type with identifier `s2`, that is `T_Named(s2)`;
 - * `b` is **TRUE** if and only if `s1` is equal to `s2`.
- All of the following apply (`TENUM`):

- * $\mathbf{t1}$ is an enumeration type with identifier $\mathbf{l1}$, that is $\mathbf{T_Enum}(\mathbf{l1})$;
 - * $\mathbf{t2}$ is an enumeration type with identifier $\mathbf{l2}$, that is $\mathbf{T_Enum}(\mathbf{l2})$;
 - * \mathbf{b} is **TRUE** if and only if $\mathbf{l1}$ is equal to $\mathbf{l2}$.
- All of the following apply (**TSTRUCTURED**):
 - * L is either $\mathbf{T_Record}$ or $\mathbf{T_Exception}$;
 - * $\mathbf{t1}$ is a **structured type** with list of fields $\mathbf{fields1}$, that is $L(\mathbf{fields1})$;
 - * $\mathbf{t2}$ is a **structured type** with list of fields $\mathbf{fields2}$, that is $L(\mathbf{fields2})$;
 - * checking whether the set of field names in $\mathbf{fields1}$ is equal to the set of field names in $\mathbf{fields2}$ yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each field \mathbf{f} in the set of fields of $\mathbf{fields1}$, testing whether the type associated with \mathbf{f} in $\mathbf{fields1}$ is equivalent to the type associated with \mathbf{f} in $\mathbf{fields2}$ in \mathbf{tenv} yields $\mathbf{b_f} \text{ // \#TE}$;
 - * \mathbf{b} is **TRUE** if and only if $\mathbf{b_f}$ is **TRUE** for each field \mathbf{f} in the set of fields of $\mathbf{fields1}$.
 - All of the following apply (**TTUPLE**):
 - * $\mathbf{t1}$ is a tuple type with list of types $\mathbf{ts1}$, that is $\mathbf{T_Tuple}(\mathbf{ts1})$;
 - * $\mathbf{t2}$ is a tuple type with list of types $\mathbf{ts2}$, that is $\mathbf{T_Tuple}(\mathbf{ts2})$;
 - * checking whether the list of types $\mathbf{ts1}$ has the same length as the list of types $\mathbf{ts2}$ yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index i in the list $\mathbf{ts1}$, testing whether $\mathbf{ts1}[i]$ is equivalent to $\mathbf{ts2}[i]$ in \mathbf{tenv} yields $\mathbf{b_i} \text{ // \#TE}$;
 - * \mathbf{b} is **TRUE** if and only if $\mathbf{b_i}$ is **TRUE** for each index i in the list $\mathbf{ts1}$.

25.11.2 Formally

$$\frac{\text{DIFFERENT_LABELS} \quad \mathit{ast_label}(\mathbf{t1}) \neq \mathit{ast_label}(\mathbf{t2})}{\mathit{type_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{FALSE}}$$

$$\frac{\text{TBOOL_TREAL_TSTRING} \quad \mathit{ast_label}(\mathbf{t1}) = \mathit{ast_label}(\mathbf{t2}) \quad \mathit{ast_label}(\mathbf{t1}) \in \{\mathbf{T_Bool}, \mathbf{T_Real}, \mathbf{T_String}\}}{\mathit{type_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{TRUE}}$$

TINT_UNCONSTRAINED	$\text{type_equal}(\text{tenv}, \text{unconstrained_integer}, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{TRUE}$
TINT_PARAMETERIZED	$\frac{\text{b} := \text{i1} = \text{i2}}{\text{type_equal}(\text{tenv}, \text{T_Int}(\text{Parameterized}(\text{i1})), \text{T_Int}(\text{Parameterized}(\text{i2}))) \xrightarrow{\text{type}} \text{b}}$
TINT_WELLCONSTRAINED	$\frac{\text{constraints_equal}(\text{tenv}, \text{c1}, \text{c2}) \xrightarrow{\text{type}} \text{b} \quad \# \text{TE}}{\text{type_equal}(\text{tenv}, \text{T_Int}(\text{WellConstrained}(\text{c1})), \text{T_Int}(\text{WellConstrained}(\text{c2}))) \xrightarrow{\text{type}} \text{b}}$
TBITS	$\frac{\begin{array}{l} \text{bitwidth_equal}(\text{tenv}, \text{w1}, \text{w2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE} \\ \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{type_equal}(\text{tenv}, \text{T_Bits}(\text{w1}, \text{bf1}), \text{T_Bits}(\text{w2}, \text{bf2})) \xrightarrow{\text{type}} \text{b}}$
TARRAY	$\frac{\begin{array}{l} \text{expr_equal}(\text{tenv}, \text{l1}, \text{l2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{type_equal}(\text{tenv}, \text{T_Array}(\text{l1}, \text{t1}), \text{T_Array}(\text{l2}, \text{t2})) \xrightarrow{\text{type}} \text{b}}$
TNAMED	$\frac{\text{b} := \text{s1} = \text{s2}}{\text{type_equal}(\text{tenv}, \text{T_Named}(\text{s1}), \text{T_Named}(\text{s2})) \xrightarrow{\text{type}} \text{b}}$
TENUM	$\frac{\text{b} := \text{l1} = \text{l2}}{\text{type_equal}(\text{tenv}, \text{T_Enum}(\text{l1}), \text{T_Enum}(\text{l2})) \xrightarrow{\text{type}} \text{b}}$
TSTRUCTURED	$\frac{\begin{array}{l} L \in \{\text{T_Record}, \text{T_Exception}\} \\ \text{bool_transition}(\text{field_names}(\text{fields1}) = \text{field_names}(\text{fields2})) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{f} \in \text{field_names}(\text{fields1}) : \\ \text{type_equal}(\text{tenv}, \text{field_type}(\text{fields1}, \text{f}), \text{field_type}(\text{fields2}, \text{f})) \xrightarrow{\text{type}} \text{b}_f \quad \# \text{TE} \\ \text{b} := \bigwedge_{\text{f} \in \text{field_names}(\text{fields1})} \text{b}_f \end{array}}{\text{type_equal}(\text{tenv}, L(\text{fields1}), L(\text{fields2})) \xrightarrow{\text{type}} \text{b}}$

$$\begin{array}{c}
\text{TTUPLE} \\
\text{equal_length}(\text{ts1}, \text{ts2}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{ts1}) : \text{type_equal}(\text{tenv}, \text{ts1}[i], \text{ts2}[i]) \xrightarrow{\text{type}} \text{b}_i \parallel \text{\#TE} \\
\text{b} := \bigwedge_{i \in \text{indices}(\text{ts1})} \text{b}_i \\
\hline
\text{type_equal}(\text{tenv}, \text{T_Tuple}(\text{ts1}), \text{T_Tuple}(\text{ts2})) \xrightarrow{\text{type}} \text{b}
\end{array}$$

25.12 TypingRule.BitwidthEqual

The function

$$\text{bitwidth_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{w1}}, \overbrace{\text{expr}}^{\text{w2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the bitwidth expression **w1** is equivalent to the bitwidth expression **w2** in environment **tenv** and yields the result in **b**. Otherwise, the result is a type error.

25.12.1 Prose

Testing whether the expressions **w1** and **w2** are equivalent in **tenv** yields **b**//**\#TE**.

25.12.2 Formally

$$\frac{\text{expr_equal}(\text{tenv}, \text{w1}, \text{w2}) \xrightarrow{\text{type}} \text{b} \parallel \text{\#TE}}{\text{bitwidth_equal}(\text{tenv}, \text{w1}, \text{w2}) \xrightarrow{\text{type}} \text{b}}$$

25.13 TypingRule.BitFieldsEqual

The function

$$\text{bitfields_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bf1}}, \overbrace{\text{bitfield}^*}^{\text{bf2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the list of bitfields **bf1** is equivalent to the list of bitfields **bf2** in environment **tenv** and yields the result in **b**. Otherwise, the result is a type error.

25.13.1 Prose

One of the following applies:

- All of the following apply (**DIFFERENT_LENGTHS**):
 - * the number of bitfields in **bf1** is different from the number of bitfields in **bf2**;

- * b is **FALSE**.
- All of the following apply (SAME_LENGTHS):
 - * the number of bitfields in $\mathbf{bf1}$ is the same as the number of bitfields in $\mathbf{bf2}$;
 - * testing whether the bitfield $\mathbf{bf1}[i]$ is equivalent to $\mathbf{bf2}[i]$ in \mathbf{tenv} for every index of $\mathbf{bf1}$ yields b_i **#TE**;
 - * b is **TRUE** if and only if b_i is **TRUE** for every index of $\mathbf{bf1}$.

25.13.2 Formally

$$\begin{array}{c}
 \text{DIFFERENT_LENGTHS} \\
 \frac{\text{equal_length}(\mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \text{FALSE}}{\text{bitfields_equal}(\mathbf{tenv}, \mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SAME_LENGTHS} \\
 \frac{\begin{array}{c} \text{equal_length}(\mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \text{TRUE} \\ i \in \text{indices}(\mathbf{bf1}) : \text{bitfield_equal}(\mathbf{tenv}, \mathbf{bf1}[i], \mathbf{bf2}[i]) \xrightarrow{\text{type}} b_i \\ b := \bigwedge_{i \in \text{indices}(\mathbf{bf1})} b_i \end{array}}{\text{bitfields_equal}(\mathbf{tenv}, \mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} b}
 \end{array}$$

25.14 TypingRule.BitFieldEqual

The function

$$\text{bitfield_equal}(\overbrace{\text{SE}}^{\mathbf{tenv}}, \overbrace{\text{bitfield}}^{\mathbf{bf1}}, \overbrace{\text{bitfield}}^{\mathbf{bf2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the bitfield $\mathbf{bf1}$ is equivalent to the bitfield $\mathbf{bf2}$ in environment \mathbf{tenv} and yields the result in b . Otherwise, the result is a type error.

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of $\mathbf{bf1}$ and $\mathbf{bf2}$ are different;
 - * b is **FALSE**.
- All of the following apply (BITFIELD_SIMPLE):
 - * $\mathbf{bf1}$ is a simple bitfield with name $\mathbf{name1}$ and list of slices $\mathbf{slices1}$, that is, `BitField.Simple(name1, slices1)`;
 - * $\mathbf{bf2}$ is a simple bitfield with name $\mathbf{name2}$ and list of slices $\mathbf{slices2}$, that is, `BitField.Simple(name2, slices2)`;
 - * checking whether $\mathbf{name1}$ is equal to $\mathbf{name2}$ yields $b1$;

- * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b2//#TE`;
- * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (`BITFIELD_NESTED`):

- * `bf1` is a nested bitfield with name `name1`, list of slices `slices1`, and nested bitfields `bf1_1`, that is, `BitField_Nested(name1, slices1, bf1_1)`;
- * `bf2` is a nested bitfield with name `name2`, list of slices `slices2`, and nested bitfields `bf2_1`, that is, `BitField_Nested(name2, slices2, bf2_1)`;
- * checking whether `name1` is equal to `name2` yields `b1`;
- * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b2//#TE`;
- * testing whether the bitfields `bf1_1` and `bf2_1` are equivalent in `tenv` yields `b2//#TE`;
- * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- All of the following apply (`BITFIELD_TYPED`):

- * `bf1` is a typed bitfield with name `name1`, list of slices `slices1`, and type `t1`, that is, `BitField_Type(name1, slices1, t1)`;
- * `bf2` is a typed bitfield with name `name2`, list of slices `slices2`, and type `t2`, that is, `BitField_Type(name2, slices2, t2)`;
- * checking whether `name1` is equal to `name2` yields `TRUE//FALSE`;
- * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b1//#TE`;
- * testing whether the types `t1` and `t2` are equivalent in `tenv` yields `b2//#TE`;
- * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

25.14.1 Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{\text{ast_label}(\text{bf1}) \neq \text{ast_label}(\text{bf2})}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{BITFIELD_SIMPLE} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Simple}(\text{name1}, \text{slices1}) \\ \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Simple}(\text{name2}, \text{slices2}) \quad \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{b1} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{BITFIELD_NESTED} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Nested}(\text{name1}, \text{slices1}, \text{bf1_1}) \\ \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bf2_1}) \\ \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE}, \\ \text{bitfields_equal}(\text{tenv}, \text{bf1_1}, \text{bf2_1}) \xrightarrow{\text{type}} \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{BITFIELD_TYPED} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Type}(\text{name1}, \text{slices1}, \text{t1}) \quad \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Type}(\text{name2}, \text{slices2}, \text{t2}) \\ \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}}
\end{array}$$

25.15 TypingRule.ConstraintsEqual

The function

$$\text{constraints_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

conservatively tests whether the constraint list `cs1` is equivalent to the constraint list `cs2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

25.15.1 Prose

All of the following apply:

- checking whether the number of constraints in `cs1` is the same as the number of constraints in `cs2` yields `TRUE`/`FALSE`;

- testing whether the constraint $\text{cs1}[i]$ is equivalent to the constraint $\text{cs2}[i]$ in tenv yields b_i for each index i in the indices for cs1 ($i \in \text{indices}(\text{cs1})$) \#TE ;
- b is **TRUE** if and only if all b_i are **TRUE** for each index i in the indices for cs1 .

25.15.2 Formally

$$\begin{array}{c}
 \text{equal_length}(\text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
 i \in \text{indices}(\text{cs1}) : \text{constraint_equal}(\text{tenv}, \text{cs1}[i], \text{cs2}[i]) \xrightarrow{\text{type}} \text{b}_i \parallel \text{\#TE} \\
 \text{b} := \bigwedge_{i \in \text{indices}(\text{cs1})} \text{b}_i \\
 \hline
 \text{constraints_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{b}
 \end{array}$$

25.16 TypingRule.ConstraintEqual

The function

$$\text{constraint_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^{\text{c1}}, \overbrace{\text{int_constraint}}^{\text{s2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the constraint c1 is equivalent to the constraint c2 in environment tenv and yields the result in b . Otherwise, the result is a type error.

25.16.1 Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of c1 and c2 are different;
 - * define b as **FALSE**.
- All of the following apply (CONSTRAINT_EXACT):
 - * c1 is an exact constraint with subexpression e1 , that is, $\text{Constraint_Exact}(\text{e1})$;
 - * c2 is an exact constraint with subexpression e2 , that is, $\text{Constraint_Exact}(\text{e2})$;
 - * applying *expr_equal* to e1 and e2 yields b \#TE .
- All of the following apply (CONSTRAINT_RANGE):
 - * c1 is a range constraint with subexpressions e1_1 and e1_2 , that is, $\text{Constraint_Range}(\text{e1_1}, \text{e1_2})$;
 - * c2 is a range constraint with subexpressions e2_1 and e2_2 , that is, $\text{Constraint_Range}(\text{e2_1}, \text{e2_2})$;
 - * applying *expr_equal* to e1_1 and e2_1 yields b1 \#TE ;
 - * applying *expr_equal* to e1_2 and e2_2 yields b2 \#TE ;
 - * define b as **TRUE** if and only if both b1 and b2 are **TRUE**.

25.16.2 Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{\text{ast_label}(\mathbf{c1}) \neq \text{ast_label}(\mathbf{c2})}{\text{constraint_equal}(\text{tenv}, \mathbf{c1}, \mathbf{c2}) \xrightarrow{\text{type}} \mathbf{FALSE}} \\
\\
\text{CONSTRAINT_EXACT} \\
\frac{\begin{array}{c} \mathbf{c1} \stackrel{\text{is}}{=} \text{Constraint_Exact}(\mathbf{e1}) \\ \mathbf{c2} \stackrel{\text{is}}{=} \text{Constraint_Exact}(\mathbf{e2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \quad \# \text{TE} \end{array}}{\text{constraint_equal}(\text{tenv}, \mathbf{c1}, \mathbf{c2}) \xrightarrow{\text{type}} \mathbf{b}} \\
\\
\text{CONSTRAINT_RANGE} \\
\frac{\begin{array}{c} \mathbf{bf1} \stackrel{\text{is}}{=} \text{Constraint_Range}(\mathbf{e1_1}, \mathbf{e1_2}) \\ \mathbf{bf2} \stackrel{\text{is}}{=} \text{Constraint_Range}(\mathbf{e2_1}, \mathbf{e2_2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \quad \# \text{TE} \\ \text{expr_equal}(\text{tenv}, \mathbf{e1_2}, \mathbf{e2_2}) \xrightarrow{\text{type}} \mathbf{b2} \quad \# \text{TE} \\ \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2} \end{array}}{\text{constraint_equal}(\text{tenv}, \mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \mathbf{b}}
\end{array}$$

25.17 TypingRule.SlicesEqual

The function

$$\text{slices_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices1}}, \overbrace{\text{slice}^*}^{\text{slices2}}) \longrightarrow \overbrace{\{\mathbf{TRUE}, \mathbf{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\mathbf{TypeError}}^{\# \text{TE}}$$

conservatively tests whether the list of slices `slices1` is equivalent to the list of slices `slices2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

25.17.1 Formally

One of the following applies:

- All of the following apply (`DIFFERENT_LENGTHS`):
 - * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields `FALSE`;
 - * `b` is `FALSE`.
- All of the following apply (`SAME_LENGTHS`):
 - * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields `TRUE`;
 - * determining whether the expression `slices1[i]` is equivalent to `slices2[i]` in `tenv` for each index in the indices for `slices1` ($i \in \text{indices}(\text{slices1})$) yields $\mathbf{b_i} \# \text{TE}$;
 - * `b` is `TRUE` if and only if all $\mathbf{b_i}$ are `TRUE` for each index in the indices for `slices1`.

25.17.2 Formally

$$\begin{array}{c}
\text{DIFFERENT_LENGTHS} \\
\frac{\text{equal_length}(\text{lices1}, \text{lices2}) \xrightarrow{\text{type}} \text{FALSE}}{\text{slices_equal}(\text{tenv}, \text{lices1}, \text{lices2}) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{SAME_LENGTHS} \\
\frac{
\begin{array}{c}
\text{equal_length}(\text{lices1}, \text{lices2}) \xrightarrow{\text{type}} \text{TRUE} \\
i \in \text{indices}(\text{lices1}) : \text{slices_equal}(\text{tenv}, \text{lices1}[i], \text{lices2}[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\
b := \bigwedge_{i \in \text{indices}(\text{lices1})} b_i
\end{array}
}{\text{slices_equal}(\text{tenv}, \text{lices1}, \text{lices2}) \xrightarrow{\text{type}} b}
\end{array}$$

25.18 TypingRule.SliceEqual

The function

$$\text{slices_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\text{slice1}}, \overbrace{\text{slice}}^{\text{slice2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

conservatively tests whether the slice `slice1` is equivalent to the slice `slice2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

25.18.1 Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * `slice1` and `slice2` have different AST labels;
 - * `b` is `FALSE`.
- All of the following apply (SLICE_SINGLE):
 - * `slice1` is a slice for a single position, given by the expression `e1`, that is, `Slice_Single(e1)`;
 - * `slice2` is a slice for a single position, given by the expression `e2`, that is, `Slice_Single(e2)`;
 - * testing `e1` and `e2` for equivalence yields `b // #TE`.
- All of the following apply (SLICE_RANGE):
 - * `slice1` is a slice for a range of positions, given by the expressions `e1.1` and `e1.2`, that is, `Slice_Range(e1.1, e1.2)`;
 - * `slice2` is a slice for a range of positions, given by the expressions `e2.1` and `e2.2`, that is, `Slice_Range(e2.1, e2.2)`;

- * testing `e1_1` and `e2_1` for equivalence yields `b1` *//* `#TE`;
 - * testing `e1_2` and `e2_2` for equivalence yields `b2` *//* `#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.
- All of the following apply (`SLICE_LENGTH`):
 - * `slice1` is a slice for a range of positions, given by the start expression `e1_1` and length expression `e1_2`, that is, `Slice.Length(e1_1, e1_2)`;
 - * `slice2` is a slice for a range of positions, given by the start expression `e2_1` and length expression `e2_2`, that is, `Slice.Length(e2_1, e2_2)`;
 - * testing `e1_1` and `e2_1` for equivalence yields `b1` *//* `#TE`;
 - * testing `e1_2` and `e2_2` for equivalence yields `b2` *//* `#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

25.18.2 Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABEL} \\
 \frac{\text{ast_label}(\text{slice1}) \neq \text{ast_label}(\text{slice2})}{\text{lices_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SLICE_SINGLE} \\
 \frac{\text{expr_equal}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{b} \text{ // } \text{\#TE}}{\text{lices_equal}(\text{tenv}, \text{Slice.Single}(\text{e1}), \text{Slice.Single}(\text{e2})) \xrightarrow{\text{type}} \text{b}} \\
 \\
 \text{SLICE_RANGE} \\
 \frac{\begin{array}{c} \text{expr_equal}(\text{tenv}, \text{e1_1}, \text{e2_1}) \xrightarrow{\text{type}} \text{b1} \text{ // } \text{\#TE} \\ \text{expr_equal}(\text{tenv}, \text{e2_1}, \text{e2_2}) \xrightarrow{\text{type}} \text{b2} \text{ // } \text{\#TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{lices_equal}(\text{tenv}, \text{Slice.Range}(\text{e1_1}, \text{e1_2}), \text{Slice.Range}(\text{e2_1}, \text{e2_2})) \xrightarrow{\text{type}} \text{b}} \\
 \\
 \text{SLICE_LENGTH} \\
 \frac{\begin{array}{c} \text{expr_equal}(\text{tenv}, \text{e1_1}, \text{e2_1}) \xrightarrow{\text{type}} \text{b1} \text{ // } \text{\#TE} \\ \text{expr_equal}(\text{tenv}, \text{e2_1}, \text{e2_2}) \xrightarrow{\text{type}} \text{b2} \text{ // } \text{\#TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{lices_equal}(\text{tenv}, \text{Slice.Length}(\text{e1_1}, \text{e1_2}), \text{Slice.Length}(\text{e2_1}, \text{e2_2})) \xrightarrow{\text{type}} \text{b}}
 \end{array}$$

25.19 TypingRule.ArrayLengthEqual

The function

$$\text{array_length_equal}(\overbrace{\text{array_index}}^{11}, \overbrace{\text{array_index}}^{12}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{\texttt{TypeError}}}^{\text{\#TE}}$$

tests whether the array lengths 11 and 12 are equivalent and yields the result in `b`. Otherwise, the result is a type error.

25.19.1 Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * 11 and 12 have different AST labels;
 - * b is **FALSE**.
- All of the following apply (EXPR_EXPR):
 - * 11 is an integer type length expression with subexpression **e1_1**, that is, `ArrayLength_Expr(e1_1)`;
 - * 12 is an integer type length expression with subexpression **e2_1**, that is, `ArrayLength_Expr(e2_1)`;
 - * testing whether **e1_1** and **e2_1** are equivalent in **tenv** yields **b** *//* **#TE**.
- All of the following apply (ENUM_ENUM):
 - * 11 is an enumeration type length expression over the enumeration **s1**, that is, `ArrayLength_Enum(s1, _)`;
 - * 12 is an enumeration type length expression over the enumeration **s2**, that is, `ArrayLength_Enum(s2, _)`;
 - * b is **TRUE** if and only if **s1** is equal to **s2**.

25.19.2 Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABELS} \\
 \frac{\text{ast_label}(11) \neq \text{ast_label}(12)}{\text{array_length_equal}(11, 12) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{EXPR_EXPR} \\
 \frac{\text{expr_equal}(\mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b} \text{ // } \mathbf{\#TE}}{\text{array_length_equal}(\text{ArrayLength_Expr}(\mathbf{e1_1}), \text{ArrayLength_Expr}(\mathbf{e2_1})) \xrightarrow{\text{type}} \mathbf{b}} \\
 \\
 \text{ENUM_ENUM} \\
 \frac{\mathbf{b} := \mathbf{s1} = \mathbf{s2}}{\text{array_length_equal}(\text{ArrayLength_Enum}(\mathbf{s1}, _), \text{ArrayLength_Enum}(\mathbf{s2}, _)) \xrightarrow{\text{type}} \mathbf{b}}
 \end{array}$$

25.20 TypingRule.LiteralEqual

The function

$$\text{literal_equal}(\overbrace{\text{literal}}^{v1}, \overbrace{\text{literal}}^{v2}) \rightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b$$

tests whether literal **v1** is **v2** by equating them.

25.20.1 Prose

b is **TRUE** if and only if $v1$ is equal to $v2$.

25.20.2 Formally

$$\frac{b := v1 = v2}{\text{literal_equal}(v1, v2) \xrightarrow{\text{type}} b}$$

25.21 TypingRule.ReduceIR

The function

$$\text{reduce_ir}(\overbrace{\text{polynomial}}^p) \longrightarrow \overbrace{\text{polynomial}}^{\text{new_p}}$$

simplifies the polynomial p , yielding the simplified polynomial new_p .

25.21.1 Prose

All of the following apply:

- p is **Sum**(f) where f binds monomials to their integer coefficients;
- g is the restriction of f to the bindings where the coefficients are non-zero;
- new_p is **Sum**(g).

25.21.2 Formally

$$\frac{g := f|_{\{\mathfrak{m} \in \text{dom}(f) \mid f(\mathfrak{m}) \neq 0\}}}{\text{reduce_ir}(\overbrace{\text{Sum}(f)}^p) \xrightarrow{\text{type}} \overbrace{\text{Sum}(g)}^{\text{new_p}}}$$

25.22 TypingRule.PolynomialToExpr

The function

$$\text{polynomial_to_expr}(\overbrace{\text{polynomial}}^p) \xrightarrow{\text{type}} \overbrace{\text{expr}}^e$$

transforms a polynomial p into the corresponding expression e .

25.22.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * p is the polynomial with an empty list of monomials, that is, $\text{Sum}(\emptyset_\lambda)$;
 - * define e as the literal expression for 0.
- All of the following apply (NON_EMPTY):
 - * p is the polynomial $\text{Sum}(f)$;
 - * sorting (see *sort* for details) the graph of f (see *func_graph* for details) yields **monoms** — a list consisting of pairs of unitary monomials and rationals. In principle, any total order of the graph of f is acceptable for sorting. The function *compare_monomial_bindings* provides one such way of ordering the graph of f ;
 - * transforming **monoms** to an expression and sign via *monomials_to_expr* yields the expression $e1$ and sign $s1$;
 - * define e as $e1$ if $s1$ is 1, the integer literal expression for 0 if $s1$ is 0, and the unary expression negating $e1$, that is, $\text{E_Unop}(\text{NEG}, e1)$, if $s1$ is -1 .

25.22.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{polynomial_to_expr}(\overbrace{\text{Sum}(\emptyset_\lambda)}^p) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int}(0))}^e \\
 \\
 \text{NON_EMPTY} \\
 \text{sort}(\text{func_graph}(f), \text{compare_monomial_bindings}) = \text{monoms} \\
 \text{monomials_to_expr}(\text{monoms}) \xrightarrow{\text{type}} (e1, s1) \quad e := \begin{cases} \text{E_Literal}(\text{L_Int}(0)) & \text{if } s1 = 0 \\ e1 & \text{if } s1 = 1 \\ \text{E_Unop}(\text{NEG}, e1) & \text{if } s1 = -1 \end{cases} \\
 \hline
 \text{polynomial_to_expr}(\overbrace{\text{Sum}(f)}^p) \xrightarrow{\text{type}} e
 \end{array}$$

25.23 TypingRule.CompareMonomialBindings

The function

$$\text{compare_monomial_bindings}(\overbrace{(\text{monomial} \times \mathbb{Q})}^{m1, q1}, \overbrace{(\text{monomial} \times \mathbb{Q})}^{m2, q2}) \longrightarrow \overbrace{\{-1, 0, 1\}}^s$$

compares two monomial bindings given by $(m1, q1)$ and $(m2, q2)$ and yields in s -1 to mean that the first monomial binding should be ordered before the second, 0 to mean that any ordering of the monomial bindings is acceptable, and 1 to mean that the second monomial binding should be ordered before the first.

25.23.1 Prose

One of the following applies:

- All of the following apply (EQUAL_MONOMIALS):
 - * $m1$ is $\text{Prod}(f)$ and $m2$ is $\text{Prod}(g)$;
 - * f is equal to g ;
 - * s is the sign of $q2 - q1$.
- All of the following apply (DIFFERENT_MONOMIALS):
 - * $m1$ is $\text{Prod}(f)$ and $m2$ is $\text{Prod}(g)$;
 - * f is different from g ;
 - * ids is the list obtained by taking the set of identifiers in the domain of f and in the domain of g , and sorting them according to the lexical order for identifiers (ASCII string order);
 - * v is the first identifier in ids for which f and g behave differently (either one of them is defined for v and the other is not, or they both bind v to a different value);
 - * s is determined as follows: 1 if v is not in the domain of f and is in the domain of g ; -1 if v is not in the domain of g and is in the domain of f ; otherwise it is the sign of $g(v) - f(v)$.

25.23.2 Formally

The function *compare_identifier* compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

$$\begin{array}{c}
 \text{EQUAL_MONOMIALS} \\
 \hline
 \frac{f = g \quad s := \text{sign}(q2 - q1)}{\text{compare_monomial_bindings}(\overbrace{(\text{Prod}(f), q1)}^{m1}, \overbrace{(\text{Prod}(g), q2)}^{m2}) \xrightarrow{\text{type}} s} \\
 \\
 \text{DIFFERENT_MONOMIALS} \\
 \frac{
 \begin{array}{l}
 f \neq g \quad ids := \text{sort}(\text{dom}(f) \cup \text{dom}(g), \text{compare_identifier}) \\
 ids \stackrel{\text{is}}{=} ids1 + ids2 \quad i \in \text{indices}(ids1) : f(ids1[i]) = g(ids1[i]) \\
 v := ids2[1] \quad s := \begin{cases} 1 & f(v) = \perp \wedge g(v) \neq \perp \\ -1 & f(v) \neq \perp \wedge g(v) = \perp \\ \text{sign}(g(v) - f(v)) & f(v) \neq \perp \wedge g(v) \neq \perp \end{cases}
 \end{array}
 }{\text{compare_monomial_bindings}(\overbrace{(\text{Prod}(f), q1)}^{m1}, \overbrace{(\text{Prod}(g), q2)}^{m2}) \xrightarrow{\text{type}} s}
 \end{array}$$

25.24 TypingRule.MonomialsToExpr

The function

$$\text{monomials_to_expr}(\overbrace{(\underbrace{\text{unitary_monomial}}_m \times \underbrace{\mathbb{Q}}_q)^*}_{\text{monoms}}) \longrightarrow (\overbrace{\text{expr}}^e, \overbrace{\{-1, 0, 1\}}^s)$$

transforms a list consisting of pairs of unitary monomials and rational factors **monoms** (so, general monomials), into an expression **e**, which represents the absolute value of the sum of all the monomials, and a sign value **s**, which indicates the sign of the resulting sum.

25.24.1 Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **monoms** is an empty list;
 - * **e** is the literal expression for the integer 0 and **s** is 0.
- All of the following apply (**NON_EMPTY**):
 - * **monoms** is a list with (m, q) as its **head** and **monoms1** as its **tail**;
 - * transforming the unitary monomial **m** to an expression via *unitary_monomials_to_expr* yields **e1'**;
 - * transforming **e1'** and **q** via *monomial_to_expr* yields the expression **e1** and sign **s1**;
 - * transforming **monoms** to an expression and sign via *monomials_to_expr* yields **(e2, s2)**;
 - * symbolically adding **e1, s1, e2, s2** via *sym_add_expr* yields **(e, s)**.

25.24.2 Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{monomials_to_expr}(\overbrace{[\]}_{\text{monoms}}) \xrightarrow{\text{type}} (\overbrace{\text{E_Literal(L_Int(0))}}^e, \overbrace{0}^s) \\
 \\
 \text{NON_EMPTY} \\
 \frac{
 \begin{array}{l}
 \text{unitary_monomials_to_expr}(m) \xrightarrow{\text{type}} e1' \quad \text{monomial_to_expr}(e1', q) \xrightarrow{\text{type}} (e1, s1) \\
 \text{monomials_to_expr}(\text{monoms}) \xrightarrow{\text{type}} (e2, s2) \quad \text{sym_add_expr}(e1, s1, e2, s2) \xrightarrow{\text{type}} (e, s)
 \end{array}
 }{
 \text{monomials_to_expr}(\overbrace{[(m, q)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} (e, s)
 }
 \end{array}$$

25.25 TypingRule.MonomialToExpr

The function

$$\text{monomial_to_expr}(\overbrace{\text{expr}}^{\mathbf{e}}, \overbrace{\mathbb{Q}}^q) \longrightarrow (\overbrace{\text{expr}}^{\text{new_e}} \times \overbrace{\{-1, 0, 1\}}^{\mathbf{s}})$$

transforms an expression \mathbf{e} and rational q into the expression new_e , which represents the absolute value of \mathbf{e} multiplied by q , and the sign of q as \mathbf{s} .

25.25.1 Prose

One of the following applies:

- All of the following apply (Q_ZERO):
 - * q is 0;
 - * new_e is the literal expression for 0;
 - * \mathbf{s} is 0.
- All of the following apply (Q_NATURAL):
 - * q a strictly positive;
 - * symbolically multiplying the literal expression for q and \mathbf{e} via *sym_mul_expr* yields new_e ;
 - * \mathbf{s} is 1.
- All of the following apply (Q_POSITIVE_FRACTION):
 - * q a strictly positive fraction, that is, not an integer;
 - * the reduced representation of the fraction q is $\frac{d}{n}$;
 - * symbolically multiplying the literal expression for q and \mathbf{e} via *sym_mul_expr* yields $\mathbf{e2}$;
 - * \mathbf{e} is the binary expression with operator DIV and operands $\mathbf{e2}$ and the literal expression for n ;
 - * \mathbf{s} is 1.
- All of the following apply (Q_NEGATIVE):
 - * q a strictly negative;
 - * transforming \mathbf{e} with $-q$ to an expression and a sign via *monomial_to_expr* yields $(\text{new_e}, 1)$;
 - * \mathbf{s} is -1 .

25.25.2 Formally

$$\begin{array}{c}
\text{Q_ZERO} \\
\hline
q = 0 \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\overbrace{\text{E_Literal}(\text{L_Int}(0))}^{\text{new_e}}, \overbrace{0}^{\mathbf{s}}) \\
\\
\text{Q_NATURAL} \\
q > 0 \quad q \in \mathbb{N} \quad \text{sym_mul_expr}(\text{E_Literal}(\text{L_Int}(q)), \mathbf{e}) \xrightarrow{\text{type}} \text{new_e} \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{1}^{\mathbf{s}}) \\
\\
\text{Q_POSITIVE_FRACTION} \\
q > 0 \quad q \notin \mathbb{N} \\
q \stackrel{\text{is}}{=} \frac{d}{n} \quad \text{is the reduced fraction for } q \\
\text{sym_mul_expr}(\text{E_Literal}(\text{L_Int}(d)), \mathbf{e}) \xrightarrow{\text{type}} \mathbf{e2} \\
\text{new_e} := \text{E_Binop}(\text{DIV}, \mathbf{e2}, \text{E_Literal}(\text{L_Int}(n))) \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{1}^{\mathbf{s}}) \\
\\
\text{Q_NEGATIVE} \\
q < 0 \quad \text{monomial_to_expr}(\mathbf{e}, -q) \xrightarrow{\text{type}} (\text{new_e}, 1) \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{-1}^{\mathbf{s}})
\end{array}$$

25.26 TypingRule.SymAddExpr

The function

$$\text{sym_add_expr}(\overbrace{\text{expr}}^{\mathbf{e1}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s1}}, \overbrace{\text{expr}}^{\mathbf{e2}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s2}}) \xrightarrow{\text{type}} (\overbrace{\text{expr}}^{\mathbf{e}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s}})$$

symbolically sums the expressions $\mathbf{e1}$ and $\mathbf{e2}$ with respective signs $\mathbf{s1}$ and $\mathbf{s2}$ yielding the expression \mathbf{e} and sign \mathbf{s} .

The effect of the function can be summarized by the following table:

	s1		
s2	-1	0	1
-1	($\mathbf{e1} + \mathbf{e2}, \mathbf{s1}$)	($\mathbf{e2}, \mathbf{s2}$)	($\mathbf{e1} - \mathbf{e2}, \mathbf{s1}$)
0	($\mathbf{e1}, \mathbf{s1}$)	($\mathbf{e1}, \mathbf{s1}$)	($\mathbf{e1}, \mathbf{s1}$)
1	($\mathbf{e1} - \mathbf{e2}, \mathbf{s1}$)	($\mathbf{e2}, \mathbf{s2}$)	($\mathbf{e1} + \mathbf{e2}, \mathbf{s1}$)

25.26.1 Prose

One of the following applies:

- All of the following apply (ZERO):
 - * either **s1** is 0 or **s2** is 0;
 - * the result is (**e2**, **s2**) if **s1** is 0 and (**e1**, **s1**), otherwise.
- All of the following apply (SAME_SIGN):
 - * both **s1** and **s2** are not 0;
 - * **s1** is equal to **s2**;
 - * **e** is the binary expression with operator PLUS and operands **e1** and **e2**, that is, E_Binop(PLUS, **e1**, **e2**);
 - * **s** is **s1**;
- All of the following apply (SAME_SIGN):
 - * both **s1** and **s2** are not 0;
 - * **s1** is different from **s2**;
 - * **e** is the binary expression with operator MINUS and operands **e1** and **e2**, that is, E_Binop(MINUS, **e1**, **e2**);
 - * **s** is **s1**;

25.26.2 Formally

$$\begin{array}{c}
 \text{ZERO} \\
 \frac{(\mathbf{s1} = 0 \vee \mathbf{s2} = 0) \quad (\mathbf{e}, \mathbf{s}) := \text{choice}(\mathbf{s1} = 0, (\mathbf{e2}, \mathbf{s2}), (\mathbf{e1}, \mathbf{s1}))}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{s1}, \mathbf{e2}, \mathbf{s2}) \xrightarrow{\text{type}} (\mathbf{e}, \mathbf{s})} \\
 \\
 \text{SAME_SIGN} \\
 \frac{\mathbf{s1} \neq 0 \wedge \mathbf{s2} \neq 0 \quad \mathbf{s1} = \mathbf{s2}}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{PLUS}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}, \overbrace{\mathbf{s1}}^{\mathbf{s}})} \\
 \\
 \text{DIFFERENT_SIGNS} \\
 \frac{\mathbf{s1} \neq 0 \wedge \mathbf{s2} \neq 0 \quad \mathbf{s1} \neq \mathbf{s2}}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{MINUS}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}, \overbrace{\mathbf{s1}}^{\mathbf{s}})}
 \end{array}$$

25.27 TypingRule.UnitaryMonomialsToExpr

The function

$$\text{unitary_monomials_to_expr}(\overbrace{(\text{identifier} \times \mathbb{N})^*}^{\text{monoms}}) \longrightarrow \overbrace{\text{expr}}^{\mathbf{e}}$$

transforms a list of single-variable unitary monomials **monoms** into an expression **e**. Intuitively, **monoms** represented a multiplication of the single-variable unitary monomials.

25.27.1 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `monoms` is the empty list;
 - * `e` is the literal expression for 1.
- All of the following apply (EXP_ZERO):
 - * `monoms` is a list where the first element is $(v, 0)$ and its tail is `monoms`;
 - * transforming `monoms1` to an expression yields `e`.
- All of the following apply (EXP_ONE):
 - * `monoms` is a list where the first element is $(v, 1)$ and its tail is `monoms`;
 - * `e1` is the variable expression for `v`;
 - * transforming `monoms1` to an expression yields `e2`;
 - * symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.
- All of the following apply (EXP_TWO):
 - * `monoms` is a list where the first element is $(v, 2)$ and its tail is `monoms`;
 - * `e1` is the binary expression with operator `MUL` and operands `E.Var(v)` and `E.Var(v)` (that is, `v` squared);
 - * transforming `monoms1` to an expression yields `e2`;
 - * symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.
- All of the following apply (EXP_GT_TWO):
 - * `monoms` is a list where the first element is (v, n) and its tail is `monoms`;
 - * `n` is greater than 1;
 - * `e1` is the binary expression with operator `POW` and base operand being the variable expression for `v` and the exponent operand being the variable expression for `n`;
 - * transforming `monoms1` to an expression yields `e2`;
 - * symbolically multiplying `e1` and `e2` via *sym_mul_expr* yields `e`.

25.27.2 Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{unitary_monomials_to_expr}(\overbrace{[\]}^{\text{monoms}}) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int}(1))}^{\text{e}} \\
\\
\text{EXP_ZERO} \\
\frac{\text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 0)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}} \\
\\
\text{EXP_ONE} \\
\frac{\text{e1} := \text{E_Var}(v) \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e2} \quad \text{sym_mul_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 1)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}} \\
\\
\text{EXP_TWO} \\
\frac{\text{e1} := \overbrace{\text{E_Var}(v) \text{ MUL } \text{E_Var}(v)}^{\text{E_Binop}} \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e2} \quad \text{sym_mul_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 2)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}} \\
\\
\text{EXP_GT_TWO} \\
\frac{n \geq 2 \quad \text{e1} := \overbrace{\text{E_Var}(v) \text{ POW } \text{E_Literal}(n)}^{\text{E_Binop}} \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e2} \quad \text{sym_mul_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, n)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}}
\end{array}$$

25.28 TypingRule.SymMulExpr

The function $\text{sym_mul_expr}(\overbrace{\text{expr}}^{\text{e1}}, \overbrace{\text{expr}}^{\text{e2}}) \xrightarrow{\text{type}} \overbrace{\text{expr}}^{\text{e}}$ produces an expression representing the multiplication of expressions **e1** and **e2**, simplifying away the case where one of the operands is the literal one.

25.28.1 Prose

One of the following applies:

- All of the following apply (**ONE_OPERAND**):
 - * either **e1** or **e2** is the literal expression for 1;
 - * **e** is **e2** if **e1** is the literal expression for 1 and **e1**, otherwise.

25.28.2 Formally

$$\begin{array}{c}
\text{ONE_OPERAND} \\
(e1 = \text{E_Literal}(\text{L_Int}(1)) \vee e2 = \text{E_Literal}(\text{L_Int}(1))) \\
e := \text{choice}(e1 = \text{E_Literal}(\text{L_Int}(1)), e2, e1) \\
\hline
\text{sym_mul_expr}(\text{E_Binop}(\text{MUL}, e1, e2)) \xrightarrow{\text{type}} e
\end{array}$$

$$\begin{array}{c}
\text{NO_ONE_OPERAND} \\
(e1 \neq \text{E_Literal}(\text{L_Int}(1)) \wedge e2 \neq \text{E_Literal}(\text{L_Int}(1))) \quad e := \text{E_Binop}(\text{MUL}, e1, e2) \\
\hline
\text{sym_mul_expr}(\text{E_Binop}(\text{MUL}, e1, e2)) \xrightarrow{\text{type}} e
\end{array}$$

Chapter 26

Utility Rules

26.1 Checked Transitions

We define the following rules to allow us asserting that a condition holds, returning a type error otherwise:

$$\begin{array}{l} \text{CHECK_TRANS_TRUE} \\ \text{check}(\text{TRUE}, \langle \text{message} \rangle) \longrightarrow \text{TRUE} \\ \\ \text{CHECK_TRANS_FALSE} \\ \text{check}(\text{FALSE}, \langle \text{message} \rangle) \longrightarrow \text{TypeError}(\langle \text{message} \rangle) \end{array}$$

26.2 Converting a List of Pairs to a Map

The parametric function

$$\text{pairs_to_map}(\overbrace{(\text{identifier} \times T)^*}^{\text{pairs}}) \longrightarrow \overbrace{(\text{identifier} \rightarrow T)}^f \cup \text{TypeError}$$

converts a list of pairs — **pairs** — where each pair consists of an identifier and a value of type T into a function mapping each identifier to its respective value in the list. If a duplicate identifier exists in **pairs** then a type error is returned.

26.2.1 Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **pairs** is empty;
 - * f is the empty function.
- All of the following apply (**ERROR**):

- * there exist two different positions in the list where the identifier is the same;
- * the result is a type error indicating the existence of a duplicate identifier.
- All of the following apply (OKAY):
 - * all identifiers occurring in the list are unique;
 - * f is a function that associates to each identifier the value appearing with it in `pairs`.

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{pairs_to_map}([\] \xrightarrow{\text{type}} \emptyset_\lambda \\
 \\
 \text{ERROR} \\
 \frac{i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} \text{TypeError}(\text{DuplicateIdentifier})} \\
 \\
 \text{OKAY} \\
 \frac{\forall i, j \in 1..k. \text{id}_i \neq \text{id}_j \quad f := \lambda \text{id}. \begin{cases} t_i & \text{if } i \in 1..k \wedge \text{id} = \text{id}_i \\ \perp & \text{otherwise} \end{cases}}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} f}
 \end{array}$$

26.3 TypingRule.CheckNoDuplicates

The function

$$\text{check_no_duplicates}(\overbrace{\text{identifier}^*}^{\text{id}_{1..k}}) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

checks whether a non-empty list of identifiers contains a duplicate identifier. If it does not, the result is `TRUE` and otherwise the result is a type error.

26.3.1 Prose

One of the following applies:

- All of the following apply (OKAY):
 - * the set containing all identifiers in the list has the same cardinality as the length of the list;
 - * the result is `TRUE`.
- All of the following apply (ERROR):
 - * there exist two different positions in the list where the identifier is the same;
 - * the result is a type error indicating the existence of a duplicate identifier.

$$\begin{array}{c}
\text{OKAY} \\
\frac{|\{\text{id}_{1..k}\}| = k}{\text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{ERROR} \\
\frac{i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j}{\text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TypeError}(\text{DuplicateIdentifier})}
\end{array}$$

26.4 Annotating Field Initializers

The function

$$\text{annotate_field_init}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})}^{(\text{name}, \text{e}')} , \overbrace{\text{field}^*}^{\text{field_types}}) \longrightarrow \overbrace{(\text{identifier} \times \text{expr})}^{(\text{name}, \text{e}'')}$$

annotates a field initializers (name, e') in a record expression with list of fields `field_types` and returns the annotated field initializer $(\text{name}, \text{e}'')$. Otherwise, the result is a type error.

26.4.1 Prose

All of the following apply:

- annotating the expression e' in `tenv` yields $(\text{t}', \text{e}'') \text{ \#TE}$;
- One of the following applies:
 - * All of the following apply (OKAY):
 - the unique type associated with `name` in `field_types` is `t_spec'`;
 - determining whether t' *type-satisfies* `t_spec'` in `tenv` yields $\text{TRUE} \text{ \#TE}$;
 - * All of the following apply (ERROR):
 - there is no type associated with `name` in `field_types`;
 - the result is a type error indicating that the field `name` does not exist.

26.4.2 Formally

OKAY

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t', e'') \text{ // \#TE} \\ \text{field_type}(\text{field_types}, \text{name}) = t_spec' \\ \text{checked_typesat}(\text{tenv}, t', t_spec') \xrightarrow{\text{type}} \text{TRUE // \#TE} \end{array}}{\text{annotate_field_init}(\text{tenv}, (\text{name}, e'), \text{field_types}) \xrightarrow{\text{type}} (\text{name}, e'')}$$

ERROR

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t', e'') \text{ // \#TE} \\ \text{field_type}(\text{field_types}, \text{name}) = \perp \end{array}}{\text{annotate_field_init}(\text{tenv}, (\text{name}, e'), \text{field_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})}$$

26.5 TypingRule.BitFieldGetName

The function

$$\text{bitfield_get_name} : \overbrace{\text{bitfield}}^{\text{bf}} \longrightarrow \overbrace{\text{identifier}}^{\text{name}}$$

returns the name of a bitfield — `name`, given a bitfield `bf`.

26.5.1 Prose

One of the following applies:

- `b` is a simple bitfield with name `name`, that is, `BitField.Simple(name, _)`;
- `b` is a nested bitfield with name `name`, that is, `BitField.Nested(name, _, _)`;
- `b` is a typed bitfield with name `name`, that is, `BitField.Type(name, _, _)`.

26.5.2 Formally

SIMPLE

$$\text{bitfield_get_name}(\text{BitField.Simple}(\text{name}, _)) \xrightarrow{\text{type}} \text{name}$$

NESTED

$$\text{bitfield_get_name}(\text{BitField.Nested}(\text{name}, _, _)) \xrightarrow{\text{type}} \text{name}$$

TYPE

$$\text{bitfield_get_name}(\text{BitField.Type}(\text{name}, _, _)) \xrightarrow{\text{type}} \text{name}$$

26.6 TypingRule.CheckVarNotInGEnv

The function

$$\text{check_var_not_in_genv}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{id}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether `id` is already declared in the global environment of `tenv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

26.6.1 Prose

All of the following apply:

- `b` is `TRUE` if and only if one of the following applies:
 - * `id` is declared as a global identifier in `tenv`;
 - * `id` is declared as a subprogram in `tenv`;
 - * `id` is declared as a type in `tenv`.
- checking whether `b` is `FALSE` yields `TRUE` or a type error indicating that `id` has already been declared, thereby short-circuiting the rule.

26.6.2 Formally

$$\begin{array}{l} \text{b} := G^{\text{tenv}}.\text{global_storage_types}(\text{id}) \neq \perp \vee \\ \quad G^{\text{tenv}}.\text{subprograms}(\text{id}) \neq \perp \vee \\ \quad G^{\text{tenv}}.\text{declared_types}(\text{id}) \neq \perp \\ \hline \text{check}(\neg \text{b}, \text{IdentifierAlreadyDeclared}) \longrightarrow \text{TRUE} \parallel \#TE \\ \hline \text{check_var_not_in_genv}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE} \end{array}$$

26.7 TypingRule.CheckVarNotInEnv

The function

$$\text{var_in_env}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

determines whether an identifier `id` is declared in the static environment `tenv`.

The function

$$\text{check_var_not_in_env}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{id}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

checks whether `id` is already declared in `tenv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

26.7.1 Prose

$\text{var_in_env}(\text{tenv}, \text{id})$ is true if and only if one of the following applies:

- id is declared as a local identifier in tenv ;
- id is declared as a global identifier in tenv ;
- id is declared as a subprogram in tenv ;
- id is declared as a type in tenv .

26.7.2 Formally

$$\begin{array}{c}
 \text{b} := \begin{array}{l} L^{\text{tenv}}.\text{local_storage_types}(\text{id}) \neq \perp \vee \\ G^{\text{tenv}}.\text{global_storage_types}(\text{id}) \neq \perp \vee \\ G^{\text{tenv}}.\text{subprograms}(\text{id}) \neq \perp \vee \\ G^{\text{tenv}}.\text{declared_types}(\text{id}) \neq \perp \end{array} \\
 \hline
 \text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{b} \\
 \\
 \text{OKAY} \\
 \begin{array}{c} \text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{FALSE} \\ \hline \text{check_var_not_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE} \end{array} \\
 \\
 \text{ERROR} \\
 \begin{array}{c} \text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE} \\ \hline \text{check_var_not_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError(AlreadyDeclared)} \end{array}
 \end{array}$$

26.8 TypingRule.AddLocal

The function

$$\text{add_local}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

adds the identifier id as a local storage element with type ty and local declaration keyword ldk to the local environment of tenv , resulting in the static environment new_tenv .

26.8.1 Prose

All of the following apply:

- the map $\text{new_local_storagetypes}$ is defined by updating the map $\text{local_storage_types}$ of tenv with the binding id to the type ty and local declaration keyword ldk , that is, (ty, ldk) ;
- new_tenv is defined by updating the local environment with the binding of $\text{local_storage_types}$ to $\text{new_local_storagetypes}$.

26.8.2 Formally

$$\frac{\begin{array}{l} \text{new_local_storagetypes} := L^{\text{tenv}}.\text{local_storage_types}[\text{id} \mapsto (\text{ty}, \text{ldk})] \\ \text{new_tenv} := (G^{\text{tenv}}, L^{\text{tenv}}[\text{local_storage_types} \mapsto \text{new_local_storagetypes}]) \end{array}}{\text{add_local}(\text{tenv}, \text{id}, \text{ty}, \text{ldk}) \xrightarrow{\text{type}} \text{new_tenv}}$$

26.9 TypingRule.DeclaredType

The function

$$\text{declared_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \text{TypeError}$$

retrieves the type associated with the identifier `id` in the static environment `tenv`. If the identifier is not associated with a declared type, a type error is returned.

26.9.1 Prose

One of the following applies:

- All of the following apply (EXISTS):
 - * `id` is bound in the global environment to the type `t`.
- All of the following apply (TYPE_NOT_DECLARED):
 - * `id` is not bound in the global environment to any type;
 - * the result is a type error indicating the lack of a type declaration for `id`.

26.9.2 Formally

$$\frac{\begin{array}{c} \text{EXISTS} \\ G^{\text{tenv}}.\text{declared_types}(\text{id}) = \text{t} \end{array}}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{t}}$$

$$\frac{\begin{array}{c} \text{TYPE_NOT_DECLARED} \\ G^{\text{tenv}}.\text{declared_types}(\text{id}) = \perp \end{array}}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeNotDeclared})}$$

26.10 TypingRule.FindBitfieldOpt

The function

$$\text{find_bitfield_opt}(\overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}) \longrightarrow \overbrace{\langle \text{bitfield} \rangle}^{\text{r}}$$

returns the bitfield associated with the name `name` in the list of bitfields `bitfields`, if there is one. Otherwise, the result is `None`.

26.10.1 Prose

One of the following applies:

- All of the following apply (MATCH):
 - * `bitfields` starts with a bitfield `bf`;
 - * obtaining the name associated with `bf` yields `name`;
 - * the result is `bf`.
- All of the following apply (TAIL):
 - * `bitfields` starts with a bitfield `bf` and continues with the tail list `bitfields'`;
 - * obtaining the name associated with `bf` yields `name'`, which is different than `name`;
 - * finding the bitfield associated with `name` in `bitfields'` yields the result `r`.
- All of the following apply (EMPTY):
 - * `bitfields` is an empty list;
 - * the result is `None`.

$$\begin{array}{c}
 \text{MATCH} \\
 \hline
 \text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
 \hline
 \text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}}^{\text{bitfields}}) \xrightarrow{\text{type}} \overbrace{\langle \text{bf} \rangle}^{\text{r}} \\
 \\
 \text{TAIL} \\
 \hline
 \text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name}' \\
 \text{name} \neq \text{name}' \quad \text{find_bitfield_opt}(\text{name}, \text{bitfields}') \xrightarrow{\text{type}} \text{r} \\
 \hline
 \text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{r} \\
 \\
 \text{EMPTY} \\
 \hline
 \text{find_bitfield_opt}(\text{name}, \overbrace{[]}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{None}
 \end{array}$$

26.11 TypingRule.MemBfs

The function

$$\text{mem_bfs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^+}^{\text{bfs2}}, \overbrace{\text{bitfield}}^{\text{bf1}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

checks whether the bitfield `bf` exists in `bfs2` in the context of `tenv`, returning the result in `b`.

26.11.1 Prose

One of the following applies:

- All of the following apply (NONE):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **None**;
 - * **b** is **FALSE**.
- All of the following apply (SIMPLE_ANY):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a simple bitfield;
 - * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (NESTED_SIMPLE):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a simple bitfield;
 - * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (NESTED_NESTED):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a nested bitfield with name **name1**, slices **slice1**, and **bfs1**;
 - * **b1** is true if and only if **name1** is equal to **name2**;
 - * symbolically equating the slices **slices1** and **slices2** in **tenv** yields **b2**;
 - * checking **bfs1** is included in **bfs2'** in the context of **tenv** yields **b3**;
 - * **b** is defined as the conjunction of **b1**, **b2**, and **b3**.
- All of the following apply (NESTED_TYPED):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a typed bitfield;
 - * **b** is **FALSE**.

- All of the following apply (TYPED_SIMPLE):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;
 - * **bf1** is a simple bitfield;
 - * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (TYPED_NESTED):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;
 - * **bf1** is a nested bitfield;
 - * **b** is **FALSE**.
- All of the following apply (TYPED_TYPED):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;
 - * **bf1** is a typed bitfield with name **name1**, slices **slices1**, and type **ty1**;
 - * **b1** is true if and only if **name1** is equal to **name2**;
 - * symbolically equating the slices **slices1** and **slices2** in **tenv** yields **b2**;
 - * checking whether **ty1** structurally subtypes **ty2** in **tenv** yields **b3**;
 - * **b** is defined as the conjunction of **b1**, **b2**, and **b3**.

NONE

$$\frac{\text{bitfield_get_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{None}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{FALSE}}$$

SIMPLE_ANY

$$\frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{ast_label}(\text{bf2}) = \text{BitField.Simple} \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}}$$

NESTED_SIMPLE

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

NESTED_NESTED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} = \text{BitField_Nested}(\text{name1}, \text{slices1}, \text{bfs1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\
\text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}') \xrightarrow{\text{type}} \text{b3} \quad \text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

NESTED_TYPED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \quad \text{ast_label}(\text{bf1}) = \text{BitField_Type} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

TYPED_SIMPLE

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
\text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TYPED_NESTED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \quad \text{ast_label}(\text{bf1}) = \text{BitField_Nested} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

TYPED_TYPED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
\text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Type}(\text{name1}, \text{slices1}, \text{ty1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\
\text{structural_subtype_satisfies}(\text{tenv}, \text{ty1}, \text{ty2}) \xrightarrow{\text{type}} \text{b3} \quad \# \text{TE} \\
\text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

26.12 TypingRule.BitFieldsIncluded

The predicate

$$\text{bitfields_included}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bfs1}}, \overbrace{\text{bitfield}^*}^{\text{bfs2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

tests whether the set of bit fields **bfs1** is included in the set of bit fields **bfs2** in environment **tenv**, returning a type error, if one is detected.

26.12.1 Prose

All of the following apply:

- checking whether each field **bf** in **bfs1** exists in **bfs2** via *mem.bfs* yields $\text{b}_{\text{bf}} \text{\#TE}$;
- the result — **b** — is the conjunction of b_{bf} for all bitfields **bf** in **bfs1**.

$$\frac{\text{bf} \in \text{bfs1} : \text{mem.bfs}(\text{bfs2}, \text{bf}) \xrightarrow{\text{type}} \text{b}_{\text{bf}} \text{\#TE} \quad \text{bf} := \bigwedge_{\text{bf} \in \text{bfs1}} \text{b}_{\text{bf}}}{\text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}) \xrightarrow{\text{type}} \text{b}}$$

26.13 TypingRule.TypeOfArrayLength

The function

$$\text{type_of_array_length}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{array_index}}^{\text{size}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}}$$

returns the type for the array index **size** in the static environment **tenv**.

26.13.1 Prose

One of the following applies:

- All of the following apply (ENUM):
 - * **size** is an enumeration index over the enumeration **s**, that is, `ArrayLength_Enum(s, _)`;
 - * **t** is the named type for **s**, that is, `T_Named(s)`.
- All of the following apply (EXPR):
 - * **size** is an expression index for **e**, that is, `ArrayLength_Expr(e)`;
 - * applying *normalize* to simplify the expression corresponding to **e** − 1 in **tenv** yields the expression **m**;
 - * **c** is the range constraint for 0..**m**, that is, `Constraint_Range(E.Literal(0), m)`;
 - * **t** is the well-constrained integer with the single constraint **c**.

26.13.2 Formally

$$\begin{array}{c}
 \text{ENUM} \\
 \text{type_of_array_length}(\text{tenv}, \text{ArrayLength_Enum}(s, _)) \xrightarrow{\text{type}} \text{T_Named}(s) \\
 \\
 \text{EXPR} \\
 \frac{\text{normalize}(\text{tenv}, \text{E_Binop}(\text{MINUS}, e, \text{E_Literal}(1))) \xrightarrow{\text{type}} m \quad c := \text{Constraint_Range}(\text{E_Literal}(0), m)}{\text{type_of_array_length}(\text{tenv}, \text{ArrayLength_Expr}(e)) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}([c]))}
 \end{array}$$

26.14 TypingRule.CheckStructureInteger

The function

$$\text{check_structure_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \{\text{TRUE}\} \cup \text{T_TypeError}$$

returns **TRUE** if t has the **structure** an integer type and a type error otherwise.

26.14.1 Prose

One of the following applies:

- All of the following apply (OKAY):
 - * determining the **structure** of t yields $\text{t}' \text{ // } \# \text{TE}$;
 - * t' is an integer type;
 - * the result is **TRUE**;
- All of the following apply (ERROR):
 - * determining the **structure** of t yields $\text{t}' \text{ // } \# \text{TE}$;
 - * t' is not an integer type;
 - * the result is a type error indicating that t was expected to have the **structure** of an integer.

26.14.2 Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{t}' \text{ // } \# \text{TE} \quad \text{ast_label}(\text{t}') = \text{T_Int}}{\text{check_structure_integer}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{t}' \quad \text{ast_label}(\text{t}') \neq \text{T_Int}}{\text{check_structure_integer}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{ExpectedIntegerStructure})}
 \end{array}$$

26.15 TypingRule.CheckStructure

The function

$$\text{check_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{L}}^{\text{l}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

returns **TRUE** if t has the **structure** a of type corresponding to the AST label l and a type error otherwise.

26.15.1 Prose

One of the following applies:

- All of the following apply (OKAY):
 - * determining the **structure** of t yields $\text{t}' \text{ // } \# \text{TE}$;
 - * t' has the label l ;
 - * the result is **TRUE**;
- All of the following apply (ERROR):
 - * determining the **structure** of t yields $\text{t}' \text{ // } \# \text{TE}$;
 - * t' does not have the label l ;
 - * the result is a type error indicating that t was expected to have the **structure** of a type with the AST label l .

26.15.2 Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{t}' \text{ // } \# \text{TE} \quad \text{ast_label}(\text{t}') = \text{l}}{\text{check_structure}(\text{tenv}, \text{t}, \text{l}) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{get_structure}(\text{t}) \xrightarrow{\text{type}} \text{t}' \quad \text{ast_label}(\text{t}') \neq \text{l}}{\text{check_structure}(\text{tenv}, \text{t}, \text{l}) \xrightarrow{\text{type}} \text{TypeError}(\text{UnexpectedTypeStructure})}
 \end{array}$$

26.16 TypingRule.StorageIsPure

The function

$$\text{storage_is_pure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{s}}) \longrightarrow \overbrace{\text{B}}^{\text{b}} \cup \text{TTypeError}$$

b is true if and only if the identifier s corresponds to a **pure** storage element in the static environment tenv .

26.16.1 Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * s is a locally declared storage element;
 - * b is true if and only if s is declared as a constant or as an immutable variable (`let`).
- All of the following apply (GLOBAL):
 - * s is a globally declared storage element;
 - * b is true if and only if s is declared as a constant, a configuration variable, or an immutable variable.
- All of the following apply (ERROR):
 - * s is not defined in the environment as a storage element;
 - * the result is a type error indicating that s is not defined as a storage element.

26.16.2 Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = (_, \text{ldk}) \quad b := \text{ldk} \in \{\text{LDK_Constant}, \text{LDK_Let}\} \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} b \\
 \\
 \text{GLOBAL} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = (_, \text{gdk}) \\
 \quad b := \text{gdk} \in \{\text{GDK_Constant}, \text{GDK_Config}, \text{GDK_Let}\} \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} b \\
 \\
 \text{ERROR} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = \perp \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{TypeError}(\text{UndefinedIdentifier})
 \end{array}$$

26.17 TypingRule.CheckStaticallyEvaluable

The function

$$\text{check_statically_evaluable}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

returns `TRUE` if e is a `statically evaluable` expression in the static environment tenv and a type error otherwise.

26.17.1 Prose

All of the following applies:

- symbolically simplifying e in tenv via *normalize* yields $e1$;
- determining the set of used identifiers in $e1$ yields use_set ;
- b is true if and only if every identifier in use_set is pure;
- the result is **TRUE** if b is **TRUE**, otherwise it is a type error indicating that the expression is not statically evaluable.

26.17.2 Formally

$$\frac{\begin{array}{l} \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \\ \text{use_e}(e1) \xrightarrow{\text{type}} \text{use_set} \quad \text{id} \in \text{use_set} : \text{storage_is_pure}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} b_{\text{id}} \\ b := \bigwedge_{\text{id} \in \text{use_set}} b_{\text{id}} \quad \text{check}(b, \text{NotStaticallyEvaluable}) \longrightarrow \text{TRUE} \parallel \#TE \end{array}}{\text{check_statically_evaluable}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TRUE}}$$

26.18 TypingRule.ToWellConstrained

The function

$$\text{to_well_constrained}(\overset{\text{t}}{\text{ty}}) \longrightarrow \overset{\text{t}'}{\text{ty}}$$

returns the *well-constrained version* of a type $\text{t} \rightarrow \text{t}'$, which is defined as follows.

One of the following applies:

- All of the following apply ($\text{T_INT_PARAMETERIZED}$):
 - * t is a *parameterized integer type* for the variable v ;
 - * t' is the well-constrained integer constrained by the variable expression for v , that is, $\text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(\text{E_Var}(v))))$.
- All of the following apply (T_INT_OTHER , OTHER):
 - * t is not a *parameterized integer type* for the variable v ;
 - * t' is t .

26.18.1 Formally

$$\begin{array}{c} \text{T_INT_PARAMETERIZED} \\ \text{to_well_constrained}(\text{T_Int}(\text{Parameterized}(v))) \xrightarrow{\text{type}} \\ \text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(\text{E_Var}(v)))) \\ \\ \text{T_INT_OTHER} \quad \text{OTHER} \\ \frac{\text{ast_label}(i) \neq \text{Parameterized}}{\text{to_well_constrained}(\text{T_Int}(i)) \xrightarrow{\text{type}} t} \quad \frac{\text{ast_label}(t) \neq \text{T_Int}}{\text{to_well_constrained}(t) \xrightarrow{\text{type}} t} \end{array}$$

26.19 TypingRule.GetWellConstrainedStructure

The function

$$\text{get_well_constrained_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}'} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the [well-constrained structure](#) of a type t in the static environment $\text{tenv} \multimap \text{t}'$, which is defined as follows. Otherwise, the result is a type error.

26.19.1 Prose

All of the following apply:

- the [structure](#) of t in tenv is $\text{t1} \text{\#TE}$;
- the well-constrained version of t1 is t' .

26.19.2 Formally

$$\frac{\begin{array}{c} \text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t1} \text{ \#TE} \\ \text{to_well_constrained}(\text{t1}) \xrightarrow{\text{type}} \text{t}' \end{array}}{\text{get_well_constrained_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}'}$$

26.20 TypingRule.GetBitvectorWidth

The function

$$\text{get_bitvector_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{expr}}^{\text{e}} \cup \text{TTypeError}$$

returns the expression e , which represents the width of the bitvector type t , or a type error if t is not a bitvector type or another type error is detected.

26.20.1 Prose

One of the following applies:

- All of the following apply (OKAY):
 - * obtaining the [structure](#) of t in tenv yields a bitvector type with width expression e , that is, $\text{T_Bits}(\text{e}, _)\text{\#TE}$;
 - * the result is e .
- All of the following apply (ERROR):
 - * obtaining the [structure](#) of t in tenv yields a type that is not a bitvector type;
 - * the result is a type error indicating that a bitvector type was expected.

26.20.2 Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \text{T_Bits}(\mathbf{e}, _) \text{ // } \#TE}{\text{get_bitvector_width}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{e}} \\
 \\
 \text{ERROR} \\
 \frac{\text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}' \quad \text{ast_label}(\mathbf{t}') \neq \text{T_Bits}}{\text{get_bitvector_width}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_EBT})}
 \end{array}$$

26.21 TypingRule.CheckBitsEqualWidth

The function

$$\text{check_bits_equal_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\mathbf{t1}}, \overbrace{\text{ty}}^{\mathbf{t2}}) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

tests whether the types $\mathbf{t1}$ and $\mathbf{t2}$ are bitvector types of the same width. If the answer is positive, the result is **TRUE**. Otherwise, the result is a type error.

26.21.1 Prose

All of the following apply:

- obtaining the width of $\mathbf{t1}$ in tenv (via *get_bitvector_width*) yields the expression $\mathbf{n} \text{ // } \#TE$;
- obtaining the width of $\mathbf{t2}$ in tenv (via *get_bitvector_width*) yields the expression $\mathbf{m} \text{ // } \#TE$;
- One of the following applies:
 - * All of the following apply (**TRUE**):
 - symbolically checking whether the bitwidth expressions \mathbf{n} and \mathbf{m} are equal (via *bitwidth_equal*) yields **TRUE**;
 - the result is **TRUE**.
 - * All of the following apply (**ERROR**):
 - symbolically checking whether the bitwidth expressions \mathbf{n} and \mathbf{m} are equal (via *bitwidth_equal*) yields **FALSE**;
 - the result is a type error indicating that the bitwidths are different.

26.21.2 Formally

$$\begin{array}{c}
\text{TRUE} \\
\frac{
\begin{array}{l}
\text{get_bitvector_width}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{n} \text{ // } \#TE \\
\text{get_bitvector_width}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{m} \text{ // } \#TE \\
\text{bitwidth_equal}(\text{tenv}, \text{n}, \text{m}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}
}{
\text{check_bits_equal_width}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TRUE}
} \\
\\
\text{ERROR} \\
\frac{
\begin{array}{l}
\text{get_bitvector_width}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{n} \text{ // } \#TE \\
\text{get_bitvector_width}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{m} \text{ // } \#TE \\
\text{bitwidth_equal}(\text{tenv}, \text{n}, \text{m}) \xrightarrow{\text{type}} \text{FALSE}
\end{array}
}{
\text{check_bits_equal_width}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TypeError}(\text{DifferentBitwidths})
}
\end{array}$$

26.22 AssocOpt

The function

$$\text{assoc_opt}(\overbrace{(\text{identifier} \times T)^*}^{\text{li}}, \overbrace{\text{identifier}}^{\text{id}}) \xrightarrow{\text{type}} \langle \overbrace{T}^{\text{v}} \rangle$$

returns the value v associated with the identifier id in the list of pairs li or `None`, if no such association exists.

26.22.1 Prose

One of the following applies:

- All of the following apply (MEMBER):
 - * a pair (id, v) exists in the list li ;
 - * the result is $\langle v \rangle$.
- All of the following apply (NOT_MEMBER):
 - * every pair $(x, _)$ in the list li has $x \neq id$;
 - * the result is `None`.

26.22.2 Formally

$$\begin{array}{c}
\text{NOT_MEMBER} \\
\frac{(\text{x}, \text{v}) \in \text{li} : \text{x} \neq \text{id}}{\text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \text{None}} \\
\\
\text{MEMBER} \\
\frac{(\text{id}, \text{v}) \in \text{li}}{\text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \langle \text{v} \rangle}
\end{array}$$

26.23 LookupConstant

The function

$$\text{lookup_constant}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{s}}) \longrightarrow \overbrace{\text{literal}}^{\text{v}} \cup \{\perp\}$$

looks up the environment **tenv** for a constant **v** associated with an identifier **s**. The result is \perp if **s** is not associated with any constant.

26.23.1 Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * **s** is associated with a constant **v** in the local environment of **tenv**;
- All of the following apply (GLOBAL):
 - * **s** is not associated with a constant in the local environment of **tenv**;
 - * **s** is associated with a constant **v** in the global environment of **tenv**;
- All of the following apply (NONE):
 - * **s** is not associated with a constant in the local environment of **tenv**;
 - * **s** is not associated with a constant in the global environment of **tenv**;
 - * the result is \perp .

26.23.2 Formally

$$\frac{\text{LOCAL} \quad L^{\text{tenv}}.\text{constant_values}(\text{s}) = \text{v}}{\text{lookup_constant}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{v}}$$

$$\frac{\text{GLOBAL} \quad L^{\text{tenv}}.\text{constant_values}(\text{s}) = \perp \quad G^{\text{tenv}}.\text{constant_values}(\text{s}) = \text{v}}{\text{lookup_constant}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{v}}$$

$$\frac{\text{NONE} \quad L^{\text{tenv}}.\text{constant_values}(\text{s}) = \perp \quad G^{\text{tenv}}.\text{constant_values}(\text{s}) = \perp}{\text{lookup_constant}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \perp}$$

26.24 TypeOf

The function

$$\text{type_of}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{s}}) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

looks up the environment **tenv** for a type **ty** associated with an identifier **s**. The result is type error if **s** is not associated with any type.

26.24.1 Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * s is associated with a type ty in the local environment of $tenv$;
- All of the following apply (GLOBAL):
 - * s is not associated with a type in the local environment of $tenv$;
 - * s is associated with a type ty in the global environment of $tenv$;
- All of the following apply (ERROR):
 - * s is not associated with a type in the local environment of $tenv$;
 - * s is not associated with a type in the global environment of $tenv$;
 - * the result is a type error indicating that s was expected to be associated with a type.

26.24.2 Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 \hline
 L^{tenv}.\text{local_storage_types}(s) = ty \\
 \hline
 type_of(tenv, s) \xrightarrow{\text{type}} ty \\
 \\
 \text{GLOBAL} \\
 \hline
 L^{tenv}.\text{local_storage_types}(s) = \perp \quad G^{tenv}.\text{global_storage_types}(s) = ty \\
 \hline
 type_of(tenv, s) \xrightarrow{\text{type}} ty \\
 \\
 \text{NONE} \\
 \hline
 L^{tenv}.\text{local_storage_types}(s) = \perp \quad G^{tenv}.\text{global_storage_types}(s) = \perp \\
 \hline
 type_of(tenv, s) \xrightarrow{\text{type}} \text{TypeError}(TE_UI)
 \end{array}$$

26.25 TypingRule.IsUndefined

The function

$$is_undefined(\overbrace{\text{SE}}^{tenv}, \overbrace{\text{identifier}}^x) \longrightarrow \overbrace{\text{B}}^b$$

checks whether the identifier x is defined as a storage element in the static environment $tenv$.

26.25.1 Prose

b is **TRUE** if and only if x is not bound in to a global storage in $tenv$ and x is not bound to a local storage in $tenv$.

26.25.2 Formally

$$\frac{b := G^{\text{tenv}}.\text{global_storage_types}(x) = \perp \wedge L^{\text{tenv}}.\text{local_storage_types}(x) = \perp}{\text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b}$$

26.26 Sorting Lists

The parametric function

$$\text{sort}(\overbrace{T^*}^{11}, \overbrace{(T \times T) \rightarrow \{-1, 0, 1\}}^{\text{compare}}) \xrightarrow{\text{type}} \overbrace{T^*}^{12}$$

sorts a list of elements of type T — 11 — using the comparison function `compare`, resulting in the sorted list 12. `compare(a, b)` returns 1 to mean that a should be ordered before b , 0 to mean that a and b can be ordered in any way, and -1 to mean that b should be ordered before a .

26.26.1 Prose

One of the following applies:

- All of the following apply (EMPTY_OR_SINGLE):
 - * 11 is either empty or contains a single element;
 - * 12 is 11.
- All of the following apply (TWO_OR_MORE):
 - * 11 contains at least two elements;
 - * f is a permutation of $1..n$;
 - * 12 is the application of the permutation f to 11;
 - * applying `compare` to every pair of consecutive elements in 12 yields either 0 or 1.

26.26.2 Formally

$$\frac{\text{EMPTY_OR_SINGLE} \quad \frac{|11| = n \quad n < 2}{\text{sort}(11, \text{compare}) \xrightarrow{\text{type}} \overbrace{11}^{12}}}{\text{TWO_OR_MORE} \quad \frac{\begin{array}{l} |11| = n \quad f : 1..n \rightarrow 1..n \text{ is a bijection} \\ 12 := [i = 1..n : 11[f(i)]] \quad i = 1..n - 1 : \text{compare}(12[i], 12[i+1]) \geq 0 \end{array}}{\text{sort}(11, \text{compare}) \xrightarrow{\text{type}} 12}}$$

Chapter 27

Type Error Codes

TE_EBT This error indicates that a bitvector type was expected where a non-bitvector type was given. See `TypingRule.CheckBinop.PLUS_MINUS_BITS_BITS` (Section 8.9) for an example.

TE_EET This error indicates that an enumeration type was expected where a non-enumeration type was given. See Section 9.8 for an example.

TE_EST This error indicates that a [structured type](#) was expected where a non-[structured type](#) was given. See Section 11.15 for an example.

TE_ETT This error indicates that a tuple type was expected where a non-tuple type was given. See Section 12.3 for an example.

TE_SWG: ASL requires each setter for a given identifier to have a corresponding getter for the same identifier. The specification either does not contain a getter for the same identifier or a getter for the same identifier exists, but it does not have the expected signature (see Section 21.20).

TE_UI An identifier that is missing a definition of the appropriate kind. See `TypingRule.SubprogramForName` (Section 19.11) for an example.

TE_LMM This error indicates that two lists that are expected to have the same length have different lengths. See Section 12.3 for an example.

TE_SDM At least two subprograms in the specification clash. See Section 21.19 for an example.

TE_NCC A function call, given by its name and list of formal argument types, does not match any defined subprogram. See Section 19.11 for an example.

TE_TMC A function call, given by its name and list of formal argument types, matches more than one subprogram, which does not allow the type-checker to decide which subprogram the call refers to. See Section 19.11 for an example.

- TE_PWD** A subprogram includes a parameter that is not associated with any variable appearing in one of the arguments. See Section 21.9 for an example.
- TE_LCA** A conditional expressions results in two types that have no common ancestor type that can represent both. See Section 8.7 for an example.
- TE_MRV** A call to a function must result in a returned value, whereas a call to a procedure must not. This error occurs when a call to a function or a getter is inferred to refer to a procedure or a setter, or a call to a procedure or a setter is inferred to refer to a function or a getter. See Section 19.2 for an example.
- TE_CBA** A call to a subprogram must have the same number of arguments as the list of formal arguments declared for the subprogram. This error indicates that the number of arguments is different to the number of declared formal arguments. See Section 19.2 for an example.
- TE_BRA** Only subprogram declarations may be mutually recursive. This error indicates that at least one declaration in a given list of mutually recursive declarations is not a subprogram. See Section 22.6 for an example.
- TE_LBI** The expressions defining the bounds of a `for` loop are required to have the [structure](#) of an integer type. This error indicates that at least one of the start expression and end expression violate this requirement. See Section 22.6 for an example.
- TE_MFI** This error indicates that an initialization of a [structured type](#) is missing an expression to initialize one of its fields. See Section 11.15 for an example.
- TE_RSB** This error indicates that two bitvector types are required to have the same bitwidths but the type-checker was not able to prove it. See Section 23.3.3 for an example.
- TE_TAF** This error indicates that a given at type assertion expression will always fail. See Section 11.33 for an example.
- TE_OFC** This error indicates that a binary expression appearing in a constraint will always fail dynamically. This means that the set of values that the type containing the constraint can take is empty. See Section 8.12 for an example.
- TE_OTB** This error indicates that the operator of a binary expression cannot be applied to its operand expressions due to their types. See Section 8.9 for an example.
- TE_IAF** This error indicates that an anonymous type is being used as a type annotation in a context where anonymous types are not allowed. See Section 9.11 for an example.
- TE_AIM** This error indicates that an assignment has a left-hand-side storage element that is immutable. See Section 12.2 for an example.
- TE_MF** This error indicates that an access is made (for either reading or writing) to a field that is not declared by the respective [structured type](#) or a bitfield that is not declared by the respective bitvector type. See Section 12.6 for an example.

TE_DII This error indicates that a [statically evaluable](#) expression contain an integer division expression where the denominator does not divide the numerator. See `TYPINGRULE.BINOPLITERALS.DIV_INT` (Section [23.3](#)) for an example.

Bibliography

- [1] Arm Architecture Technology Group. *ASL Abstract Syntax Reference*. 2024.
- [2] Arm Architecture Technology Group. *ASL Semantics Reference*. 2024.
- [3] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.