HLL

Lars Helander

This is one of two documents that are published together.

- The "HLL Language Definition" document describes the syntax and semantics of the formal modelling language HLL.
- The "HLL Logical Foundation Document" shows how the new version of the language has evolved from the previously published version 2.7.

RATP released HLL 2.7 Logical Foundations Document (LFD) in 2018. The intention was to build a community with a diverse and rich environment around HLL. Since then, tool providers and users have come together to discuss the evolution of the language, and this was the beginning of the HLL Forum.

The HLL version pr4.0rc1 presented in this document is the result of a collaborative effort among the members of the HLL Forum. It was based on the work of RATP, Prover and Systerel, who have agreed to strive for merging their various HLL versions into one common version for the benefit of the HLL community.

Should you find something in this document that you want to change for a future version, please submit your opinion to HLL Forum or to Prover, Systerel or RATP, who maintain this document together.

HLL

Language Definition

Id: HLL-LDD	Date: July 12, 2021
Version: pr4.0rc1	Pages: 123

Legal Notice

History

The language HLL was developed by Prover Technology (Prover) from 2008 to 2012 in collaboration with RATP. The language emerged as a successor of the TeclaTool language, which itself was a successor of the Tecla language, both developed by Prover before 2008.

In 2018, HLL 2.7 Logical Foundations Document (LFD) was published on the Internet [1].

Prover renamed and profoundly rewrote the LFD in order to organize it in a more modular way (all aspects of each language construct being grouped together in a single module) with the main purpose of making it easier to ensure the completeness of the language definition. The result was release candidates for version 3.0 of the document with the new title “HLL Language Definition”.

In 2018 the HLL Forum initiative started as a working group of tool providers and users of HLL. New features of HLL (compared to version 2.7) were developed jointly by the HLL Forum.

In 2021 it was decided that we use 4.0 for the upcoming version of HLL, in order not to confuse it with previous tentative versions 3.x.

This document is a prerelease (proposal) by Prover to HLL Forum of an upcoming version 4.0. It has not been approved by other members of the HLL Forum, rather it is published to enable review and feedback. When review remarks have been received and this specification has been improved according to them, the result will be published as a version 4.0, which will be jointly developed further by the HLL Forum.

Author

Lars Helander, Prover Technology SAS.

Authors of previous versions

The present document has been based on previous language specifications.

- The document "Tecla LFD", 2008, Prover Technology AB, defined the semantics of streams and was written by Gunnar Smith and Ilya Beylin.
- The document "HLL LFD", versions 1.0 to 2.7, 2012, Prover Technology SAS, defined previous versions of HLL and was written by Nicolas Breton and Jean-Louis Colaço.

Revision History

In the following table, revisions marked with a star have been **approved**.

Version	Date	Reason for change
pr4.0rc1	*Prerelease for review*	Added preamble in order to remove Prover as an owner and prepare for handover to HLL Forum.
3.0rc10	*July 5, 2021*	Removed (QuantMinMaxDomainNonEmpty) and added **nil**-behaviour instead for these quantifiers in case the domain is empty. Minor fix regarding the semantics of the domain of `SELECT`: it is static due to the new restriction (SelectQuantNoItemsDomain). Specified that line-ending comments and pragmas (`//`, `@`) are ended by the `'\n'`-character. Added (CollectionStaticFlag) which was missing. Changed the syntax of <quantif_expr> by refactoring out the `SELECT` case and accordingly removed the restriction (QuantRestrictedDefault) which is no longer needed, and renamed (QuantDefault) as (QuantSelectDefault). Added Appendix C with an overview of restrictions. Plus some other minor fixes and clarifications.
3.0rc9	*February 15, 2021*	Added authors on page 2. Marked version 3.0rc8 as approved (by the technical reviewers). Added Appendix **??** with contact details. Changed the document id from T-810712-LFD-HLL to HLL-LDD.
3.0rc8*	*January 20, 2021*	Forbid the combination of `SELECT` and `$items` by introduction of the restriction (SelectQuantNoItemsDomain). Modified (DefDeclaredLhsAssignableRhs) and renamed it (DefRhsTypeAssignableToLhsType). Minor addition to (DefCollectionRhs) regarding the type of collections. Also addressed issues arising from technical review 10609.

Version	Date	Reason for change
3.0rc7	*November 18, 2020*	Removed the appendix containing the type system. Updated the semantics of branch variables of case expressions (they are now static streams). This fixes issue 10587. Make undeclared and defined variables default to type `bool` if they are defined using a recursion. Updated the typing of untyped `PRE` expressions. Simplified the causality restriction for definitions. Updated the type of the `SELECT` to be the union type of the types of the selected value and the default value (issue 10215). Plus some minor clarifications.
3.0rc6	*February 28, 2020*	Updated (QuantVarStaticFlag) due to the introduction of `$items`.
3.0rc5	*December 3, 2019*	Provided more details for the causality restriction (DefCausality). Clarified the typing of untyped pre expressions. Adding a definition of the many-sorted model. Also added the new features agreed upon by the HLL forum. That is: 1. Unfolding definitions. 2. `SELECT` operator (as a quantifier expression). 3. A `<domain>` is allowed to be just `"bool"` or `"int"`. (The other extensions on this point having already been added in a previous version.) 4. Quantification (including `SELECT`) over composite streams. 5. Arrays and functions as proof obligations. 6. Extended integer literals (binary and hexadecimal, plus underscores for readability). 7. Always definitions aligned with latch definitions for integer streams (added a check that the value fits in the target type). 8. A new Section 2.2: Logic of Exceptions.

Version	Date	Reason for change
3.0rc4	*January 23, 2019*	Addressed issues arising from technical review 9089. Redefined `$or` so that it does not absorb **nil**.
3.0rc3	*September 24, 2018*	Addressed issues arising from technical review 8986.
3.0rc2	*September 7, 2018*	Addressed issues arising from technical review 8868.
3.0rc1	*July 17, 2018*	Included a type system in appendix.

Version	Date	Reason for change
3.0a	*March 8, 2018*	Rewritten from scratch with another document structure. Notable changes and additions to HLL include: 1. Added an order on the values of the bool, enum and integer types. 2. Functions over ordered domains are now generalized arrays, and can be used whenever arrays can. 3. Function types are compatible and assignable only if their parameter types are equal (same sets of values). (Issue 1251.) 4. Empty arrays are allowed. 5. Empty integer range types are allowed. 6. Equality and non-equality operators extended to allow any pair of objects with finite number of scalar components. 7. Membership/elementhood extended to allow any scalar type as domain. 8. Added **nil** as an exceptional value and defined its sources (divisions by zero, array indexing out of bounds, overflows, et.c.) and its propagation (can be absorbed by if-then-else, case, and some Boolean operators and quantifiers). 9. Streams of empty type (including inputs and memories) are now in principle allowed, but such a stream will carry **nil** in each time step. 10. Removed the "sized" restriction on pre expressions.
2.8	*February 21, 2014*	issues: 3987. Fix definition of dependency relation for `pre`. Move misplaced sentence in same subsection. Update the rule references in the text following the enumerated list in the definition of the dependency relation.

Version	Date	Reason for change
2.7*	*March 13, 2012*	issues: 262. Add missing case for collections in functions defined on types.
2.6	*February 24, 2012*	issues: 1243, 1244. Modification of the typing rule (case) to forbid multiple occurrence of the same variable in a pattern. Fix rule (c-definition), was two restrictive the assignability condition was missing.
2.5	*February 1, 2012*	Add a missing check on rules for quantifiers (no simultaneous multi-introduction of an identifier in the scope). Revisit the other rules that already defined this check (lhs-iterators, lhs-parameters, lambda par function and lambda par array). Reorder items in the definition of H in rule system. Add a missing condition on memories and inputs about empty sorts.
2.4	*December 19, 2011*	New syntax for the `with` (issue: 1063), Integration of RATP remarks given in FA_12_LFD-HLL_AQL-Prover_03, reorganization of the syntax, introduction of `lambda` definitions, some precisions about array projection and function application. Issue 1127
2.3	*August 23, 2011*	Integration of RATP remarks given in FA_Qualif_v04_LFD-HLL_AQL-Prover_01
2.2	*May 17, 2011*	Issues: 237, 239, 241, 245, 247, 248, 249, 250, 251, 252, 254, 256, 257, 258, 259, 260, 261, 263, 264, 265, 267, 268, 271, 272, 273, 274, 275, 276, 277, 278, 280, 281
2.1	*April 28, 2011*	Issues: 157, 126, 125, 123, 122, 121, 120, 119, 117, 115, 114, 113, 112, 111, 103, 104, 105, 106, 107, 87, 90, 91, 93
2.0	*February 28, 2011*	Major extension of the language with: quantifiers, pre, namespaces, functions, sorts and new switch-case.
1.16	*November 4, 2010*	Improvement of the postfix array type notation specification.
1.15	*October 27, 2010*	Fix BNF, the terminating "s" was missing for the keyword "obligations".
1.14	*September 20, 2010*	Change static flag for cast; this operator is not consider as static anymore. Change the semantics of memories constrained by an implementation type, the implicit cast is removed.
1.13	*January 18, 2010*	Add a comment to the typing rule of definitions, as recommended by RATP in FA-03_LFD-HLL_rqs_RATP.

Version	Date	Reason for change
1.12	*December 30, 2009*	Integration of RATP remarks in FA-03_LFD-HLL_rqs_RATP. Modification of the syntax to allow uncapitalised section names, as raised by the parser review.
1.11	*November 12, 2009*	Minor spelling corrections.
1.10	*October 5, 2009*	Adding missing rules for type `int` and collections. Fix the constraints in rules (int-signed) and (int-unsigned), were shifted.
1.9	*September 14, 2009*	Introduction of tags for the requirements. Fixes of issues found by the validation activity. Revisit the typing rule (Case with Default) to make explicit that cases values are pairwise different. Fix (array-declaration) rule, sizes must be constant.
1.8	*June 8, 2009*	Introduction of tags for the requirements. Fixes of issues found by the validation activity.
1.7	*April 17, 2009*	Modification of the semantics of integer memories, it depends now on the way it is declared (a range or an implementation type). Improvement of the presentation of integer types. Integration of the feedbacks from the approbation team (iteration 3).
1.6	*April 7, 2009*	change the associativity of the power operator, now it associates to the right.
1.5	*April 7, 2009*	Integration of the feedbacks from the approbation team (iteration 2).
1.4	*April 4, 2009*	Integration of the feedbacks from the approbation team.
1.3	*March 30, 2009*	Fix typos, revisit the section about *other static checks*.
1.2	*March 27, 2009*	Fix a lot of typos found by peer review.
1.1	*March 20, 2009*	Remove operator $\leftarrow$ (left implication) Change the associativity of $i \rightarrow$, now it is right associative. Complete the HLL semantics.
1.0	*March 19, 2008*	Initial Version.

Contents

1 Introduction

This document presents the *syntactical* and *semantical* aspects of the HLL[1] modelling language.

HLL is a declarative stream-based language with a large panel of types and operators. It is suitable for modelling discrete-time sequential behaviours and expressing temporal properties of these behaviours.

1.1 Purpose

The purpose of the document is to provide a formal definition of all aspects of HLL in order to be used for the implementation of tools considering this language as a source or a target.

This document is not intended as a user's guide or introduction to HLL.

1.2 Definitions, Terms and Abbreviations

Please refer to Appendix D.

1.3 Overview

- Section 2 introduces the basic concepts, notions and notation on which the remainder of the document rests, and should be read first of all.
- Section 3 describes the lexical structure of HLL, including comments and pragmas.
- Section 4 to Section 16 describe the semantics and restrictions of the HLL language around its EBNF syntax definition.
  - Section 4 describes identifiers.
  - Section 5 describes user namespaces and path identifiers.
  - Section 6 describes lists.
  - Section 7 describes declarators.
  - Section 8 describes types.
  - Section 9 describes accessors.
  - Section 10 describes stream expressions.
  - Section 11 describes stream declarations.
  - Section 12 describes stream definitions.
  - Section 13 describes constants.
  - Section 14 describes constraints.
  - Section 15 describes proof obligations.
  - Section 16 describes sections.

[1] HLL stands for *High Level Language*

The top-level nonterminal `<HLL>` that characterizes an HLL text is defined in Section 16 (the sections above have been ordered mostly bottom-up, with the top-level last).

- Appendix A gives the complete ENBF syntax definition in a single place, for an overview of the language, together with the operator precedence and associativity rules in A.1.
- Appendix B lists the reserved words of HLL.
- Appendix C gives an overview of the different restrictions that apply to each language construct of HLL.
- Appendix D is a glossary of words, terms and abbreviations used within this document.
- Appendix E gives an index of (external) labels exported by this document.

2 Preliminaries

2.1 Streams

HLL is a language based on the notion of *Streams*. A stream s represents an infinite sequence of values, one for each time step:

$$s: \quad s_0 \quad s_1 \quad s_2 \quad s_3 \quad \ldots \quad s_n \quad s_{n+1} \ldots$$

Streams are typed, and the type of a stream can be understood as just a set of values. For example the Boolean type (written `bool` in HLL) is the set of values $\{\text{false}, \text{true}\}$. The values s_i of the stream s are thus members of the type of s. HLL supports a large number of different types, for example Boolean, integer, enum, struct and array.

In HLL, there are two fundamentally different types of operators:

1. combinatorial operators (which are mappings from values to values), and
2. temporal operators (which are mappings from streams to streams).

The combinatorial operators that can be applied to values of a type T can be lifted to streams of type T by point-wise application in each time step. For example, if a and b are integer streams:

$$\begin{array}{lllllll} a: & a_0 & a_1 & a_2 & \ldots & a_n & \ldots \\ b: & b_0 & b_1 & b_2 & \ldots & b_n & \ldots \end{array}$$

Then the expression $a + b$ represents the stream:

$$a + b: \quad a_0 + b_0 \quad a_1 + b_1 \quad a_2 + b_2 \quad \ldots \quad a_n + b_n \quad \ldots$$

Thus, assuming that a and b have the following values:

$$\begin{array}{lllllllll} a: & 2 & 3 & 5 & 7 & 11 & 13 & \ldots \\ b: & 1 & 1 & 2 & 3 & 5 & 8 & \ldots \end{array}$$

Then:

$$a + b: \quad 3 \quad 4 \quad 7 \quad 10 \quad 16 \quad 21 \quad \ldots$$

It is important to note that lifting a combinatorial operator (like the example above) to the stream level does not make it a temporal operator: in each time step the lifted combinatorial operator has only access to the values *in the current time step* of its stream operands. A temporal operator, on the other hand, is free to access the entire stream of values of its operands. The `X` operator (read "next operator") is an example of a temporal operator which is usually thought of as returning the "next value" of its stream operand. To be precise however, the `X` operator returns a stream which is shifted one step to the left relative to its operand. To continue our example:

$$\texttt{X}(a + b): \quad 4 \quad 7 \quad 10 \quad 16 \quad 21 \quad \ldots$$

In Section 10 the semantics of the HLL nonterminal `<expr>` is defined using streams (an `<expr>` *is a* stream). Therefore we will use the term stream both for the concept described above and for any HLL expression `<expr>`.

In the following sections we will introduce some concepts and aspects of streams, which will be used to define the semantics of HLL. The most central concept is the (typed) stream model M_T which is just a mapping from a pair (stream of type T, time step) to a value of the type T, or **nil** (which is an exceptional value used to model undefined behaviour such as a division by zero). For example, M_{int} applied to the stream a of type `int` above would return the following values:

$$
\begin{aligned}
M_{\text{int}}(a, 0) &= 2 \\
M_{\text{int}}(a, 1) &= 3 \\
M_{\text{int}}(a, 2) &= 5 \\
&\vdots
\end{aligned}
$$

2.1.1 Models

Definition 1 (stream model)**.** A *Stream Model* M_T of the set S of HLL stream expressions (defined in Section 10) of type T is a binary function $M_T : S \times \mathbb{N} \to T \cup \{\textbf{nil}\}$, where $\mathbb{N}$ is the set of non-negative integers denoting the time steps of the streams (with 0 representing the initial time step) and **nil** is an exceptional value defined below.

The precise rules for the computation of the stream models are defined throughout the document (mainly in Sections 10 and 12). A general rule however, is that any stream s which has not its value defined in time step k by one of these precise rules, is free to take any value of its type T in time step k in any model M_T, *i.e.* $M_T(s, k) \in T$. This general rule applies typically to input streams.

Definition 2 (many-sorted model)**.** Letting $\mathcal{T}$ denote the set of all possible HLL types (defined in Section 8), both explicit and implicit ones, we will define a *Many-Sorted Model* M as the following set: $M = \{M_T \mid T \in \mathcal{T}\}$.

2.1.2 Exceptional Value

Definition 3 (nil)**.** A stream s of type T which is not well-defined at time step k in some model M_T takes the exceptional polymorphic value **nil** in that time step, *i.e.* $M_T(s, k) = \textbf{nil}$.

2.1.3 Propositions

Definition 4 (proposition)**.** Given a Boolean stream s, the set of possible *Propositions* over s, and their meaning, are:

Proposition	*Meaning*
$\mathbb{I}\ s$	s is true in time step 0
$\Box s$	s is always true

2.1.4 Consequences

Definition 5 (consequence)**.** A proposition q is a *Consequence* of a set of propositions P iff for all many-sorted models M, the following holds:

$$(\forall p \in P : M \models_w p) \to (M \models_s q) \tag{1}$$

where $\models_D$ is a semantic relation between models and propositions defined as:

1. $M \models_D \mathbb{I}\ s$ iff $M_{\text{bool}}(s, 0) \in D$ for $M_{\text{bool}} \in M$.
2. $M \models_D \Box s$ iff $M_{\text{bool}}(s, n) \in D$ for $M_{\text{bool}} \in M$ and for all n.

The weak variant of $\models_D$, written $\models_w$, uses the definition above with $D = \{\text{true}, \textbf{nil}\}$, and the strong variant, written $\models_s$, uses the definition above with $D = \{\text{true}\}$. We note that if the streams underlying the propositions in $P \cup \{q\}$ are all well-defined in all time steps of all models, then this distinction in a weak and a strong case becomes unnecessary.

(An alternative term for "model" is "scenario" (such as a counterexample), and the formal definition above simply states that all those, and only those, scenarios (or models) which do not falsify any of the propositions in P need to satisfy the proposition q in order for the latter to be a consequence of the former. We can think of the propositions P as the set of constraints in an HLL text H, and the proposition q as a proof obligation. The problem of deciding whether q is a consequence of P is known as "model checking" [2], and it amounts to checking the formula (1) above for all models. If there is no counter-model M to (1), we say that H is a model for q (q is true in H). Admittedly, the use of the word "model" for different purposes may be confusing, but it has historical reasons, and for the purposes of this document it is enough to consider a "model" to be synonymous with "scenario".)

2.1.5 Static Flag

Definition 6 (static flag)**.** The function $\mathcal{SF} : S \to \{0, 1, 2\}$ returns the *Static Flag* of a stream. A stream s which is *Static* takes the same value in each time step, *i.e.* $M_T(s, n) = v$ for some $v \in T \cup \{\textbf{nil}\}$ and for all n. The static flag for each type of stream expression will be given in association with the semantic description of the expression (*i.e.* throughout the document). However, the informal meaning of the values of the static flag is given in the following table (in order to provide the reader with an intuition about these values).

Static Flag	*Informal Meaning*
$\mathcal{SF}(s) = 0$	s is not known to be static
$\mathcal{SF}(s) = 1$	s is static
$\mathcal{SF}(s) = 2$	s is static and a combination of only constants and literal values

The static flag is used to restrict the set of possible HLL types and stream expressions.

In Section 10.4 we extend the static flag to domains (nonterminal `<domain>`), which are streams of sets of values, and in Section 12 we extend it to collections of streams (nonterminal `<collection>`).

[2]The original formulation of the model checking problem was: Given a Kripke structure M and a temporal formula f, check whether f is true in M, *i.e.* whether M is a model for f.

2.2 Logic of Exceptions

In HLL, many operations such as division by zero, overflow, array indexing out of bounds et.c., give rise to the exceptional value **nil** introduced in Section 2.1.2.

This exceptional value, **nil**, propagates unhindered through most HLL operations, but are absorbed by a few, such as the if-then-else and the Boolean operations with absorbing values (and, or, implication). In order to preserve the commutativity of the Boolean operators, they are defined to be symmetric in the sense that **nil** can be absorbed on either side of the operator[3]. (This means that "true or **nil**" and "**nil** or true" both mean "true".)

We note, however, that any implementation of HLL which lets **nil** propagate more freely, and for example reduces "**nil** or true" to **nil** instead of "true", is still *safe* due to the fact that **nil** is accepted by $\models_w$ on the left hand side of the consequence relation defined by formula (1) in Section 2.1.4 above, whereas it is rejected by $\models_s$ on the right hand side. This asymmetry in the consequence relation ensures that an implementation which propagates **nil** more freely will accept more models in the antecedent and less models in the consequent making it strictly harder to satisfy formula (1).

[3]This semantics of three-valued logic corresponds to "Kleene's strong logic of indeterminacy" (Kleene's K_3^S).

2.3 Namespaces and Scoping

In HLL, names (identifiers) reside in different namespaces depending on which kind of entity they name. (This means that entities of different kind may have the same name within the same scope, as explained below.) The different kinds of entities and their corresponding namespaces are:

1. streams,
2. types,
3. user namespaces, and
4. struct components.

For each of the first three kinds of entities in the list above, there is a single namespace, whereas for struct components, there is one namespace per struct type.

A namespace can (at some fixed point in an HLL text) be divided into scopes, which are stacked downwards one upon the other. A namespace, in this document, is thus not a single set of names, but rather a collection of sets of names, one for each scope.

Assume such a scope stack, for example $[\mathtt{S}_1\ \mathtt{S}_2\ \ldots\ \mathtt{S}_n]$ where $\mathtt{S}_1$ is the top-level scope and $\mathtt{S}_n$ is the bottom-most scope (the "current" scope), and some element $\mathtt{S}_i$ with $\mathtt{i} \in [1, n]$. We will call a scope $\mathtt{S}_j$ with $\mathtt{j} < \mathtt{i}$ an *Ascendant* scope of $\mathtt{S}_i$, and we will call a scope $\mathtt{S}_k$ with $\mathtt{k} > \mathtt{i}$ a *Descendant* scope of $\mathtt{S}_i$. Note that the scopes $\mathtt{S}_1 \ldots \mathtt{S}_{i-1}$ constitute all the ascendant scopes of $\mathtt{S}_i$ whereas the scopes $\mathtt{S}_{i+1} \ldots \mathtt{S}_n$ only constitute a subset of the descendant scopes of $\mathtt{S}_i$ since some of these scopes may have other descendant scopes at other points in the HLL text.

The entities that are *Visible* (*i.e.* that can be referenced) in a given scope are those that exist in that scope or in one above it (an ascendant scope).

Names must be unique[4] within a given scope of a namespace, *i.e.* two different entities existing in the same scope of a namespace cannot have the same name. However, *Hiding* is allowed (but not encouraged), meaning that an entity $\mathtt{E}_1$ may have the same name as another visible entity $\mathtt{E}_2$ that exists in an ascendant scope (one higher up in the stack). In such a case we say that $\mathtt{E}_1$ *Hides* $\mathtt{E}_2$ in all scopes in which $\mathtt{E}_1$ is visible.

It is important not to confuse "namespace" with "user namespace"; the former is the abstract concept described above, the latter is a concrete concept of HLL which is defined in Section 5.

Scopes are introduced by language constructs such as user namespaces, lambda expressions (Section 10.2), case expressions (Section 10.7.9) and quantifier expressions (Section 10.7.10). The formal descriptions of those constructs define the start and end points of the scopes they introduce, and the namespaces the scopes operate on.

Scopes introduced by user namespaces differ slightly from the scopes introduced by the other aforementioned language constructs, in that the entities inside the former can be referenced from the outside by using a *path identifier* (Section 5.1), whereas the entities inside the latter can be referenced only within the same scope or a descendant one. These referencing rules are formally defined in Section 5.1.

[4] This general requirement is instantiated as a number of restrictions throughout the document, typically on the form: *"Some named {stream, type, ... } E may not be defined more than once per scope of the namespace of {streams, types, ... }."*

The namespaces containing struct components are not subdivided into scopes, since each struct type introduces its own namespace for the components, as formalised in Section 8.2.2.

2.4 Notation

See Appendix D for a comprehensive list of words, terms and abbreviations used within this document.

2.4.1 Syntax-Related Notation

The syntax is given using the following grammar notation (similar to EBNF).

- a nonterminal symbol is written `<symbol>`;
- a nonterminal definition (a production rule) is introduced by `::=` with the defined nonterminal as a left-hand side;
- a terminal symbol is given by a string separated with quotes (`"terminal_string"`);
- the pipe `|` separates alternative items (`<item1> | <item2>`);
- square brackets represent the optional items (`[<may-be-used>]`);
- braces represent 0 or more times repetitions (`{<item>}`);
- braces extended with `+` represent 1 or more times repetitions (`{<item>}+`);
- parentheses are used for explicit grouping in grammar expressions.

For the terminals that are described with a regular expression, the right-hand side of the rule starts with `regexp:`.

2.4.2 Semantics-Related Notation

We will use (mostly uppercase) letters in `This Font` to denote variables representing nonterminals (*i.e.* syntactic variables). For example, `V`, `T` and `E` are typically used to represent respectively the nonterminals `<id>` (Section 4), `<type>` (Section 8) and `<expr>` (Section 10). However, often we simply use the nonterminal itself for the same purpose.

In a slight abuse of notation (but in an effort to ensure visual coherency), we extend the use of these symbols to also denote semantic entities in general (even if they do not have a concrete HLL syntax), for example the letter `T` is used throughout the document to denote any type, even one which does not have an explicit HLL syntax (see Section 8.4 for examples).

In order to have a compact definition of HLL, the semantics of the various language constructs is often defined by translation to other, more basic or general, language constructs.

Typically, for a language contruct `C1` and a more basic language contruct `C2`, we will use the following two relations to relate `C1` to `C2`:

Relation	*Type*	*Meaning*
`C1` *is equivalent to* `C2`	equivalence relation (reflexive, symmetric transitive)	Either of `C1` and `C2` can be used in place of the other, without change of meaning or correctness, anywhere in an HLL text in which all subexpressions have been explicitly grouped[5].
`C1` *is reducible to* `C2`	preorder (reflexive, transitive)	`C2` can be used in place of `C1`, without change of meaning or correctness, anywhere in an HLL text in which all subexpressions have been explicitly grouped.

To avoid redundancy, whenever we say that `C1` is equivalent or reducible to `C2` this means that all relevant restrictions that apply to `C2` (directly or indirectly), also apply to `C1`, even if this is not explicitly said. Furthermore, if `C1` and `C2` are expressions they will have the same type even if this is not explicitly said.

As an example, the expression `E1 != E2` is defined as equivalent to `~(E1 = E2)`. This means that the relevant restriction that says that the operands of `=` be of "compatible types with a finite number of scalar components" also applies to operator `!=`. By contrast, the restriction that says that the operand of `~` be of type `bool` is irrelevant, and does not apply to operator `!=`.

To be more precise about which restrictions are relevant, one should consider a "language construct" as being a function from one or more explicitly grouped HLL strings to a single explicitly grouped HLL string. For example, if we let `=` denote the function `EQ(<expr>,<expr>)` ⇒ `(<expr> = <expr>)`, and `~` the function `NOT(<expr>)` ⇒ `~<expr>`, then we can define `!=` as being equivalent to the function composition `NOT` ∘ `EQ` (*i.e.* first apply `EQ` then `NOT`). The restrictions that apply to the domain of definition of `!=` are thus those that apply to the domain of definition of `NOT` ∘ `EQ` (which is the same as the domain of `EQ`).

If a language construct `C1` has additional restrictions compared to another construct `C2`, but they are otherwise equivalent, we will typically say that `C1` is reducible to `C2`. As an example, the expression `E1 <-> E2` is reducible to `E1 = E2` since the type `bool` required of the operands of `<->` satisfies the restrictions on `=`, but not the other way around.

Occasionally, we will use the relation "`C1` is equivalent to `C2` except for some property `P`". This means, naturally, that the semantics of `C1` and `C2` is identical except with regards to the property `P` (which is of minor importance). This relation will be used whenever neither of `C1` and `C2` is reducible to the other, but the semantics of the language constructs are still closely related.

[5] Expressions such as "`a & b & c`" and "`a + b * c`" have to be rewritten into respectively "`((a & b) & c)`" and "`(a + (b * c))`" in order for these substitutions to work in the general case.

2.5 Document Structure

This document has been structured around the syntax of HLL[6]. The syntax, semantics and restrictions for each set of closely related nonterminals of the language are grouped together whenever possible and the resulting groups are sorted with the goal of minimizing forward references. Each such group is placed in its own section that may contain a short introduction with a few examples followed by a formal definition. Please note that the introduction and examples are only intended as a help to understand the formal definition.

In the next section we give an example of how such a section may look like.

2.5.1 Language Construct Example

Here we may give a short informal introduction with a few examples as a help for the reader. Below, in the blue box, follows the formal definition. The labels (ExampleSyntax), (ExampleSemantics) and (ExampleRestriction) can be used for referencing purposes either within this document or another one. An index of labels can be found in Appendix E.

Syntax

(ExampleSyntax)

```
<nonterminal1> ::= <nonterminal2>
                 | <nonterminal3>
<nonterminal2> ::= "terminal1" <nonterminal4> "terminal2"
```

Forward References

1. Here we give references to the definitions of the nonterminals appearing on the right hand side of a nonterminal definition and defined in a subsequent section of the document. Note that nonterminals which are defined in a subsequent *sub*section of the present section are *not* listed here.

Semantics

1. (ExampleSemantics) Here we define the semantics (meaning) of all nonterminals appearing on the left hand side of a nonterminal definition above. In this case it means `<nonterminal1>` and `<nonterminal2>`.

Restrictions

1. (ExampleRestriction) Here we list all the cases of HLL strings which are valid syntactically, but still not part of the language.

Related Notation

1. Here we will sometimes introduce notation related to the current language construct, that is to be used elsewhere in the document.

[6] For an overview of the complete syntax, please refer to Appendix A.

3 Lexical Structure

Characters in an HLL text shall be encoded in ASCII, or any 8-bit extension of it.

The following characters may be inserted freely anywhere between terminals in an HLL text:

Character	*ASCII value*
`'\t'` (horizontal tab)	9
`'\n'` (new line)	10
`'\r'` (carriage return)	13
`' '` (white space)	32

Comments and pragmas, defined below, may also be inserted freely between terminals.

3.1 Comments

Informal Description

An HLL text can contain comments of the following forms:

1. (CommentDoubleSlash) lines containing a `"//"` (double slash) are ignored starting from the `"//"` sequence up to, and including, the end of the line character `"\n"` (including `"/*"` and `"*/"`);
2. (CommentSlashStar) characters present between `"/*"` and `"*/"` are ignored (including `"//"`); comments of this kind can be nested.

The tokens `"//"`, `"/*"` and `"*/"` are considered in the order they appear in the file.

Here are some examples that illustrate this specification:

```
int a; // this "/*" is not seen as a comment start
 /* the one at the beginning of this line is
  // The previous "//" on this line does not start a comment. */

int a; /* the present text is inside a comment
          /* this one too */
         this one also */
```

3.2 Pragmas

Informal Description

(Pragma) All the characters after an `"@"` are interpreted as the text of a pragma up to, and including, the end of the line character `"\n"`.

Pragmas may be used by tools taking HLL as input language. The semantics of such pragmas is outside the scope of this document. From the point of view of this document, `"@"` is equivalent to `"//"`.

4 Identifiers

Syntax

(IdSyntax)

```
<id>      ::= regexp: [a-zA-Z_][a-zA-Z0-9_]*
            | regexp: '[^\n']+'
            | regexp: "[^\n"]+"
```

Semantics

1. (Id) An identifier (`<id>`) identifies a named entity of an HLL text by its name, where an entity is either either a type, a stream, a struct component or a user namespace. An identifier may be used either to *give* an entity its name (by a declaration or definition), or to refer to an existing named entity. In a few cases a reference to a nonexisting entity may cause the entity to exist. This is called *implicit declaration by reference*.

2. (IdSignificantChars) Identifiers are case sensitive and all characters (including quotes) of an identifier are significant.

Restrictions

1. (ReservedWords) An `<id>` may not be a reserved word. The reserved words are listed in Appendix B.

5 User Namespaces

Syntax

(NamespaceSyntax)

```
<namespace> ::= <id> "{" <HLL> "}"
```

Forward References

1. `<HLL>` is defined in Section 16.

Semantics

1. (UserNamespace) A *User namespace* (`<namespace>`) is a named container for names of an `<HLL>` text, meaning that names contained in different user namespaces will not clash. More precisely, a user namespace `N { HLL }` introduces local scopes in the namespaces of streams, types and user namespaces that start at the `{` and ends at the `}`. These local scopes are called the top-level scopes of the user namespace, and are not to be confused with the global top-level scopes.
2. (UserNamespaceName) The `<id>` defines the name of the namespace and it resides in the namespace of user namespaces.
3. (UserNamespaceScattering) Two `<namespace>` in the same scope and with the same `<id>` refers to the same user namespace. This means that `N { HLL`$_1$ `}` ... `N { HLL`$_2$ `}` (where ... represents anything in the text that may come in between of the two `<namespace>`) is equivalent to `N { HLL`$_1$ `HLL`$_2$ `}`.

Restrictions

(empty)

5.1 Path Identifiers

Path identifiers allow referencing named types or streams in any user namespace (all top-level members of a user namespace are public and can be referenced from the outside). An example:

```
Namespaces:
 NS1 { Inputs: x;
       Outputs: NS2::x; // Refers to x inside NS2
     }
 NS2 { Inputs: x;
       Outputs: NS1::x; // Refers to x inside NS1
     }
Outputs:
 NS1::x; // Refers to x inside NS1
 NS2::x; // Refers to x inside NS2
```

Syntax

(PathIdSyntax)

```
<path_id>       ::= <relative_path> <id>
                  | <absolute_path> <id>
<relative_path> ::= { <id> "::" }
<absolute_path> ::= "::" { <id> "::" }
```

Semantics

1. (PathId) A `<path_id>` refers to an entity with name `<id>` in some user namespace (or on global top-level) designated by an optional prefix which is either a *Relative path* (`<relative_path>`) or an *Absolute path* (`<absolute_path>`). An empty prefix designates either the user namespace in which the `<path_id>` occurs or else the global top-level (if the `<path_id>` occurs there).
2. (PathRelative) A `<relative_path>` $\texttt{NS}_1 :: \texttt{NS}_2 :: \ldots :: \texttt{NS}_n ::$ designates a user namespace $\texttt{NS}_n$ that is nested inside user namespaces $\texttt{NS}_1 \ldots \texttt{NS}_{n-1}$ where $\texttt{NS}_1$ is selected by applying the following rules in order:
   (a) If the `<relative_path>` occurs in a user namespace `N` and `N` has a directly nested user namespace $\texttt{NS}_1$, then this $\texttt{NS}_1$ is selected.
   (b) Otherwise, the user namespace $\texttt{NS}_1$ defined on the global top-level is selected[7].
3. (PathAbsolute) An `<absolute_path>` $:: \texttt{NS}_1 :: \texttt{NS}_2 :: \ldots :: \texttt{NS}_n ::$ designates a user namespace $\texttt{NS}_n$ that is nested inside user namespaces $\texttt{NS}_1 \ldots \texttt{NS}_{n-1}$ where $\texttt{NS}_1$ is defined on the global top-level.

[7] In this case, if there is no $\texttt{NS}_1$ on the global top-level then the corresponding `<path_id>` is invalid.

4. (PathIdLookup) Given a <path_id> [PATH::]ID, the following rules are applied to lookup the entity E with the name ID:

   (a) If no PATH is given, then the scope stack to consider for the search is the maximal one ending where the <path_id> occurs, *i.e.* [S_1 S_2 ... S_n], where S_1 corresponds to the global top-level and S_n corresponds to the current (bottom- or inner-most) scope.

   (b) If a PATH is given, then the search for ID is made only at the top-level[8] scope S_{PATH} of the user namespace designated by PATH. The scope stack to consider for the search is thus [S_{PATH}].

   (c) Given a scope stack [S_1 S_2 ... S_k] as determined from cases 4a and 4b, the search for ID starts in S_k and then proceeds in the order $S_{k-1}, S_{k-2}, \ldots S_2, S_1$. The search stops as soon as an entity E with the name ID is encountered in one of the S_i, and it is this entity E which the <path_id> refers to. The semantics or correctness of a <path_id> that does not refer to any existing entity E depends on the context in which the <path_id> is used. The general rule is that such a <path_id> is incorrect, and the only exception to this rule are unqualified named expressions, which implicitly declare the nonexisting stream variables they refer to, causing them to exist (see (NamedExprImplicitDecl) of Section 10.7.3).

Restrictions

1. (PathIdNoImplicitDecl) A qualified <path_id> (*i.e.* a <path_id> with at least one occurrence of ::) shall refer to an existing entity (no implicit declaration by reference).

[8] This means that the path identifiers i and N :: i in the lambda expression below do not refer to the same entity named i, since the first one refers to the lambda parameter and the second one to the input variable.

```
Namespaces : N { Inputs :  i;
                 Outputs : lambda(bool) : (i) := i & N :: i; }
```

6 Lists

Lists are a purely syntactic concept that has no particular semantics.

Syntax

```
<id_list>   ::= <id> {"," <id>}

<type_list> ::= <type> {"," <type>}

<expr_list> ::= <expr> {"," <expr>}
```

Forward References

1. `<type>` is defined in Section 8.
2. `<expr>` is defined in Section 10.

Semantics

(empty)

Restrictions

(empty)

7 Declarators

Declarators in HLL provide a way to declare objects of array and function type that reflects the *use* of the objects. For example:

```
Inputs:
 bool A[4][3]; // A is an array of 4 arrays of 3 bool
Outputs:
 A[3][2];      // The last element of A (array-indexing is 0-based)
               // (This use of A reflects the declaration of A)
```

An equivalent notation, shown below, is the array-type notation introduced in Section 8.2.4, and it is this rewriting from declarators to proper type notation that is performed by the function `calc_type` on the next page.

```
Inputs:
 bool^(3)^(4) A; // A is an array of 4 arrays of 3 bool
Outputs:
 A[3][2];        // The use of A does not reflect
                 // the declaration of A anymore
```

An analogue example can be made using function declarators and function-type notation, for example the declaration `bool f(int)` corresponds to the declaration `(int -> bool) f` using the function-type notation introduced in Section 8.2.3.

Note that for multidimensional arrays the syntax `bool A[4, 3]` can be used instead of `bool A[4][3]` above (the corresponding array-type notation is then `bool^(4, 3)`). The two ways to express multidimensionality are similar but not equivalent. The same is true of functions: in HLL we can both have a function taking several parameters (called a multivariate function) or a function returning another function.

Declarators can be used also in type definitions, as shown in the example below, which is an equivalent formulation of our example above:

```
Types:
 bool T[4][3]; // T is the type bool^(3)^(4)
Inputs:
 T A;
Outputs:
 A[3][2];
```

Syntax

(DeclaratorSyntax)

```
<declarator>        ::= <id> {<declarator_suffix>}
<declarator_suffix> ::= "[" <expr_list> "]"
                      | "(" <type_list> ")"
```

Semantics

1. (Declarator) A *Declarator* (`<declarator>`) consists of an identifier `<id>` and an optional declarator suffix (`<declarator_suffix>`).

2. (DeclaratorTypeCalc) The recursive function `calc_type` defined below takes as input a base type `T` (a `<type>`) and a list of `<declarator_suffix>` `D` and returns an augmented type (which is typically associated with the identifier `<id>` of the corresponding `<declarator>`, if any). Note that the base type `T` is not provided from a `<declarator>`. Instead it typically comes from an enclosing syntactic rule. See for example `<type_def>` in Section 8.

```
calc_type(<type> T, {<declarator_suffix>} D)  {
  let L be the last <declarator_suffix> of D and let D \ L
  denote D without L in :
     if L is [E1, E2, ... En] :
        return calc_type(T^(E1, E2, ... En), D \ L).
     else if L is (T1, T2, ... Tn) :
        return calc_type((T1 * T2 * ... * Tn -> T), D \ L).
     else (D is empty) :
        return T.
}
```

Please note that the array type notation $\texttt{T\^{}}(\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ is defined in Section 8.2.4 and the function type notation $(\texttt{T}_1 * \texttt{T}_2 * \ldots * \texttt{T}_n \texttt{ -> T})$ in Section 8.2.3.

Restrictions

1. (DeclArrayDimInteger) Each $\texttt{E}_i$ of a declarator suffix $[\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n]$ shall be of integer type (see Section 8.1.2).

2. (DeclArrayDimConstant) $\mathcal{SF}(\texttt{E}_i) = 2$ for a declarator suffix $[\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n]$.

3. (DeclFunctionParamScalar) Each $\texttt{T}_i$ of a declarator suffix $(\texttt{T}_1, \texttt{T}_2, \ldots \texttt{T}_n)$ shall be scalar (see Section 8.1).

8 Types

Types in HLL should be understood as sets of values. As an example the Boolean type (written `bool`) is the set $\{\text{false}, \text{true}\}$. Types are assigned to stream variables (named streams) either by explicit declaration (see Section 11), by inference (see (DefUndeclaredType) of Section 12), or in rare cases by reference (see (NamedExprImplicitDecl) of Section 10.7.3). Types are assigned to stream expressions (unnamed streams) by type inference based on the operator and the types of operands (if the operator is overloaded to handle more than one type of operands). The precise typing rules are given in connection with the definition of the expressions' semantics (*i.e.* throughout Section 10).

A stream can only take values of its assigned type (regardless of how the stream was assigned the type), or the exceptional value **nil** in response to an exceptional event such as a division by zero or an overflow.

Syntax

(TypeSyntax)

```
<type>      ::= <bool>
              | <integer>
              | <tuple>
              | <structure>
              | <function>
              | <array>
              | <named_type>

<type_def>  ::= <type> <declarator> {"," <declarator>}
              | <enum_def>
              | <sort_def>
```

Semantics

1. (Type) A *Type* (`<type>`) is a (possibly empty[9], possibly infinite) set of values. For example, the Boolean type `bool` is the set $\text{false}, \text{true}\}$.
2. (TypeDef) `<type_def>` is a type definition (or a definition of a named type) that associates an identifier `<id>` to a type. The identifier can be used, as part of a `<path_id>`, wherever a `<type>` is expected. If a type is defined using a `<declarator>`, then it is the first `<id>` of that `<declarator>` that is being defined.
3. (TypeIdSpace) The identifier of a type resides in the namespace of types, and is visible everywhere in its scope, regardless of the position of the `<type_def>`.
4. (TypeCalc) The resulting type associated to the identifier being defined by a type definition (a `<type_def>`) involving a `<type>` (the base type) and a `<declarator>` is calculated according to procedure `calc_type` in Section 7.

[9] We will treat **nil** as a truly exceptional value that does not belong to the types propers.

5. (InlineMultTypeDef) A single type definition `<type>` $\texttt{D}_1, \texttt{D}_2, \ldots \texttt{D}_n$ where each $\texttt{D}_i$ is a `<declarator>` is equivalent to `n` type definitions `<type>` $\texttt{D}_1$, `<type>` $\texttt{D}_2$, ... `<type>` $\texttt{D}_n$.

6. (TypeCompatibility) The *Compatibility* relation $\texttt{C}(\texttt{T}_1, \texttt{T}_2)$ between types is an equivalence relation (reflexive, symmetric and transitive) defined in the remainder of the document. Two types are said to be *Compatible* iff $\texttt{C}(\texttt{T}_1, \texttt{T}_2) = \text{true}$. In Section 10.4 we will extend this relation to include domains (`<domain>`).

7. (TypeAssignability) The *Assignability* relation $\texttt{A}(\texttt{T}_1, \texttt{T}_2)$ between types is a preorder (reflexive and transitive) defined in the remainder of the document. A type $\texttt{T}_1$ is said to be *Assignable to* another type $\texttt{T}_2$ iff $\texttt{A}(\texttt{T}_1, \texttt{T}_2) = \text{true}$.

Restrictions

1. (TypeDefUnicity) A named type which is defined by a `<type_def>` which is not a `<sort_def>` may only be defined once per scope of the namespace of types.

2. (TypeDefCausality) A named type may not be defined in terms of itself, either directly or indirectly. This restriction also applies to sort contributions: a sort may not contribute to its own definition.

8.1 Scalar Types

Scalar Types are the subset of HLL types that consists of scalar (or atomic) values. These are the Boolean, integer, enum and sort types.

8.1.1 Boolean Type

Syntax

(BoolSyntax)

```
<bool> ::= "bool"
```

Semantics

1. (BoolValues) The type `bool` is comprised of the two values true and false.
2. (BoolCompatibility) The `bool` type is compatible with itself.
3. (BoolAssignability) The `bool` type is assignable to itself.
4. (BoolValueOrder) false $<$ true.

Restrictions

(empty)

8.1.2 Integer Types

HLL's integer types can be either finite or infinite (the set $\mathbb{Z}$). Finite types are restricted by either an inclusive range (for example `int [4, 9]`), or an "implementation" (for example `int signed 32`). Historically, the main purpose of finite integer types has been to give bounds to inputs (free variables) and state-holding elements (latches and pre-expressions), thus ensuring that the state-space of an HLL system is finite. However, since HLL version 3.0, all stream variables declared with a finite integer type `T` will in each time step only take values from `T`, except on overflow which will result in **nil** being taken instead.

Syntax

(IntSyntax)

```
<integer>   ::= "int"
              | "int" <sign>
              | "int" <range>
<sign>      ::= "signed" <id_or_int>
              | "unsigned" <id_or_int>
<id_or_int> ::= <id>
              | <int_literal>
<range>     ::= "[" <expr> "," <expr> "]"
```

Forward References

1. `<expr>` is defined in Section 10.

Semantics

1. (IntValues) The type `int` is comprised of all integers (the set $\mathbb{Z}$).
2. (IntSignedValues) The type `int signed` $\mathtt{E}_1$ is comprised of the integers in the interval $[-2^{\mathtt{E}_1-1}, 2^{\mathtt{E}_1-1}-1]$.
3. (IntUnsignedValues) The type `int unsigned` $\mathtt{E}_2$ is comprised of the integers in the interval $[0, 2^{\mathtt{E}_2}-1]$.
4. (IdOrInt) `<id_or_int>` is a constant stream expression restricted to integer constants (`<id>`) and integer literals (`<int_literal>`).
5. (IntRangeValues) The type `int` $[\mathtt{E}_3, \mathtt{E}_4]$ is comprised of the integers in the interval $[\mathtt{E}_3, \mathtt{E}_4]$.
6. (IntCompatibility) All integer types are compatible with each other.
7. (IntAssignability) All integer types are assignable to each other.
8. (IntValueOrder) $\mathtt{i} < \mathtt{i}+1$ for any integer `i`.

Restrictions

1. (IntSizeInteger) $\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3$ and $\mathtt{E}_4$ shall be of integer type.
2. (IntSizeConstant) $\mathcal{SF}(\mathtt{E}_i) = 2$ for $\mathtt{i} \in [1, 4]$.

3. (SignedBitsPositive) $\texttt{E}_1 > \texttt{0}$.
4. (UnsignedBitsNonNegative) $\texttt{E}_2 \geq \texttt{0}$.
5. (IntSizeNotNil) $M_{\text{int}}(\texttt{E}_\texttt{i}, k) \neq \textbf{nil}$ for $\texttt{i} \in [\texttt{1}, \texttt{4}]$ and all k.

Related Notation

1. The types `int <sign>` are called *Integer implementation* types.
2. The types `int <range>` are called *Integer range* types.

8.1.3 Enum Types

Syntax

(EnumSyntax)

```
<enum_def>   ::= <enumerated> <id>
<enumerated> ::= "enum" "{" <id_list> "}"
```

Semantics

1. (EnumDef) The enum type defined as `enum` $\{\mathtt{V}_1, \mathtt{V}_2, \ldots \mathtt{V}_n\}$ is comprised of the values $\{\mathtt{V}_1, \mathtt{V}_2, \ldots \mathtt{V}_n\}$.
2. (EnumValueSpace) The values of an enum type reside in the namespace of streams.
3. (EnumValueDef) The definition of an enum type also defines its values.
4. (EnumCompatibility) An enum type is compatible with itself.
5. (EnumAssignability) An enum type is assignable to itself.
6. (EnumValueOrder) $\mathtt{V}_i < \mathtt{V}_{i+1}$.

Static Flag

1. (EnumValueStaticFlag) $\mathcal{SF}(\mathtt{V}) = 2$ for an enum value `V`.

Restrictions

1. (EnumValueUnicity) An enum value may not be defined more than once per scope of the namespace of streams.

8.1.4 Sort Types

Syntax

(SortSyntax)

```
<sort_def>     ::= "sort" [ <sort_contrib> "<" ] <id>
<sort_contrib> ::= <path_id_list>
                 | "{" <id_list> "}"
<path_id_list> ::= <path_id> {"," <path_id>}
```

Semantics

1. (SortDef) A `<sort_def>` is a contribution to the definition of a sort type. A `<sort_def>` without a `<sort_contrib>` is an empty contribution to the sort type. (The name of the sort type is given by the `<id>` as in any type definition.)

2. (SortContrib) A sort type is defined by one or more contributions. The values of the sort type is the union of the values of its contributions.

3. (SortContribScope) Contributions to a sort type `S` can only be made within the same scope.[10]

4. (SortValueSpace) The values of a sort type reside in the namespace of streams.

5. (SortSubTypeContrib) `sort` $\mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_k$ `<` `S` denotes the contribution of sort types $\mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_k$ to sort type `S`, and the inclusion of their values into `S` (*i.e.* $\mathtt{S}_1 \cup \mathtt{S}_2 \cup \ldots \cup \mathtt{S}_k \subseteq \mathtt{S}$).

6. (SortValueContrib) `sort` $\{\mathtt{V}_1, \mathtt{V}_2, \ldots \mathtt{V}_n\}$ `<` `S` denotes the definition of the values $\mathtt{V}_1, \mathtt{V}_2, \ldots \mathtt{V}_n$, and their inclusion into the sort type `S` (*i.e.* $\{\mathtt{V}_1, \mathtt{V}_2, \ldots \mathtt{V}_n\} \subseteq \mathtt{S}$).

7. (SortCompatibility) All sort types are compatible with each other.

8. (SortAssignability) A sort type $\mathtt{T}_1$ is assignable to another sort type $\mathtt{T}_2$ if either $\mathtt{T}_1$ is the same type as $\mathtt{T}_2$, or $\mathtt{T}_1$ contributes, either directly or indirectly, to the definition of $\mathtt{T}_2$.

9. (SortValueOrder) Not defined.

Static Flag

1. (SortValueStaticFlag) $\mathcal{SF}(\mathtt{V}) = 2$ for a sort value `V`.

Restrictions

1. (SortValueUnicity) A sort value may not be defined more than once per scope of the namespace of streams.

2. (SortSubTypes) $\mathtt{S}_i$ for $\mathtt{i} \in [1, \mathtt{k}]$ of (SortSubTypeContrib) shall refer to sort types.

[10] Two different sort types named `S` are defined in the following example. One is defined on the global top-level and one within the user namespace `N`:

```
Types : sort {V1} < S;
Namespaces : N { Types : sort {V2} < S; }
```

8.2 Composite Types

Composite Types are the subset of HLL types that are not scalar. A composite type is a composition of HLL types, with each type of the composition said to be a *Component* of the composite type. By extension, a stream of composite type is made up by components, which are themselves streams. The composite types are tuples, structs, arrays, and functions.

8.2.1 Tuple Types

A tuple type `tuple {bool, int [0, 2]}` consists of the $2 * 3 = 6$ values in the set `bool` $\times$ `int` $[0, 2]$, *i.e.* $\{(\text{false}, 0), (\text{false}, 1), (\text{false}, 2), (\text{true}, 0), (\text{true}, 1), (\text{true}, 2)\}$.

Syntax

(TupleSyntax)

```
<tuple> ::= "tuple" "{" <type_list> "}"
```

Semantics

1. (TupleType) A tuple is a composition of ordered unnamed components. The components are ordered according to the order they appear in the text.
2. (TupleValues) The type `tuple` $\{\texttt{T}_1, \texttt{T}_2, \ldots \texttt{T}_\texttt{n}\}$ is comprised of the values in the `n`-fold Cartesian product of its component types, *i.e.* the set $\texttt{T}_1 \times \texttt{T}_2 \times \ldots \times \texttt{T}_\texttt{n}$.
3. (TupleCompatibility) Two tuple types are compatible iff they have the same number of components and the types of those components are pairwise compatible.
4. (TupleAssignability) A tuple type $\texttt{T}_1$ is assignable to another tuple type $\texttt{T}_2$ iff they have the same number of components and each component of $\texttt{T}_1$ is assignable to its corresponding component in $\texttt{T}_2$.

Restrictions

(empty)

Related Notation

1. Given a value $\texttt{V}_\texttt{T}$ of tuple type `tuple` $\{\texttt{T}_1, \texttt{T}_2, \ldots \texttt{T}_\texttt{n}\}$ and an integer literal `K` with $0 \leq \texttt{K} < \texttt{n}$, we will write $\texttt{V}_\texttt{T}$@K to denote the component of $\texttt{V}_\texttt{T}$ of type $\texttt{T}_{\texttt{K}+1}$ at the 0-based index `K`. Note that @ is an operation on tuple values, and distinct from the HLL tuple accessor (`.K`) that operates on tuple streams.

8.2.2 Struct Types

From a user's perspective, the only difference between a struct and a tuple is the way to access the components: struct components are accessed by their names, whereas tuple components are accessed using a 0-based integer index. The two types are nevertheless distinct and cannot be mixed.

Syntax

(StructSyntax)

```
<structure>   ::= "struct" "{" <member_list> "}"

<member_list> ::= <id> ":" <type> {"," <id> ":" <type>}
```

Semantics

1. (StructType) A struct is a composition of ordered named components. The components are ordered according to the order they appear in the text.
2. (StructValues) The type `struct` $\{\mathtt{M}_1 : \mathtt{T}_1, \mathtt{M}_2 : \mathtt{T}_2, \ldots \mathtt{M}_n : \mathtt{T}_n\}$ is comprised of the same values as the corresponding tuple type `tuple` $\{\mathtt{T}_1, \mathtt{T}_2, \ldots \mathtt{T}_n\}$ (see (TupleValues)).
3. (StructCompIdSpace) The identifiers of the components reside together in their own namespace. (One can see this as the struct type introducing a new namespace to which the named components belong. This means that two struct types may have components with the same name without any clash, even if one is nested within the other.)
4. (StructCompatibility) Two struct types are compatible iff they have the same number of components and the types of those components are pair-wise compatible and the names of those components pair-wise equal.
5. (StructAssignability) A struct type $\mathtt{T}_1$ is assignable to another struct type $\mathtt{T}_2$ iff they have the same number of components and each component of $\mathtt{T}_1$ has the same name as, and is assignable to, its corresponding[11] component in $\mathtt{T}_2$.

Restrictions

1. (StructCompUnicity) Two components of the same struct type may not have the same name.

Related Notation

1. Given a value $\mathtt{V}_S$ of struct type `struct` $\{\mathtt{M}_1 : \mathtt{T}_1, \mathtt{M}_2 : \mathtt{T}_2, \ldots \mathtt{M}_n : \mathtt{T}_n\}$ and an identifier $\mathtt{M}_i$ with $1 \leq i \leq n$, we will write $\mathtt{V}_S@\mathtt{M}_i$ to denote the component of $\mathtt{V}_S$ of type $\mathtt{T}_i$ with the name $\mathtt{M}_i$. Note that @ is an operation on struct values, and distinct from the HLL struct accessor (`.M`) that operates on struct streams.

[11] Corresponding means here "at the same position" (the components of struct types are ordered).

8.2.3 Function Types

A function type `(int [0, 2] -> bool)` consists of the three components (of type `bool`) corresponding to the inputs `0`, `1` and `2`. The type consists of the $2^3 = 8$ values in the set $\texttt{bool}^{|\texttt{int}[0,2]|}$, *i.e.* {(false, false, false), (false, false, true), (false, true, false), (false, true, true), (true, false, false), (true, false, true), (true, true, false), (true, true, true)}.

This example of a function type adopts the point of view that the function is a composite object (much like an array) consisting of three components. However, there is also the alternative point of view of a function as a mapping between two sets; the domain and the range (or image). Seeing our example above from this point of view, we will identify the domain with the set $\{0, 1, 2\}$ and the range with the set {false, true}. A function value is thus equivalent to a mapping from a value of the function's domain to a value of the function's range[12]. For example the function value (false, true, true) from the example above corresponds to the mapping $0 \mapsto$ false, $1 \mapsto$ true, $2 \mapsto$ true.

We note that in the "object view" of a function type, the components have to be ordered in a way that allows us to find the component value corresponding to a given input value, whereas in the "mapping view" such an order is not needed. Since function values expressed as n-tuples is a completely abstract concept and not related to any HLL operation, it means that for the "object view" of function types, any order of the components will work as long as it is consistently used. That being said, some concrete HLL operations such as the definition of a function using a collection on the right hand side, written as `f := {false, true, true};` and defined in Section 12, do require that a known order is defined for the function components. This order is defined by (FunctionCompOrder) on the next page, and is only defined for ordered domain types (thus excluding functions of sort domain).

Since a function value is a mapping from a value (of the function's domain) to a value (of the function's range), it would be natural to assume that a stream of function values is a mapping from a stream to a stream. This is not the case however, and may be a source of confusion. An HLL function is a stream of combinatorial functions (*i.e.* mappings from values to values) and not a function on streams, which means it is not possible to use HLL functions to express temporal functions (*i.e.* functions which talk about the past or future values of their stream parameters). HLL functions are characterized by the following property:

for each time step k, `x = y -> f(x) = f(y)`, *regardless of the history (past or future values) of* `x` *and* `y`.

Expressions such as `f(x)` (which are introduced in Section 10.6) where both `f` and `x` are streams, should thus be understood as the point-wise application of the value of `f` on the value of `x` in each time step:

f :	f_0	f_1	f_2	...	f_n	...
x :	x_0	x_1	x_2	...	x_n	...
f(x) :	$f_0(x_0)$	$f_1(x_1)$	$f_2(x_2)$	...	$f_n(x_n)$	...

[12] A function value is thus what we call a combinatorial function.

Syntax

(FunctionSyntax)

```
<function> ::= "(" <type> {"*" <type>} "->" <type> ")"
```

Semantics

1. (FunctionType) A function is a composition of unnamed components, which are all of the same *Return type*.
2. (FunctionValues) A function type ($\mathtt{T_1} * \mathtt{T_2} * \ldots * \mathtt{T_n}$ `->` $\mathtt{T}$) consists of:
   (a) Parameter types $\mathtt{T_1}$ to $\mathtt{T_n}$. The set $\mathtt{T_1} \times \ldots \times \mathtt{T_n}$ is also called the function's *Domain*.
   (b) A return type (or component type) $\mathtt{T}$. The set $\mathtt{T}$ is also called the function's *Range*.

   The function type is comprised of the values in the Cartesian power $\mathtt{T}^{|\mathtt{T_1} \times \mathtt{T_2} \times \ldots \times \mathtt{T_n}|}$ (alternatively written as the Cartesian product $\prod_{\mathtt{i} \in \mathtt{T_1} \times \mathtt{T_2} \times \ldots \times \mathtt{T_n}} \mathtt{T}$.)
   If $\mathtt{n} > 1$ then the function type is said to be *Multivariate*.
3. (FunctionCompOrder) The components of a function are ordered iff the parameter types are ordered, *i.e.* iff they each have an order defined on their values. In that case the order of the components is the same as the order of the parameter types' values, where the first parameter type is the most significant one.
4. (FunctionDomainEquality) Two function parameter types $\mathtt{T_1}$ and $\mathtt{T_2}$ are said to be *equal* iff they are mutually assignable to each other and their sets of values are equal, *i.e.* $\mathtt{v} \in \mathtt{T_1} \leftrightarrow \mathtt{v} \in \mathtt{T_2}$.[13]
5. (FunctionCompatibility) Two function types are compatible iff they have the same number of parameter types and those are all pair-wise equal according to (FunctionDomainEquality), and their return types are compatible.
6. (FunctionAssignability) A function type ($\mathtt{T_1} * \mathtt{T_2} \ldots * \mathtt{T_n}$ `->` $\mathtt{T}$) is assignable to another function type ($\mathtt{U_1} * \mathtt{U_2} \ldots * \mathtt{U_n}$ `->` $\mathtt{U}$) iff $\mathtt{T_i}$ is equal to $\mathtt{U_i}$ (according to (FunctionDomainEquality)) for $\mathtt{i} \in [\mathtt{1}, \mathtt{n}]$ and $\mathtt{T}$ is assignable to $\mathtt{U}$.

Restrictions

1. (FunctionDomainScalar) The parameter types $\mathtt{T_i}$ for $\mathtt{i} \in [\mathtt{1}, \mathtt{n}]$ of a function type ($\mathtt{T_1} * \mathtt{T_2} * \ldots * \mathtt{T_n}$ `->` $\mathtt{T}$) shall be of scalar type.

[13] This means that *e.g.* `int` $[0, 7]$ is considered equal to `int unsigned 3`.

Related Notation

1. Given a value $\texttt{V}_\texttt{F}$ of function type $(\texttt{T}_1 * \texttt{T}_2 * \ldots * \texttt{T}_\texttt{n}$ `->` $\texttt{T})$ and values $\texttt{V}_\texttt{i}$ with $\texttt{V}_\texttt{i} \in \texttt{T}_\texttt{i}$, we will write $\texttt{V}_\texttt{F}(\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_\texttt{n})_\texttt{V}$ to denote the component (or output) value of $\texttt{V}_\texttt{F}$ of type $\texttt{T}$ corresponding to the inputs $\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_\texttt{n}$. We employ a subscript $\texttt{V}$ on the right parenthesis $()_\texttt{V}$ in order to emphasize that this is an operation on function values, and distinct from the HLL function accessor that operates on function streams. Of course, the subscript $\texttt{V}$ is not strictly necessary since function values are functions in the mathematical sense (mappings from values to values) and the operation $()_\texttt{v}$ corresponds thus to the usual function application in the mathematical sense.

8.2.4 Array Types

An array type `bool^(3)` (an array of 3 bool) is equivalent to the function type `(int [0, 2] -> bool)`, except for the way to access the components (`A[0]` vs `A(0)`).

Syntax

(ArraySyntax)

```
<array> ::= <type> "^" "(" <expr_list> ")"
```

Semantics

1. (ArrayType) An array is a composition of ordered unnamed components, which are all of the same *Base type*.
2. (ArrayValues) The type $\texttt{T\^{}(E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ is equivalent to[14] the function type $(\texttt{int}\ [0, \texttt{E}_1 - 1] * \texttt{int}\ [0, \texttt{E}_2 - 1] * \ldots * \texttt{int}\ [0, \texttt{E}_n - 1]\ \texttt{->}\ \texttt{T})$, except for the way to access the components. `T` is the base type (the component type) of the array and the $\texttt{E}_i$ are called the *Dimensions* of the array and if $\texttt{n} > 1$ then the array type is said to be *Multidimensional*.
3. (ArrayCompatibility) An array type $\texttt{T}_1\texttt{\^{}(D}_1, \texttt{D}_2, \ldots \texttt{D}_n)$ is compatible with another array type $\texttt{T}_2\texttt{\^{}(E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ if each $\texttt{D}_i = \texttt{E}_i$ and $\texttt{T}_1$ is compatible with $\texttt{T}_2$.
4. (ArrayAssignability) An array type $\texttt{T}_1\texttt{\^{}(D}_1, \texttt{D}_2, \ldots \texttt{D}_n)$ is assignable to another array type $\texttt{T}_2\texttt{\^{}(E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ if each $\texttt{D}_i = \texttt{E}_i$ and $\texttt{T}_1$ is assignable to $\texttt{T}_2$.

Restrictions

1. (ArrayDimConstant) $\mathcal{SF}(\texttt{E}_i) = 2$ for $\texttt{i} \in [1, \texttt{n}]$.
2. (ArrayDimNotNil) $M_{\text{int}}(\texttt{E}_i, k) \neq \textbf{nil}$ for $\texttt{i} \in [1, \texttt{n}]$ and all k.

Related Notation

1. Given a value $\texttt{V}_A$ of array type $\texttt{T\^{}(E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$, and values $\texttt{V}_i$ with $0 \leq \texttt{V}_i < \texttt{E}_i$ of integer type, we will write $\texttt{V}_A[\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_n]_V$ to denote the component of $\texttt{V}_A$ of type `T` at index $\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_n$. We employ a subscript `V` on the right square bracket $(]_V)$ in order to emphasize that this is an operation on array values, and distinct from the HLL array accessor that operates on array streams.

[14] This means that it is possible to consistently replace all array types by equivalent function types (while changing all accesses from $[\texttt{i}_1, \ldots \texttt{i}_n]$ to $(\texttt{i}_1, \ldots \texttt{i}_n)$) in an HLL text. However, array and function types are still not compatible with or assignable to each other as specified by (ArrayCompatibility) and (ArrayAssignability).

8.3 Named Types

Syntax

(NamedTypeSyntax)

```
<named_type> ::= <path_id>
```

Semantics

1. (NamedType) A named type (`<named_type>`) is the type that it refers to.
2. (NamedTypeCompatibility) A named type is compatible with the type it refers to.
3. (NamedTypeAssignability) A named type is mutually assignable with the type it refers to.

Restrictions

1. (NamedTypeRef) The `<path_id>` of a `<named_type>` shall refer to an existing type (in the namespace of types).

8.4 Implicit Types

Implicit types have no explicit syntax, but can appear implicitly as a result of some other syntactic construct of the language. For example a `<collection> {false, 0}` has a collection type, and an `<ite_expr> if` E_1 `then` E_2 `else` E_3 has a type which is the union of the types of E_2 and E_3.

8.4.1 Collection Types

A collection `{false, 0}` (a `<collection>`, defined in Section 12) is assignable to a variable declared with, for example, type `tuple {bool, int}` or `struct {b: bool, i:int}`. A collection `{false, true, false}` is assignable to a variable `V` declared, for example, using one of the following declarations:

```
tuple {bool, bool, bool} V; // Tuple of 3 bool
bool V[3];                  // Array of 3 bool
bool V(int [1, 3]);         // Function with 3 bool outputs
bool V(int [4, 6]);         // Same, but different indexing
bool V(ExEnum);             // + Types: enum {one, two, three} ExEnum;
```

Syntax

(empty)

Semantics

1. (CollectionReason) A collection type is the type associated with collections (`<collection>`, see Section 12).
2. (CollectionType) The type of $\{R_1, R_2, \ldots R_n\}$ where R_i is of type T_i is the collection type $\{T_1, T_2, \ldots T_n\}$, composed of n ordered components. We will count the collection types among the composite types of HLL.
3. (CollTupleCompatibility) A collection type T_1 is compatible with a tuple type T_2 iff they have the same number of components and each component of T_1 is compatible to its corresponding component in T_2.
4. (CollTupleStructAssignability) A collection type T_1 is assignable to a tuple or struct type T_2 iff they have the same number of components and each component of T_1 is assignable to its corresponding component in T_2.
5. (CollArrayAssignability) A collection type T_1 is assignable to an array type $T_2\hat{}(E)$ iff T_1 has E components and each one is assignable to T_2.
6. (CollMultiDimArrayAssignability) A collection type T_1 is assignable to a multidimensional array type $T_2\hat{}(E_1, E_2, \ldots E_n)$ $(n > 1)$ iff T_1 has E_1 components and each one is assignable to $T_2\hat{}(E_2, \ldots E_n)$.
7. (CollFuncAssignability) A collection type T_1 is assignable to a function type $(T_2$ `->` $T_3)$ iff T_2 is an ordered type and T_1 has $|T_2|$ components and each one is assignable to T_3.
8. (CollMultiVarFuncAssignability) A collection type T_1 is assignable to a multivariate function type $(T_2 * T_3 * \ldots * T_{n-1}$ `->` $T_n)$ $(n > 3)$ iff T_2 is an ordered type and T_1 has $|T_2|$ components and each one is assignable to $(T_3 * \ldots * T_{n-1}$ `->` $T_n)$.

Restrictions

(empty)

8.4.2 Unsized Types

In HLL, it is possible to restrict the set of values of integer inputs and memories, as well as other declared stream variables, either by using a "range" (e.g. `int [0, 7]`) or an "implementation" (e.g. `int unsigned 3`). However, for arbitrary integer expressions, the only type used is `int` without restriction. The *Unsized copy* of a type (defined below) is used to remove its size restriction in certain cases, for example when computing the union type of the two branches of an if-then-else (see Sections 8.4.3 and 10.1).

Syntax

(empty)

Semantics

1. (UnsizedInteger) The unsized copy of an integer type `T` is the type `int`.
2. (UnsizedScalar) The unsized copy of a non-integer scalar type `T` is `T`.
3. (UnsizedComposite) The unsized copy of a composite type `T` is a type `T'`, in all respects the same as `T`, but with each component type replaced by its unsized copy.[15]

Restrictions

(empty)

[15] To give two examples, the unsized copy of `tuple {int signed` E_1`, int [`E_2`, `E_3`]}` is `tuple {int, int}`, and the unsized copy of (`int [`E_1`, `E_2`] -> int [`E_3`, `E_4`]`) is (`int [`E_1`, `E_2`] -> int`).

8.4.3 Union Types

Syntax

(empty)

Semantics

1. (UnionScalar) The union type of `n` compatible non-sort scalar types $\texttt{T}_1 \ldots \texttt{T}_n$ is the unsized copy of $\texttt{T}_1$.
2. (UnionSort) The union type of `n` sort or sort union types $\texttt{T}_1 \ldots \texttt{T}_n$ (*i.e.* each $\texttt{T}_i$ may be either a sort type or another sort union type) is a special "sort union type" $\texttt{T}_u$. We will count the sort union type among the scalar types of HLL.
3. (UnionComposite) The union type of `n` compatible composite types $\texttt{T}_1 \ldots \texttt{T}_n$, which are not collection types, is a type $\texttt{T}_r$, in all respects the same as any of the $\texttt{T}_i$, but where each component type of $\texttt{T}_r$ is the union type of the `n` corresponding component types of $\texttt{T}_1 \ldots \texttt{T}_n$.
4. (UnionTupleCollection) The union type of a tuple type $\texttt{T}_1$ and a compatible collection type $\texttt{T}_2$ is a tuple type $\texttt{T}_r$, in all respects the same as $\texttt{T}_1$, but where each component type of $\texttt{T}_r$ is the union type of the corresponding component types of $\texttt{T}_1$ and $\texttt{T}_2$. This union is needed to represent the type of `SELECT` expressions where the default value is a collection, see (QuantSelectType).
5. (SortUnionCompatibility) A sort union type is compatible with both sort types and other sort union types.
6. (SortUnionAssignability) A sort union type $\texttt{T}_u$ of the types $\texttt{T}_1 \ldots \texttt{T}_n$ is assignable to a sort type $\texttt{T}_s$ iff each $\texttt{T}_i$ for $\texttt{i} \in [\texttt{1}, \texttt{n}]$ is assignable to $\texttt{T}_s$.[16]

Restrictions

(empty)

[16] Note that this definition relies on a recursion, since the $\texttt{T}_i$ may themselves be sort unions. The recursion is well-founded since at the base case level we only have sort types.

9 Accessors

Accessors are used to designate components of streams of composite type.

Syntax

(AccessorSyntax)

```
<accessor> ::= "." <id>
             | "." <int_literal>
             | "[" <expr_list> "]"
             | "(" [<expr_list>] ")"
```

Forward References

1. `<int_literal>` is defined in Section 10.7.2.

Semantics

1. (AccStruct) `.M` (an `<id>`) designates a struct component named `M`.
2. (AccTuple) `.N` (an `<int_literal>`) designates the `(N+1):th` component of a tuple (the indexing is 0-based).
3. (AccArray) At time step k and relative to an array type $\texttt{T\^{}}(\texttt{D}_1, \texttt{D}_2, \ldots \texttt{D}_n)$, $[\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n]$ designates the array component of type `T` at the index given by $[M_{\text{int}}(\texttt{E}_1, k), M_{\text{int}}(\texttt{E}_2, k), \ldots M_{\text{int}}(\texttt{E}_n, k)]_{\texttt{V}}$, or if there is no such component (the index is out of bounds or **nil**), it designates **nil**.
4. (AccFunction) At time step k and relative to a function type $(\texttt{T}_1 * \texttt{T}_2 * \ldots * \texttt{T}_n \texttt{ -> } \texttt{T}_r)$, $(\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ designates the function output of type $\texttt{T}_r$ corresponding to the inputs $(M_{\texttt{T}_1}(\texttt{E}_1, k), M_{\texttt{T}_2}(\texttt{E}_2, k), \ldots M_{\texttt{T}_n}(\texttt{E}_n, k))_{\texttt{V}}$, or if there is no such output (some input is out of the function domain or **nil**), it designates **nil**.

Restrictions

1. (ArrayIndexInteger) Each $\texttt{E}_i$ of an accessor $[\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n]$ shall be of integer type.
2. (FunctionInputScalar) Each $\texttt{E}_i$ of an accessor $(\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_n)$ shall be of scalar type.

10 Expressions

Syntax

(ExprSyntax)

```
<expr>          ::= <ite_expr>
                  | <lambda_expr>
                  | <binop_expr>
                  | <membership_expr>
                  | <unop_expr>
                  | <proj_expr>
```

Semantics

1. (Expr) An *Expression* `E` (an `<expr>`) is a stream (of values) of some type `T`.
2. (EmptyScalarTypeNil) If some scalar component of `T` (or `T` itself) is an empty scalar type, then the corresponding component of a stream expression `E` (or `E` itself) of type `T` will carry the value **nil** in each time step.
3. (ExprPrecedence) The precedence of expressions is given below in order from lowest to highest (same line means same precedence):
   (a) `<ite_expr>`, `<lambda_expr>`
   (b) `<binop_expr>`, `<membership_expr>`
   (c) `<unop_expr>`
   (d) `<proj_expr>`

Restrictions

(empty)

Related Notation

1. We will write $\texttt{E<V}_1 := \texttt{R}_1, \texttt{V}_2 := \texttt{R}_2, \ldots \texttt{V}_n := \texttt{R}_n\texttt{>}$ to denote substitution, *i.e.* the process of replacing all free occurrences of the variables $\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_n$ in the expression `E` with expressions $\texttt{R}_1, \texttt{R}_2, \ldots \texttt{R}_n$.

   As an example $(\texttt{x} + \texttt{y} = \texttt{z})\texttt{<z} := \texttt{5>}$ is equivalent to $\texttt{x} + \texttt{y} = \texttt{5}$. By contrast, $(\texttt{SOME z} : [0, 4]\ (\texttt{x} + \texttt{y} = \texttt{z}))\texttt{<z} := \texttt{5>}$ is equivalent to $\texttt{SOME z} : [0, 4]\ (\texttt{x} + \texttt{y} = \texttt{z})$ (no substitution performed) since the quantifier variable `z` is not free in this case (it is bound by the quantification).

10.1 If-Then-Else Expressions

Syntax

(IteExprSyntax)

```
<ite_expr> ::= "if"   <expr> "then" <expr>
               {"elif" <expr> "then" <expr>}
                "else" <expr>
```

Semantics

1. (Elif) `elif` is equivalent to `else if`.
2. (IfThenElse) In each time step, `if` $\mathtt{E}_1$ `then` $\mathtt{E}_2$ `else` $\mathtt{E}_3$ evaluates to 1) **nil** if $\mathtt{E}_1$ is **nil**, 2) $\mathtt{E}_2$ if $\mathtt{E}_1$ is true and 3) $\mathtt{E}_3$, otherwise ($\mathtt{E}_1$ is false). The type `T` of the expression is the union type of the type $\mathtt{T}_2$ of $\mathtt{E}_2$ and the type $\mathtt{T}_3$ of $\mathtt{E}_3$. Formally:

   $M_{\mathtt{T}}(\texttt{if } \mathtt{E}_1 \texttt{ then } \mathtt{E}_2 \texttt{ else } \mathtt{E}_3, k) =$

$$= \begin{cases} \textbf{nil} & \text{if } M_{\text{bool}}(\mathtt{E}_1, k) = \textbf{nil} \\ M_{\mathtt{T}_2}(\mathtt{E}_2, k) & \text{if } M_{\text{bool}}(\mathtt{E}_1, k) = \text{true} \\ M_{\mathtt{T}_3}(\mathtt{E}_3, k) & \text{otherwise } (M_{\text{bool}}(\mathtt{E}_1, k) = \text{false}) \end{cases}$$

Static Flag

1. (IteExprStaticFlag)
   $\mathcal{SF}(\texttt{if } \mathtt{E}_1 \texttt{ then } \mathtt{E}_2 \texttt{ else } \mathtt{E}_3) = \texttt{min}(\mathcal{SF}(\mathtt{E}_1), \mathcal{SF}(\mathtt{E}_2), \mathcal{SF}(\mathtt{E}_3))$.

Restrictions

1. (IteCondBool) The type of $\mathtt{E}_1$ shall be `bool`.
2. (IteBranchesCompatible) The types of $\mathtt{E}_2$ and $\mathtt{E}_3$ shall be compatible.

10.2 Lambda Expressions

Lambda expressions can be used to build unnamed streams of array or function type. For example "`lambda[3]:[i] := i = 1`" is an array of the 3 Boolean constants false (at index 0), true (at index 1), false (at index 2).

Lambda expressions can, together with a definition, be used to build recursive array or function definitions. As an example, the Fibonacci numbers are calculated by a recursion below (all proof obligations are valid).

```
Declarations:
  int fibonacci(int);
Definitions:
  fibonacci := lambda(int): (i) := if i <= 2 then 1
                                   else fibonacci(i - 1) +
                                        fibonacci(i - 2);
Proof Obligations:
  fibonacci(1) = 1;
  fibonacci(2) = 1;
  fibonacci(3) = 2;
  fibonacci(4) = 3;
  fibonacci(5) = 5;
```

Allowing recursion in this manner provides HLL with a powerful tool, but unfortunately it also means that it becomes possible to express undecidable problems in HLL, since it is possible to build recursions that do not terminate. Whether or not a recursion will terminate depends on the reasoning power of tools implementing HLL, and is thus outside the scope of this document.

A specificity of lambda expressions is how their type is computed. Two lambda expressions which at a glance may look as if of different type, may in fact be of the same type. The type of a lambda expression is computed according to (LambdaType) on the next page. In the following example, the two lambda expressions that are being compared are of the same type (the type `int^(3)^(4)`), and the proof obligation is valid.

```
Proof Obligations:
 (lambda[4][3]:[i] := (lambda[3]:[j] := 0)) =
 (lambda[4]:   [i] := (lambda[3]:[j] := 0)); // Valid PO
```

Lambda expressions are – just as any HLL function (or array) as discussed in Section 8.2.3 – streams of combinatorial functions (mappings from values to values). The formal definition of lambda arrays and functions (in (LambdaArray) and (LambdaFunction) below) defines, for a given time step k, one such combinatorial function. The combinatorial function takes as inputs the values V_1 to V_n and returns:

1. The value of the right hand side expression at time step k with the formal parameters substituted with their corresponding values, if the latter are within the domain of the array or function, or
2. **nil** otherwise.

Syntax

(LambdaExprSyntax)

```
<lambda_expr> ::= "lambda" {<declarator_suffix>}+ ":"
                  {<formal_param>}+ ":=" <expr>

<formal_param>  ::= "[" <id_list> "]"
                  | "(" <id_list> ")"
```

Semantics

1. (LambdaScope) A `<lambda_expr>` introduces a local scope in the namespace of streams, called a *Lambda scope* that starts at the `"lambda"` keyword and continues to the end of the expression. Parameters (`<formal_param>`) declared within the scope must be unique, but can hide other variables or parameters above it.

2. (LambdaType) The type `T` of a lambda expression `lambda DS : FP := E` where `E` is an expression of type $\mathtt{T}_1$ is equal to $\mathtt{calc\_type}(\mathtt{T}_2, \mathtt{DS})$, where the type $\mathtt{T}_2$ is calculated by solving the following equation: $\mathtt{T}_1 = \mathtt{calc\_type}(\mathtt{T}_2, \mathtt{DS\_suffix})$ where `DS_suffix` is the `N` last elements of the list `DS` where `N` is the difference in length of the list `DS` and the list `FP`, *i.e.* $\mathtt{N} = |\mathtt{DS}| - |\mathtt{FP}|$. The function `calc_type` is defined in Section 7.

   Note that if the length of the lists `DS` and `FP` is the same (*i.e.* $\mathtt{N} = 0$), then $\mathtt{DS\_suffix} = \{\}$ (the empty list) and $\mathtt{T}_1 = \mathtt{T}_2$.

3. (LambdaMultFormalParam) $\mathtt{lambda}\ \mathtt{D}_1 \ldots \mathtt{D}_n : \mathtt{P}_1 \ldots \mathtt{P}_n \mathtt{:=}\ \mathtt{E}$ is reducible[17] to $\mathtt{lambda}\ \mathtt{D}_1 \ldots \mathtt{D}_{n-1} : \mathtt{P}_1 \ldots \mathtt{P}_{n-1} \mathtt{:=}\ (\mathtt{lambda}\ \mathtt{D}_n : \mathtt{P}_n \mathtt{:=}\ \mathtt{E})$.

4. (LambdaDeclSuffixOverhang) $\mathtt{lambda}\ \mathtt{D}_1 \ldots \mathtt{D}_n : \mathtt{P}_1 \ldots \mathtt{P}_k \mathtt{:=}\ \mathtt{E}$ with $\mathtt{n} > \mathtt{k}$ is (if well-typed) equivalent to $\mathtt{lambda}\ \mathtt{D}_1 \ldots \mathtt{D}_k : \mathtt{P}_1 \ldots \mathtt{P}_k \mathtt{:=}\ \mathtt{E}$.

5. (LambdaArray) $\mathtt{lambda}[\mathtt{E}_1, \ldots \mathtt{E}_n] : [\mathtt{i}_1, \ldots \mathtt{i}_n] \mathtt{:=}\ \mathtt{E}$, where `E` is of type $\mathtt{T}_E$, and $\mathtt{i}_j$ for $\mathtt{j} \in [1, \mathtt{n}]$ is, by definition, of type `int`, is an expression of type $\mathtt{T}_E\hat{}(\mathtt{E}_1, \ldots \mathtt{E}_n)$ such that:

   $$M_{\mathtt{T}_E\hat{}(\mathtt{E}_1,\ldots\mathtt{E}_n)}(\mathtt{lambda}[\mathtt{E}_1, \ldots \mathtt{E}_n] : [\mathtt{i}_1, \ldots \mathtt{i}_n] \mathtt{:=}\ \mathtt{E}, k)[\mathtt{V}_1, \ldots \mathtt{V}_n]_\mathtt{V} =$$

   $$= \begin{cases} M_{\mathtt{T}_E}(\mathtt{E<i}_1 \mathtt{:=}\ \mathtt{V}_1, \ldots \mathtt{i}_n \mathtt{:=}\ \mathtt{V}_n\mathtt{>}, k) & \text{if } \forall \mathtt{j} \in [1, \mathtt{n}] : 0 \leq \mathtt{V}_j < \mathtt{E}_j \\ \textbf{nil} & \text{otherwise} \end{cases}$$

   (Note that the values $\mathtt{V}_1$ to $\mathtt{V}_n$ are implicitly converted to constant streams before they are substituted for the formal parameters $\mathtt{i}_1$ to $\mathtt{i}_n$.)

[17] $\mathtt{lambda}[1] : [\mathtt{i}] := (\mathtt{lambda}[1] : [\mathtt{i}] := \mathtt{i})$ is not reducible to $\mathtt{lambda}[1][1] : [\mathtt{i}][\mathtt{i}] := \mathtt{i}$ (since the latter expression is not legal), so the two are not equivalent.

6. (LambdaFunction) $\texttt{lambda}(\texttt{T}_1, \ldots \texttt{T}_n) : (\texttt{i}_1, \ldots \texttt{i}_n)\texttt{:= E}$, where E is of type $\texttt{T}_E$, and $\texttt{i}_j$ for $\texttt{j} \in [1, n]$ is, by definition, of type $\texttt{T}_j$, is an expression of type $(\texttt{T}_1 * \ldots * \texttt{T}_n \texttt{ -> } \texttt{T}_E)$ such that:

   $M_{(\texttt{T}_1 * \ldots * \texttt{T}_n \texttt{ -> } \texttt{T}_E)}(\texttt{lambda}(\texttt{T}_1, \ldots \texttt{T}_n) : (\texttt{i}_1, \ldots \texttt{i}_n)\texttt{:= E}, k)(\texttt{V}_1, \ldots \texttt{V}_n)_\texttt{V} =$

   $$= \begin{cases} M_{\texttt{T}_E}(\texttt{E<i}_1\texttt{:= V}_1, \ldots \texttt{i}_n\texttt{:= V}_n\texttt{>}, k) & \text{if } \forall \texttt{j} \in [1, n] : \texttt{V}_j \in \texttt{T}_j \\ \textbf{nil} & \text{otherwise} \end{cases}$$

   (Note that the values $\texttt{V}_1$ to $\texttt{V}_n$ are implicitly converted to constant streams before they are substituted for the formal parameters $\texttt{i}_1$ to $\texttt{i}_n$.)

Static Flag

1. (LambdaStaticFlag) $\mathcal{SF}(\texttt{<lambda\_expr>}) = \texttt{0}$.
2. (FormalParamStaticFlag) $\mathcal{SF}(\texttt{i}) = \texttt{1}$ for a lambda parameter i (*e.g.* lambda[1] : [i] := <expr>).

Restrictions

1. (LambdaParamUnicity) Two parameters of a lambda expression may not have the same name.
2. (LambdaParamsBound) $|\texttt{DS}| \geq |\texttt{FP}|$ (the number of elements of the list DS shall be greater than or equal to the number of elements of the list FP) for an expression lambda DS : FP := E.
3. (LambdaParamsMatch) Each element F (a <formal_param>) of the list FP in an expression lambda DS : FP := E, where F itself is a list (an <id_list>), shall contain the same number of elements (of form <id>) as the element D (a <declarator_suffix> and itself a list; either an <expr_list> or a <type_list) of the corresponding position in the list DS.

   Furthermore, if D is on the form "[" <expr_list> "]" then F shall be on the form "[" <id_list> "]", and if D is on the form "(" <type_list> ")" then F shall be on the form "(" <id_list> ")".
4. (LambdaTypeCheck) There shall be exactly one solution to the equation $\texttt{T}_1 = \texttt{calc\_type}(\texttt{T}_2, \texttt{DS\_suffix})$ of (LambdaType).

10.3 Binop Expressions

Syntax

(BinopSyntax)

```
<binop_expr> ::= <expr> <binop> <expr>
<binop>      ::= "#" | "&"  | "#!" | "->" | "<->"
               | ">" | ">=" | "<"  | "<="
               | "=" | "==" | "!=" | "<>"
               | "+" | "-"  | "*"  | "^" | "<<" | ">>"
               | "/" | "/>" | "/<" | "%"
```

Semantics

1. (BoolAnd) E_1 `&` E_2 (Boolean and) is equivalent to `~(~`E_1 `# ~`E_2`)` (De Morgan's law).
2. (BoolXor) E_1 `#!` E_2 (Boolean exclusive or) is equivalent to `~(`E_1 `<->` E_2`)`.
3. (BoolImpl) E_1 `->` E_2 (Boolean implication) is equivalent to `~`E_1 `#` E_2.
4. (BoolEquiv) E_1 `<->` E_2 (Boolean equivalence) is reducible to E_1 `=` E_2.
5. (IntGt) E_1 `>` E_2 (greater than) is equivalent to E_2 `<` E_1.
6. (IntGte) E_1 `>=` E_2 (greater than or equal to) is equivalent to `~(`E_1 `<` E_2`)`.
7. (IntLte) E_1 `<=` E_2 (less than or equal to) is equivalent to E_2 `>=` E_1.
8. (OpNeq) E_1 `!=` E_2 and E_1 `<>` E_2 are both equivalent to `~(`E_1 `=` E_2`)`.
9. (OpEqEq) `==` is equivalent to `=`.
10. (IntSub) E_1 `-` E_2 (subtraction) is equivalent to E_1 `+ (-`E_2`)`.
11. (IntLeftShift) E_1 `<<` E_2 (left shift) is reducible to E_1 `*` `(2 ^` E_2`)`.
12. (IntRightShift) E_1 `>>` E_2 (right shift) is reducible to E_1 `/>` `(2 ^` E_2`)`.
13. (IntFloorDiv) E_1 `/>` E_2 (floor division) represents the biggest integer smaller than or equal to the rational E_1/E_2. Formally, it is equivalent to:

    ```
    if (E1 < 0) = (E2 < 0) # E1 % E2 = 0
     then E1 / E2
     else E1 / E2 - 1
    ```

14. (IntCeilDiv) E_1 `/<` E_2 (ceiling division) represents the smallest integer bigger than or equal to the rational E_1/E_2. Formally, it is equivalent to:

    ```
    if (E1 < 0) = (E2 < 0) & E1 % E2 != 0
     then E1 / E2 + 1
     else E1 / E2
    ```

15. (IntRem) `E1 % E2` (remainder) is equivalent to `E1 - (E1 / E2) * E2`.

Core Constructs:

16. (BoolOr) `E1 # E2` (Boolean or) is a stream of type `bool` for which $M_{\text{bool}}(\texttt{E}_1 \texttt{ \# } \texttt{E}_2, n) = M_{\text{bool}}(\texttt{E}_1, n) \vee_{\textbf{nil}} M_{\text{bool}}(\texttt{E}_2, n)$ holds. The operator $\vee_{\textbf{nil}}$ is the usual Boolean OR operator extended to three-valued logic according to the following table:

$A \vee_{\textbf{nil}} B$		B		
		false	**nil**	true
	false	false	**nil**	true
A	**nil**	**nil**	**nil**	true
	true	true	true	true

17. (IntLt) `E1 < E2` (less than) is a stream of type `bool` for which $M_{\text{bool}}(\texttt{E}_1 \texttt{ < } \texttt{E}_2, n) = M_{\text{int}}(\texttt{E}_1, n) <_{\textbf{nil}} M_{\text{int}}(\texttt{E}_2, n)$ holds. The operator $<_{\textbf{nil}}$ is the usual "strictly less than" comparison operator defined on integers, and extended for the **nil** case according to the following table:

$A <_{\textbf{nil}} B$		B	
		nil	Y
A	**nil**	**nil**	**nil**
	X	**nil**	$X < Y$

18. (IntAdd) `E1 + E2` (addition) is a stream of type `int` for which $M_{\text{int}}(\texttt{E}_1 \texttt{ + } \texttt{E}_2, n) = M_{\text{int}}(\texttt{E}_1, n) +_{\textbf{nil}} M_{\text{int}}(\texttt{E}_2, n)$ holds. The operator $+_{\textbf{nil}}$ is the usual integer addition operator extended for the **nil** case according to the following table:

$A +_{\textbf{nil}} B$		B	
		nil	Y
A	**nil**	**nil**	**nil**
	X	**nil**	$X + Y$

19. (IntMul) `E1 * E2` (multiplication) is a stream of type `int` for which $M_{\text{int}}(\texttt{E}_1 \texttt{ * } \texttt{E}_2, n) = M_{\text{int}}(\texttt{E}_1, n) *_{\textbf{nil}} M_{\text{int}}(\texttt{E}_2, n)$ holds. The operator $*_{\textbf{nil}}$ is the usual integer multiplication operator extended for the **nil** case according to the following table:

$A *_{\textbf{nil}} B$		B	
		nil	Y
A	**nil**	**nil**	**nil**
	X	**nil**	$X * Y$

20. (IntDiv) `E1 / E2` (integer division) is a stream of type `int` for which $M_{\text{int}}(\texttt{E}_1 \texttt{ / } \texttt{E}_2, n) = M_{\text{int}}(\texttt{E}_1, n) /_{\textbf{nil}} M_{\text{int}}(\texttt{E}_2, n)$ holds. The operator $/_{\textbf{nil}}$ is the usual integer division operator (this means a truncating division; *i.e.* division with the fractional part omitted) extended for the **nil** case according to the following table:

$A/_{\mathbf{nil}}B$		B		
		0	**nil**	Y
A	**nil**	**nil**	**nil**	**nil**
	X	**nil**	**nil**	X/Y

21. (IntExp) `E1 ^ E2` (exponentiation) is a stream of type `int` for which $M_{\text{int}}(\texttt{E}_1 \texttt{ \^{} } \texttt{E}_2, n) = M_{\text{int}}(\texttt{E}_1, n)$ $\hat{}_{\mathbf{nil}}$ $M_{\text{int}}(\texttt{E}_2, n)$ holds. The operator $\hat{}_{\mathbf{nil}}$ is defined in the following table:

$A \hat{}_{\mathbf{nil}} B$		B			
		< 0	0	**nil**	Y
	0	**nil**	1	**nil**	0
A	**nil**	**nil**	**nil**	**nil**	**nil**
	X	$\frac{1}{X^{\lvert B\rvert}}$	1	**nil**	X^Y

Note: The division operation $\frac{1}{X^{|B|}}$ refers to integer division, which means that this expression can only take the values -1, 0 or 1.

22. (OpEqScalar) `E1 = E2` (scalar equality), where `E1` and `E2` are of scalar compatible types `T1` and `T2`, is a stream of type `bool` for which $M_{\text{bool}}(\texttt{E}_1 \texttt{ = } \texttt{E}_2, n) = (M_{\texttt{T}_1}(\texttt{E}_1, n) =_{\mathbf{nil}} M_{\texttt{T}_2}(\texttt{E}_2, n))$ holds. The operator $=_{\mathbf{nil}}$ is the usual equality operator extended for the **nil** case according to the following table:

$A =_{\mathbf{nil}} B$		B	
		nil	Y
A	**nil**	**nil**	**nil**
	X	**nil**	$X = Y$

23. (OpEqCompositeUniDim) `E1 = E2` (composite unidimensional equality), where `E1` and `E2` are of composite compatible types $\texttt{T}_{\texttt{E}_1}$ and $\texttt{T}_{\texttt{E}_2}$, which are neither multidimensional array types nor multivariate function types, is equivalent to:

$$\texttt{E}_1\texttt{A}_0 \texttt{ = } \texttt{E}_2\texttt{A}_0 \texttt{ \& } \texttt{E}_1\texttt{A}_1 \texttt{ = } \texttt{E}_2\texttt{A}_1 \texttt{ \& } \ldots \texttt{ \& } \texttt{E}_1\texttt{A}_{n-1} \texttt{ = } \texttt{E}_2\texttt{A}_{n-1}$$

where:

$$\texttt{A}_i = \begin{cases} \texttt{.i} & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } \texttt{tuple } \{\texttt{T}_0, \ldots \texttt{T}_{n-1}\} \\ \texttt{.M}_i & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } \texttt{struct } \{\texttt{M}_0 : \texttt{T}_0, \ldots \texttt{M}_{n-1} : \texttt{T}_{n-1}\} \\ \texttt{[i]} & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } \texttt{T\^{}(n)} \\ \texttt{(V}_i\texttt{)} & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } (\texttt{T}_\texttt{d} \texttt{ -> T}) \text{ and } \texttt{T}_\texttt{d} \text{ is } \{\texttt{V}_0, \ldots \texttt{V}_{n-1}\} \end{cases}$$

24. (OpEqCompositeMultiDim) $\texttt{E}_1 \texttt{ = } \texttt{E}_2$ (composite multidimensional equality), where $\texttt{E}_1$ and $\texttt{E}_2$ are of composite compatible types $\texttt{T}_{\texttt{E}_1}$ and $\texttt{T}_{\texttt{E}_2}$, which are either multidimensional array types or multivariate function types, is equivalent to:

$$\begin{array}{llllllll}
\texttt{E}_1\texttt{A}_{0\ldots00} & = & \texttt{E}_2\texttt{A}_{0\ldots00} & \texttt{\&} \ \ldots \ \texttt{\&} & \texttt{E}_1\texttt{A}_{0\ldots0\texttt{m}_k} & = & \texttt{E}_2\texttt{A}_{0\ldots0\texttt{m}_k} & \texttt{\&} \\
\texttt{E}_1\texttt{A}_{0\ldots10} & = & \texttt{E}_2\texttt{A}_{0\ldots10} & \texttt{\&} \ \ldots \ \texttt{\&} & \texttt{E}_1\texttt{A}_{0\ldots1\texttt{m}_k} & = & \texttt{E}_2\texttt{A}_{0\ldots1\texttt{m}_k} & \texttt{\&} \\
\vdots & & \vdots & \ddots & \vdots & & \vdots & \\
\texttt{E}_1\texttt{A}_{0\ldots\texttt{m}_{k-1}0} & = & \texttt{E}_2\texttt{A}_{0\ldots\texttt{m}_{k-1}0} & \texttt{\&} \ \ldots \ \texttt{\&} & \texttt{E}_1\texttt{A}_{0\ldots\texttt{m}_{k-1}\texttt{m}_k} & = & \texttt{E}_2\texttt{A}_{0\ldots\texttt{m}_{k-1}\texttt{m}_k} & \texttt{\&} \\
\vdots & & \vdots & \ddots & \vdots & & \vdots & \\
\texttt{E}_1\texttt{A}_{\texttt{m}_1\texttt{m}_2\ldots0} & = & \texttt{E}_2\texttt{A}_{\texttt{m}_1\texttt{m}_2\ldots0} & \texttt{\&} \ \ldots \ \texttt{\&} & \texttt{E}_1\texttt{A}_{\texttt{m}_1\texttt{m}_2\ldots\texttt{m}_k} & = & \texttt{E}_2\texttt{A}_{\texttt{m}_1\texttt{m}_2\ldots\texttt{m}_k} &
\end{array}$$

where:

$\texttt{A}_{\texttt{i}_1\ldots\texttt{i}_k} =$

$$= \begin{cases} [\texttt{i}_1, \ldots \texttt{i}_k] & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } \texttt{T\^{}}(\texttt{m}_1 \texttt{ + } 1, \ldots \texttt{m}_k \texttt{ + } 1) \\ (\texttt{V}_{\texttt{i}_1}, \ldots \texttt{V}_{\texttt{i}_k}) & \text{if } \texttt{T}_{\texttt{E}_1} \text{ is } (\texttt{T}_1 * \ldots \texttt{T}_k \texttt{ -> T}) \text{ and } \texttt{T}_{\texttt{j}\in[1,\texttt{k}]} \text{ is } \{\texttt{V}_0, \ldots \texttt{V}_{\texttt{m}_\texttt{j}}\} \end{cases}$$

Precedence and Associativity:

25. (BinopGrouping) The precedence of the operators is given below in order from lowest to highest, together with their associativity. Same line means same precedence.

Precedence	*Associativity*
`<-> #!`	left
`->`	right
`#`	left
`&`	left
`> >= < <= = == != <>`	left
`<< >>`	left
`+ -`	left
`* / /< /> %`	left
`^`	right

Static Flag

1. (BinopStaticFlag) $\mathcal{SF}(\texttt{E}_1 \texttt{ <binop> } \texttt{E}_2) = \texttt{min}(\mathcal{SF}(\texttt{E}_1), \mathcal{SF}(\texttt{E}_2))$.

Restrictions

1. (BoolOrEquivOperandsBool) The operands of `# <->` shall be of type `bool`.
2. (EqOperandsFiniteCompatible) The operands of `=` shall be either of compatible scalar types or of compatible composite types with finite numbers of scalar components (either directly or indirectly via other composite components).

3. (IntCoreBinopOperandsInt) The operands of `< + * / ^` shall be of integer type.
4. (SecondShiftOperandStatic) $\mathcal{SF}(\texttt{E}_2) \geq \texttt{1}$ for an expression $\texttt{E}_1 \circ \texttt{E}_2$ with $\circ \in \{\texttt{<<}, \texttt{>>}\}$.
5. (SecondShiftOperandNonNegative) $\texttt{E}_2 \geq \texttt{0}$ for an expression $\texttt{E}_1 \circ \texttt{E}_2$ with $\circ \in \{\texttt{<<}, \texttt{>>}\}$.

10.4 Membership Expressions

Membership (or elementhood) expressions can be used to test whether the value of some stream expression belongs to a domain, where a domain is a stream of sets of values.

Syntax

(MembershipSyntax)

```
<membership_expr> ::= <expr> ":" <domain>
<domain>          ::= <range>
                    | <type_domain>
<type_domain>     ::= <named_type>
                    | "bool"
                    | "int"
```

Semantics

1. (Domain) A *Domain* `D` (`<domain>`) is a stream of (possibly different) sets of values. We will write $\texttt{D}_k$ to denote the set of values of `D` at time step k.
2. (DomainAsType) A `<domain>` which is a `<type_domain>` consists in each time step of the set of values of the type `T` which the `<type_domain>` refers to. Such a `<domain> D` is type compatible with the type `T`.
3. (DomainAsRange) A `<domain>` which is a `<range>` $[\texttt{E}_1,\ \texttt{E}_2]$ is in each time step k the set of values given by the range $[M_{\text{int}}(\texttt{E}_1, k), M_{\text{int}}(\texttt{E}_2, k)]$. Such a `<domain>` is type compatible with the type `int`.
4. (Membership) `E : D`, where `E` is of type $\texttt{T}_\texttt{E}$, is an expression of type `bool` such that:

$$M_{\text{bool}}(\texttt{E} : \texttt{D}, k) = \begin{cases} \textbf{nil} & \text{if } M_{\texttt{T}_\texttt{E}}(\texttt{E}, k) = \textbf{nil} \\ \text{true} & \text{if } M_{\texttt{T}_\texttt{E}}(\texttt{E}, k) \in \texttt{D}_k \\ \text{false} & \text{otherwise} \end{cases}$$

5. (MembershipPrecedence) The operator `:` has the same precedence as the operator `=`.

Static Flag

1. (MembershipStaticFlag) $\mathcal{SF}(\texttt{<membership\_expr>}) = \texttt{0}$.
2. (RangeDomainStaticFlag) $\mathcal{SF}([\texttt{E}_1,\ \texttt{E}_2]) = \texttt{min}(\mathcal{SF}(\texttt{E}_1), \mathcal{SF}(\texttt{E}_2))$.
3. (TypeDomainStaticFlag) The static flag of a `<domain>` which refers to a type is 2.

Restrictions

1. (DomainScalar) A `<domain> D` shall be type compatible with a scalar type.
2. (MembershipDomainCompatible) The `<domain> D` of an expression `E : D` shall be type compatible with the type of `E`.

10.5 Unop Expressions

Syntax

(UnopSyntax)

```
<unop_expr> ::= <unop> <expr>
<unop>      ::= "~" | "-"
```

Semantics

1. (BoolNeg) `~E` (Boolean negation) is **nil**, true, or false in time step n if `E` is respectively **nil**, false or true in time step n. The type of the expression is `bool`.
2. (IntNeg) `-E` (integer negation) is the additive inverse of `E` (meaning that `E + (-E) = 0` holds). The expression is **nil** in time step n if `E` is **nil** in time step n. The type of the expression is `int`.

Static Flag

1. (UnopStaticFlag) $\mathcal{SF}$(`<unop> E`) = $\mathcal{SF}$(`E`).

Restrictions

1. (BoolNegOperandBool) The operand of `~` shall be of type `bool`.
2. (IntNegOperandInt) The operand of `-` shall be of integer type.

10.6 Projection Expressions

Projection expressions can be used to select components of streams of composite type. For example `A[4]` selects the array component at index 4, `f(true)` selects the function component (or output) that corresponds to input `true`, and so on. An important thing to note is that `f(x)`, where `x` is an arbitrary stream expression, selects, in time step k, the function output that corresponds to the value of `x` at time step k, *i.e.* $M(\mathtt{x}, k)$. This means that functions are characterized by the following property:

for each time step k, `x = y -> f(x) = f(y)`, *regardless of the history (past or future values) of* `x` *and* `y`.

Hence, streams of function type shall be understood as streams of combinatorial functions, and not as functions on streams.

Syntax

(ProjExprSyntax)

```
<proj_expr> ::= <closed_expr> { <accessor> }
```

Forward References

1. `<closed_expr>` is defined in Section 10.7.

Semantics

1. (ProjMultipleAcc) A projection expression $\mathtt{E}\ \mathtt{A}_1\ \mathtt{A}_2 \ldots \mathtt{A}_n$ is equivalent to $(\ldots((\mathtt{E}\ \mathtt{A}_1)\ \mathtt{A}_2)\ldots)\ \mathtt{A}_n$.
2. (Projection) A projection expression `E A` where `E` is a `<closed_expr>` of composite type `T` and `A` an `<accessor>` compatible with `T` is equivalent to the component of `E` designated by `A`. If `A` does not designate a valid component[18] of `E` in time step n, then the expression is equivalent to **nil** in time step n. Note that the semantics of accessors is detailed in Section 9.
3. (ProjArrayFunc) This follows directly from Semantic item 4 of Section 9 but is stated again here as a reminder due to its importance: $M_{\mathtt{T}}(\mathtt{E}(\mathtt{E}_1, \mathtt{E}_2, \ldots \mathtt{E}_n), k) = M_{(\mathtt{T}_1 * \mathtt{T}_2 * \ldots * \mathtt{T}_n \texttt{ -> } \mathtt{T})}(\mathtt{E}, k)(M_{\mathtt{T}_1}(\mathtt{E}_1, k), M_{\mathtt{T}_2}(\mathtt{E}_2, k), \ldots M_{\mathtt{T}_n}(\mathtt{E}_n, k))_{\mathtt{V}}$ for all time steps k and types `T`. (The analogue for array accessors naturally also holds and follows from Semantic item 3 of Section 9.)

[18] This is only possible for array and function accessors.

To clarify what this entails, then as an example, assume for simplicity that $\texttt{n} = 1$ and the type of $\texttt{E}_1$ is comprised of the values in the set $\{\texttt{V}_1, \texttt{V}_2, \ldots \texttt{V}_\texttt{n}\}$. Then the expression $\texttt{E}(\texttt{E}_1)$ is equivalent to[19]:

```
if E1=V1 then E(V1) else
if E1=V2 then E(V2) else
    ⋮
if E1=Vn−1 then E(Vn−1) else E(Vn)
```

4. (ProjExprNil) A projection expression `E A` where `E` is **nil** in time step n is also **nil** in time step n.

Static Flag

1. (ProjExprStaticFlag) $\mathcal{SF}$(`<proj_expr>`) = `0`.

Restrictions

1. (ProjAccCompatible) The accessor `A` of an expression `E A` must be compatible with the type `T` of the expression `E`, according to the following table:

`A`	**Compatible type** `T`
`.K` (an `<int_literal>`)	Tuple with `> K` components
`.M` (an `<id>`)	Struct with a component named `M`
$[\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_\texttt{n}]$	Array with `n` dimensions
$(\texttt{E}_1, \texttt{E}_2, \ldots \texttt{E}_\texttt{n})$	Function $(\texttt{T}_1 * \texttt{T}_2 * \ldots * \texttt{T}_\texttt{n}$ `-> T`) where the type of $\texttt{E}_\texttt{i}$ is assignable to $\texttt{T}_\texttt{i}$

[19] To concretise the example, assume that `E` is defined with `E(i) := ~i & X(i);`, then we can use this equivalence to see that `E(input)` = (`if input = true then E(true) else E(false)`) = `false`, and not `~input & X(input)`.

10.7 Closed Expressions

Closed expressions are a purely syntactic concept, and consist of the explicitly grouped expressions.

Syntax

(ClosedExprSyntax)

```
<closed_expr> ::= <bool_literal>
                | <int_literal>
                | <named_expr>
                | <next_expr>
                | <pre_expr>
                | <fun_expr>
                | <cast_expr>
                | <with_expr>
                | <case_expr>
                | <quantif_expr>
                | "(" <expr> ")"
```

Semantics

1. (GroupedExpr) $M((\texttt{E}), n) = M(\texttt{E}, n)$ for any model M and time step n. `(E)` has the same type as `E`. Explicit grouping can be used to override the implicit grouping performed by the parsing of an `<expr>`. For example `(a + b) * c` is not equivalent to `a + b * c` since operator `*` has a higher precedence than operator `+`.

Static Flag

1. (GroupedExprStaticFlag) $\mathcal{SF}(\texttt{(E)}) = \mathcal{SF}(\texttt{E})$.

Restrictions

(empty)

10.7.1 Boolean Literals

Syntax

(BoolLitSyntax)

```
<bool_literal> ::= <true> | <false>
<true>         ::= "true"  | "TRUE"  | "True"
<false>        ::= "false" | "FALSE" | "False"
```

Semantics

1. (BoolLitTrue) `<true>` is a stream of type `bool` such that $M_{\text{bool}}(\texttt{<true>}, n) = \text{true}$ for all time steps n.
2. (BoolLitFalse) `<false>` is a stream of type `bool` such that $M_{\text{bool}}(\texttt{<false>}, n) = \text{false}$ for all time steps n.

Static Flag

1. (BoolLitStaticFlag) $\mathcal{SF}(\texttt{<bool\_literal>}) = 2$.

Restrictions

(empty)

10.7.2 Integer Literals

Syntax

(IntLitSyntax)

```
<dec_literal> ::= [0-9](_?[0-9])*
<bin_literal> ::= 0[Bb][0-1][_?[0-1])*
<hex_literal> ::= 0[Xx][0-9A-Fa-f](_?[0-9A-Fa-f])*
<int_literal> ::= <dec_literal>
                | <hex_literal>
                | <bin_literal>
```

Semantics

1. (IntLitUnderscores) Underscores (`_`) may be put freely inside integer literals with the purpose of improving readability (for example `1_000_000` or `0xFFFF_FFFF`). They can be removed completely without changing the meaning of the literals they appear in, and will not be considered in the following.
2. (IntLitDecimal) A `<dec_literal>` shall be intepreted as an integer in base 10 in the standard way (most significant digit first and least significant digit last).
3. (IntLitBinary) A `<bin_literal>` starts with the prefix `0B` or `0b` and is followed by the significant bits. The significant bits shall be interpreted as an unsigned integer in base 2 in the standard way (MSB first and LSB last).
4. (IntLitHexadecimal) A `<hex_literal>` starts with the prefix `0X` or `0x` and is followed by the significant digits. The significant digits shall be interpreted as an unsigned integer in base 16 in the standard way (most significant digit first and least significant digit last).
5. (IntLiteral) An integer literal `<int_literal>` with the interpreted value `K`, as specified above, is a stream of type `int` such that $M_{\text{int}}(\texttt{<int\_literal>}, n) = \texttt{K}$ for all time steps n.

Static Flag

1. (IntLitStaticFlag) $\mathcal{SF}(\texttt{<int\_literal>}) = \texttt{2}$.

Restrictions

(empty)

10.7.3 Named Expressions

Named expressions are references to stream variables.

Syntax

(NamedExprSyntax)

```
<named_expr> ::= <path_id>
```

Semantics

1. (NamedExpr) If the `<path_id>` of a `<named_expr>` refers to an existing stream (in the namespace of streams), then the `<named_expr>` is that stream.
2. (NamedExprImplicitDecl) Otherwise, the `<named_expr>` (which must be unqualified as according to (PathIdNoImplicitDecl)) refers to a unique implicit input stream of type `bool`. The input stream is implicitly declared by the `<named_expr>`. The declaration is made in the top-level scope (of the namespace of streams) of the user namespace in which the `<named_expr>` occurs, or else on the global top-level (if the `<named_expr>` does not occur inside a user namespace).

Static Flag

1. (NamedExprStaticFlag) The static flag of a `<named_expr>` is equal to the static flag of the stream it refers to.
2. (NamedExprUndefinedStaticFlag) The static flag of an undefined stream variable is 0.

Restrictions

(empty)

10.7.4 Next Expressions

Syntax

(NextExprSyntax)

```
<next_expr> ::= "X" "(" <expr> ")"
```

Semantics

1. (NextExpr) $M(\texttt{X(E)}, n) = M(\texttt{E}, n+1)$ for any model M and time step n. `X(E)` has the same type as `E`.

Static Flag

1. (NextExprStaticFlag) $\mathcal{SF}(\texttt{<next\_expr>}) = \texttt{0}$.

Restrictions

(empty)

10.7.5 Pre Expressions

Syntax

(PreExprSyntax)

```
<pre_expr> ::= ("pre" | "PRE") ["<" <type> ">"]
               "(" <expr> ["," <expr>] ")"
```

Semantics

1. (PreUppercase) `PRE` is equivalent to `pre`.
2. (PreTypedWithInit) `pre<T>`$(\texttt{E}_1, \texttt{E}_2)$ is an initialized memory stream of type `T`, where $\texttt{E}_2$ is the initial value for the initial time step and $\texttt{E}_1$ is the memorized value for all other time steps. Formally, if $\texttt{T}_1$ and $\texttt{T}_2$ correspond respectively to the types of the expressions $\texttt{E}_1$ and $\texttt{E}_2$, then the value of the expression at time step k is defined as follows:

$$M_{\texttt{T}}(\texttt{pre<T>}(\texttt{E}_1, \texttt{E}_2), k) = \begin{cases} M_{\texttt{T}_2}(\texttt{E}_2, 0) & \text{if value} \in \texttt{T} \\ \textbf{nil} & \text{otherwise} \end{cases} \quad k = 0$$

$$\qquad\qquad\qquad\qquad\qquad\quad \begin{cases} M_{\texttt{T}_1}(\texttt{E}_1, k-1) & \text{if value} \in \texttt{T} \\ \textbf{nil} & \text{otherwise} \end{cases} \quad k > 0$$

3. (PreTyped) `pre<T>(E)` is an uninitialized memory stream of type `T` which takes the value **nil** in the initial time step. Formally, if $\texttt{T}_1$ corresponds to the type of the expression `E`, then the value of the expression at time step k is defined as follows:

$$M_{\texttt{T}}(\texttt{pre<T>}(\texttt{E}), k) = \begin{cases} \textbf{nil} & k = 0 \\ M_{\texttt{T}_1}(\texttt{E}, k-1) & \text{if value} \in \texttt{T} \\ \textbf{nil} & \text{otherwise} \end{cases} \quad k > 0$$

4. (PreUntyped) `pre(E)` is equivalent to `pre<T>(E)` where `T` is the unsized copy (see 8.4.2) of the type of `E`.
5. (PreUntypedWithInit) `pre`$(\texttt{E}_1,\ \texttt{E}_2)$ is equivalent to `pre<T>`$(\texttt{E}_1,\ \texttt{E}_2)$ where `T` is the union type (see 8.4.3) of the types of $\texttt{E}_1$ and $\texttt{E}_2$.

Static Flag

1. (PreExprStaticFlag) $\mathcal{SF}$(`<pre_expr>`) = `0`.

Restrictions

1. (PreOperandsAssignable) The types of the operands $\texttt{E}_1$ and $\texttt{E}_2$ of `pre<T>`$(\texttt{E}_1, \texttt{E}_2)$ and the type of the operand `E` of `pre<T>(E)`, shall be assignable to `T`.

10.7.6 Function-Style Expressions

Syntax

(FunopSyntax)

```
<fun_expr> ::= <fop> "(" <expr_list> ")"
<fop>      ::= "$min"
             | "$max"
             | "$abs"
             | "$or"
             | "$and"
             | "$xor"
             | "$not"
             | "bin2u"
             | "u2bin"
             | "bin2s"
             | "s2bin"
             | "population_count_lt"
             | "population_count_gt"
             | "population_count_eq"
```

Semantics

1. (IntMin) `$min`(E_1, E_2) (minimum) is equivalent to:
   (`if` E_1 `<` E_2 `then` E_1 `else` E_2).

2. (IntMax) `$max`(E_1, E_2) (maximum) is equivalent to:
   (`if` E_1 `>` E_2 `then` E_1 `else` E_2).

3. (IntAbs) `$abs`(E) (absolute value) is equivalent to:
   (`if E < 0 then -E else E`).

4. (OpBin2u) `bin2u`(E_1, E_2) interprets the E_2 first bits of an array E_1 of `bool` as an unsigned binary number and is equivalent to:
   `SUM i :` [0, E_2`-1`] (`2 ^ i *` (`if` E_1[`i`] `then 1 else 0`)) for a fresh `i`.

5. (OpBin2s) `bin2s`(E_1, E_2) interprets the E_2 first bits of an array E_1 of `bool` as a signed binary number encoded in the two's complement notation and is equivalent to:
   `-(if` E_1[E_2`-1`] `then 1 else 0) * 2 ^ (`E_2`-1) + bin2u(`E_1, E_2`-1)`.

6. (OpU2bin) `u2bin`(E_1, E_2) converts an integer E_1 into an array of type `bool^(`E_2`)` and is equivalent to:
   (`lambda`[E_2] : [`i`] `:=` `bit_is_one`(E_1, `i`))
   for a fresh `i`, together with the following declaration and definition (we do not use bitwise and (operator `$and`) in the definition of `bit_is_one` since that operator is defined in terms of `u2bin`):

   ```
   Declarations:
   ```

```
 bool bit_is_one(int, int);
Definitions:
 bit_is_one(E, i) := ((E >> i) - ((E >> (i + 1)) << 1)) == 1;
```

7. (OpS2bin) `s2bin`(E_1, E_2) is equivalent to `u2bin`(E_1, E_2).

8. (OpPopCountLt) `population_count_lt`(E_1, $E_2, \ldots E_n$, K) of `n` Boolean streams E_i and an integer stream `K` is true exactly when less than `K` of the Boolean streams are true. The expression is reducible to:

   ((`if` E_1 `then 1 else 0`) +
   (`if` E_2 `then 1 else 0`) +
   $\vdots$
   (`if` E_n `then 1 else 0`)) < K

9. (OpPopCountGt) `population_count_gt`(E_1, $E_2, \ldots E_n$, K) is equivalent to `~population_count_lt`(E_1, $E_2, \ldots E_n$, K + 1).

10. (OpPopCountEq) `population_count_eq`(E_1, $E_2, \ldots E_n$, K) is equivalent to `~population_count_lt`(E_1, $E_2, \ldots E_n$, K) &
    `~population_count_gt`(E_1, $E_2, \ldots E_n$, K)

11. (OpBitwiseOr) `$or`(E_1, E_2) (bitwise or) is **nil** in time step k if either of E_1 or E_2 is **nil** in time step k. *Otherwise* the expression is equivalent to `bin2s`((`lambda`[C] : [i] := `s2bin`(E_1, C)[i] # `s2bin`(E_2, C)[i]), C)
    for a fresh `i` and a constant `C`. `C` must be big enough to represent all significant bits of E_1 and E_2, *i.e.*

    $$C \geq \log_2(\max(\text{ub}(E_1) + 1, \text{ub}(E_2) + 1, |\text{lb}(E_1)|, |\text{lb}(E_2)|)) + 1$$

    where ub(E) and lb(E) denote, respectively, the maximum and minimum values the integer expression E may take in any model.

12. (OpBitwiseXor) `$xor`(E_1, E_2) (bitwise xor) is equivalent to `bin2s`((`lambda`[C] : [i] := `s2bin`(E_1, C)[i] #! `s2bin`(E_2, C)[i]), C)
    for a fresh `i` and a constant `C`. `C` must be big enough to represent all significant bits of E_1 and E_2, *i.e.*

    $$C \geq \log_2(\max(\text{ub}(E_1) + 1, \text{ub}(E_2) + 1, |\text{lb}(E_1)|, |\text{lb}(E_2)|)) + 1$$

    where ub(E) and lb(E) denote, respectively, the maximum and minimum values the integer expression E may take in any model.

13. (OpBitwiseNot) `$not`(E) (bitwise not) is equivalent to `bin2s`((`lambda`[C] : [i] := `~s2bin`(E, C)[i]), C)
    for a fresh `i` and a constant `C`. `C` must be big enough to represent all significant bits of E, *i.e.*

    $$C \geq \log_2(\max(\text{ub}(E) + 1, |\text{lb}(E)|)) + 1$$

    where ub(E) and lb(E) denote, respectively, the maximum and minimum values the integer expression E may take in any model.

14. (OpBitwiseAnd) `$and`(E_1, E_2) (bitwise and) is equivalent to `$not`(`$or`(`$not`(E_1), `$not`(E_2))).

Static Flag

1. (FunopUnaryStaticFlag) $\mathcal{SF}(\circ(E_1)) = \mathcal{SF}(E_1)$ for $\circ \in$ {`$abs`, `$not`}.
2. (FunopBinaryStaticFlag) $\mathcal{SF}(\circ(E_1, E_2)) = \min(\mathcal{SF}(E_1), \mathcal{SF}(E_2))$ for $\circ \in$ {`$min`, `$max`, `$and`, `$xor`, `$or`}.
3. (FunopNaryStaticFlag) $\mathcal{SF}(\circ(E_1, \ldots E_n)) = 0$ for $\circ \in$ {`bin2u`, `bin2s`, `u2bin`, `s2bin`, `population_count_lt`, `population_count_gt`, `population_count_eq`}.

Restrictions

1. (FunopUnaryCard) The cardinality of the `<expr_list>` shall be 1 for the following operators: `$abs, $not`.
2. (FunopBinaryCard) The cardinality of the `<expr_list>` shall be 2 for the following operators:
`$min, $max, bin2u, bin2s, u2bin, s2bin, $and, $xor, $or`.
3. (PopCountNumberStatic) $\mathcal{SF}(K) \geq 1$ for an expression $\circ(E_1, \ldots E_n, K)$ with $\circ \in$ {`population_count_lt`, `population_count_gt`, `population_count_eq`}.

10.7.7 Cast Expressions

Syntax

(CastExprSyntax)

```
<cast_expr> ::= "cast" "<" <type> ">" "(" <expr> ")"
```

Semantics

1. (CastExpr) A *Cast* expression `cast<T>(E)`, of type `int`, is an (unchecked) conversion of an integer expression `E` into the target type `T`.
2. (CastSigned) `cast<int signed C>(E)` is equivalent to:
   `bin2s(s2bin(E, C), C)`.
3. (CastUnsigned) `cast<int unsigned C>(E)` is equivalent to:
   `bin2u(s2bin(E, C), C)`.

Static Flag

1. (CastStaticFlag) $\mathcal{SF}$(`<cast_expr>`) = `0`.

Restrictions

1. (CastTargetIntImpl) The target type of a `cast` shall be an integer implementation type, or a named type that is defined, directly or indirectly, as an integer implementation type.

10.7.8 With Expressions

With expressions can be understood as an operation that creates a modified copy of a composite stream, where a single component has been arbitrarily modified. For example "`(A with [0] := ~A[0])`" is equivalent to the Boolean array `A` for all components except the one at index 0, which has been negated. With expressions are particularly useful when translating assignments of an imperative language into HLL.

With expressions can be written with collections on the right hand side (see the definition of `<rhs>` in Section 12). In order to explain the semantics of with expressions with collections on the right hand side, we expand them into a chain of with expressions, each with a single element of the collection on the right hand side (this expansion would have to be repeated if such an element is in turn another collection). For example: "`(A with [0] := {1, 2, 3})`" is expanded, assuming the projection expression `A[0]` is of array type (*i.e.* `A` is an array of arrays), into the equivalent formula "`(((A with [0][0] := 1) with [0][1] := 2) with [0][2] := 3)`". This expansion is formalised in (WithCollectionRhsUniDim) below.

The multidimensional case, *e.g.* when `A[0]` is a multidimensional array (or multivariate function), is a bit trickier and formalised in (WithCollectionRhsMultiDim) below. However, the basic idea is the same.

Syntax

(WithExprSyntax)

```
<with_expr> ::= "(" <expr> "with" {<accessor>}+ ":=" <rhs> ")"
```

Forward References

1. `<rhs>` is defined in Section 12.

Semantics

1. (WithCollectionRhsUniDim) $(\texttt{E with A :=} \{\texttt{R}_0, \texttt{R}_1, \ldots \texttt{R}_{n-1}\})$ where `E A` is of type $\texttt{T}_{\texttt{E}_\texttt{A}}$, which is neither a multidimensional array type nor a multivariate function type, is equivalent to:

   $(\ldots((\texttt{E with A A}_0 \texttt{ := R}_0) \texttt{ with A A}_1 \texttt{ := R}_1) \ldots \texttt{ with A A}_{n-1} \texttt{ := R}_{n-1})$ where:

   $$\texttt{A}_\texttt{i} = \begin{cases} \texttt{.i} & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } \texttt{tuple } \{\texttt{T}_0, \ldots \texttt{T}_{n-1}\} \\ \texttt{.M}_\texttt{i} & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } \texttt{struct } \{\texttt{M}_0 : \texttt{T}_0, \ldots \texttt{M}_{n-1} : \texttt{T}_{n-1}\} \\ \texttt{[i]} & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } \texttt{T\^{}(n)} \\ \texttt{(V}_\texttt{i}\texttt{)} & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } (\texttt{T}_1 \texttt{ -> T}) \text{ and } \texttt{T}_1 \text{ is } \{\texttt{V}_0, \ldots \texttt{V}_{n-1}\} \text{ with } \texttt{V}_\texttt{i} < \texttt{V}_{\texttt{i}+1} \end{cases}$$

2. (WithCollectionRhsMultiDim) $(\texttt{E with A :=} \{\texttt{R}_0, \texttt{R}_1, \ldots \texttt{R}_{m_1}\})$ where `E A` is of type $\texttt{T}_{\texttt{E}_\texttt{A}}$, which is either a multidimensional array type or a multivariate function type, is equivalent to:

   $$\begin{aligned} (\ldots(\ldots((&\texttt{E with A A}_{0\ldots00} \texttt{ := R}_{0\ldots00}) \texttt{ ... with A A}_{0\ldots0m_k} \texttt{ := R}_{0\ldots0m_k}) \\ &\texttt{with A A}_{0\ldots10} \texttt{ := R}_{0\ldots10}) \texttt{ ... with A A}_{0\ldots1m_k} \texttt{ := R}_{0\ldots1m_k}) \\ &\qquad\vdots \qquad\quad \vdots \qquad \ddots \qquad \vdots \qquad\quad \vdots \\ &\texttt{with A A}_{0\ldots m_{k-1}0} \texttt{ := R}_{0\ldots m_{k-1}0}) \texttt{ ... with A A}_{0\ldots m_{k-1}m_k} \texttt{ := R}_{0\ldots m_{k-1}m_k}) \\ &\qquad\vdots \qquad\quad \vdots \qquad \ddots \qquad \vdots \qquad\quad \vdots \\ &\texttt{with A A}_{m_1m_2\ldots0} \texttt{ := R}_{m_1m_2\ldots0}) \texttt{ ... with A A}_{m_1m_2\ldots m_k} \texttt{ := R}_{m_1m_2\ldots m_k}) \end{aligned}$$

   where:

   $\texttt{A}_{\texttt{i}_1\ldots\texttt{i}_k} =$

   $$= \begin{cases} [\texttt{i}_1, \ldots \texttt{i}_k] & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } \texttt{T\^{}}(\texttt{m}_1 \texttt{ + 1}, \ldots \texttt{m}_k \texttt{ + 1}) \\ (\texttt{V}_{\texttt{i}_1}, \ldots \texttt{V}_{\texttt{i}_k}) & \text{if } \texttt{T}_{\texttt{E}_\texttt{A}} \text{ is } (\texttt{T}_1 * \ldots \texttt{T}_k \texttt{ -> T}), \texttt{T}_{\texttt{j} \in [1,k]} \text{ is } \{\texttt{V}_0, \ldots \texttt{V}_{\texttt{m}_\texttt{j}}\}, \texttt{V}_\texttt{i} < \texttt{V}_{\texttt{i}+1} \end{cases}$$

   and $\texttt{R}_{\texttt{i}_1\ldots\texttt{i}_k}$ are constructed inductively:

   (base) $\texttt{R}_{\texttt{i}_1}$ for $\texttt{i}_1 \in [\texttt{0}, \texttt{m}_1]$ are given above (as the elements of the collection).

   (step) $\texttt{R}_{\texttt{i}_1\ldots\texttt{i}_j} =$

   $$= \begin{cases} \texttt{R}'_{\texttt{i}_\texttt{j}} & \text{if } \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_{j-1}} \text{ is } \{\texttt{R}'_0, \ldots \texttt{R}'_{\texttt{m}_\texttt{j}}\} \\ \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_p}[\texttt{i}_{p+1}, \ldots \texttt{i}_k] & \text{if } \texttt{j} = \texttt{k} \text{ and } \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_{p-1}} \text{ is } \texttt{<collection>} \text{ and } \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_p} \text{ is } \texttt{<expr>} \text{ of type } \texttt{T\^{}}(\texttt{m}_{p+1} \texttt{ + 1}, \ldots \texttt{m}_k \texttt{ + 1}) \\ \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_p}(\texttt{V}_{\texttt{i}_{p+1}}, \ldots \texttt{V}_{\texttt{i}_k}) & \text{if } \texttt{j} = \texttt{k} \text{ and } \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_{p-1}} \text{ is } \texttt{<collection>} \text{ and } \texttt{R}_{\texttt{i}_1\ldots\texttt{i}_p} \text{ is } \texttt{<expr>} \text{ of type } (\texttt{T}_{p+1} * \ldots \texttt{T}_k \texttt{ -> T}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

3. (WithMultipleAcc) $(\texttt{E}_1\ \texttt{with}\ \texttt{A}_1\ \texttt{A}_2 \ldots \texttt{A}_n\ \texttt{:=}\ \texttt{E}_2)$ is equivalent to $(\texttt{E}_1\ \texttt{with}\ \texttt{A}_1\ \texttt{A}_2 \ldots \texttt{A}_{n-1}\ \texttt{:=}\ (\texttt{E}_1\ \texttt{A}_1\ \texttt{A}_2 \ldots \texttt{A}_{n-1}\ \texttt{with}\ \texttt{A}_n\ \texttt{:=}\ \texttt{E}_2))$.

4. (WithArrayAcc) $(\texttt{E}_1\ \texttt{with}\ [\texttt{E}_3,\ \texttt{E}_4, \ldots \texttt{E}_n]\ \texttt{:=}\ \texttt{E}_2)$ where $\texttt{E}_1$ is of type $\texttt{T}\hat{}(\texttt{C}_3,\ \texttt{C}_4, \ldots \texttt{C}_n)$ is equivalent to:
$(\texttt{lambda}[\texttt{C}_3,\ \texttt{C}_4, \ldots \texttt{C}_n] :\ [\texttt{i}_3,\ \texttt{i}_4, \ldots \texttt{i}_n]\ \texttt{:=}$
$\quad \texttt{if}\ \texttt{i}_3\ \texttt{=}\ \texttt{E}_3\ \texttt{\&}\ \texttt{i}_4\ \texttt{=}\ \texttt{E}_4\ \texttt{\&} \ldots \texttt{\&}\ \texttt{i}_n\ \texttt{=}\ \texttt{E}_n\ \texttt{then}\ \texttt{E}_2\ \texttt{else}\ \texttt{E}_1[\texttt{i}_3,\ \texttt{i}_4, \ldots \texttt{i}_n])$
where $\texttt{i}_j$ with $3 \leq j \leq n$ are fresh variables.

5. (WithFunctionAcc) $(\texttt{E}_1\ \texttt{with}\ (\texttt{E}_3,\ \texttt{E}_4, \ldots \texttt{E}_n)\ \texttt{:=}\ \texttt{E}_2)$ where $\texttt{E}_1$ is of type $(\texttt{T}_3 * \texttt{T}_4 * \ldots * \texttt{T}_n\ \texttt{->}\ \texttt{T})$ is equivalent to:
$(\texttt{lambda}(\texttt{T}_3,\ \texttt{T}_4, \ldots \texttt{T}_n) :\ (\texttt{i}_3,\ \texttt{i}_4, \ldots \texttt{i}_n)\ \texttt{:=}$
$\quad \texttt{if}\ \texttt{i}_3\ \texttt{=}\ \texttt{E}_3\ \texttt{\&}\ \texttt{i}_4\ \texttt{=}\ \texttt{E}_4\ \texttt{\&} \ldots \texttt{\&}\ \texttt{i}_n\ \texttt{=}\ \texttt{E}_n\ \texttt{then}\ \texttt{E}_2\ \texttt{else}\ \texttt{E}_1(\texttt{i}_3,\ \texttt{i}_4, \ldots \texttt{i}_n))$
where $\texttt{i}_j$ with $3 \leq j \leq n$ are fresh variables.

6. (WithTupleStructAcc) $(\texttt{E}_1\ \texttt{with}\ \texttt{.M}\ \texttt{:=}\ \texttt{E}_2)$ where $\texttt{E}_1$ is of tuple or struct type T, and $\texttt{E}_2$ is of type $\texttt{T}_\texttt{M}$ is identical to $\texttt{E}_1$ for all components except the one designated by M, which is equal to $\texttt{E}_2$. A formal definition uses the following equation (with an overloading of operator @):

$$M_\texttt{T}((\texttt{E}_1\ \texttt{with}\ \texttt{.M:=}\ \texttt{E}_2), k)\texttt{@K} = \begin{cases} M_{\texttt{T}_\texttt{M}}(\texttt{E}_2, k) & \text{if } \texttt{K} = \texttt{M} \\ M_\texttt{T}(\texttt{E}_1, k)\texttt{@K} & \text{otherwise} \end{cases}$$

Static Flag

1. (WithExprStaticFlag) $\mathcal{SF}(\texttt{<with\_expr>}) = 0$.

Restrictions

1. (WithAccCompatible) The accessor A in the expression (E with A := R) must be a compatible accessor w.r.t. the type of E, meaning that the projection expression E A must be a valid expression.

2. (WithRhsAssignable) The type of the <rhs> R of an expression (E with A := R) shall be assignable to the type of the projection expression E A.

10.7.9 Case Expressions

Syntax

(CaseExprSyntax)

```
<case_expr>    ::= "(" <expr_list> {<case_item>}+ ")"
<case_item>    ::= "|" <pattern_list> "=>" <expr>
<pattern>      ::= <expr>
                 | <named_type> ( <id> | "_" )
                 | "_"
<pattern_list> ::= <pattern> { "," <pattern> }
```

Semantics

1. (CaseExpr) In each time step, a case expression

   $$\begin{array}{l}(\texttt{E}_1,\ \texttt{E}_2,\ \ldots\ \texttt{E}_n\\ |\texttt{P}_{11},\ \texttt{P}_{12},\ \ldots\ \texttt{P}_{1n}\ \texttt{=>}\ \texttt{R}_1\\ |\texttt{P}_{21},\ \texttt{P}_{22},\ \ldots\ \texttt{P}_{2n}\ \texttt{=>}\ \texttt{R}_2\\ \quad\vdots\qquad\ddots\qquad\vdots\\ |\texttt{P}_{m1},\ \texttt{P}_{m2},\ \ldots\ \texttt{P}_{mn}\ \texttt{=>}\ \texttt{R}_m)\end{array}$$

   evaluates to the first *branch* $\texttt{R}_\texttt{i}$ for which all the `<pattern>` items $\texttt{P}_{\texttt{i1}}$ to $\texttt{P}_{\texttt{in}}$ *match* the switch expressions $\texttt{E}_1$ to $\texttt{E}_\texttt{n}$. Each $\texttt{P}_{\texttt{ij}}$ is matched against, and only against, the corresponding $\texttt{E}_\texttt{j}$. The type of the expression is the union of the types of the $\texttt{R}_\texttt{i}$.

2. (CasePatternExpr) A `<pattern>` item $\texttt{E}_\texttt{p}$ which is an `<expr>` matches a switch expression $\texttt{E}_\texttt{s}$ in time step k iff $M_{\text{bool}}(\texttt{E}_\texttt{p} = \texttt{E}_\texttt{s}, k) = \text{true}$.

3. (CasePatternType) A `<pattern>` item `T x` which is a `<named_type>` (followed by either an identifier or a wildcard) matches a switch expression $\texttt{E}_\texttt{s}$ in time step k iff $M_{\text{bool}}(\texttt{E}_\texttt{s}\ :\ \texttt{T}, k) = \text{true}$.

4. (CasePatternWildcard) The `<pattern>` item `_` (a wildcard) matches any switch expression $\texttt{E}_\texttt{s}$ (in any time step).

5. (CaseCapturingVariable) Each `<case_item>` of a case expression opens a local scope in the namespace of streams, called a *Branch scope*, that starts at the `=>` and continues to the end of the `<case_item>`. Given a `<pattern>` $\texttt{P}_{\texttt{ij}}$ on the form `<named_type> <id>` of the `<case_item>`, the identifier `<id>` resides in the branch scope, and if the `<case_item>` is a match in time step k, the `<id>` refers to a local *static* stream `x` of type `T` (similar to a lambda parameter of Section 10.2), called a *Capturing variable*, whose value in each time step is equal to $M_\texttt{T}(\texttt{E}_\texttt{j}, k)$, *i.e.* $M_\texttt{T}(\texttt{x}, n) = M_{\texttt{T}_{\texttt{E}_\texttt{j}}}(\texttt{E}_\texttt{j}, k)$ for *all* time steps n. In other words, the capturing variable `x` refers to a different static stream in each matching time step, and applying a temporal operator such as `X` or `PRE` to such a variable has thus no effect.

6. (CaseExprNil) A case expression propagates **nil** in a similar manner to an equivalent sequence of if-then-else expressions (see Section 10.1). Furthermore, a case expression evaluates to **nil** in a time step k if there is no `<pattern_list>` matching the switch expressions in that time step (the case expression is not exhaustive). In order to formalize this, we will first define the binary function "match", with signature match : `<expr_list>` $\times$ `<pattern_list>` $\rightarrow$ `<expr>`, as follows:

$$\text{match}(\{\texttt{E}_1, \ldots \texttt{E}_n\}, \{\texttt{P}_1, \ldots \texttt{P}_n\}) =$$
$$= \begin{cases} \text{match}(\{\texttt{E}_1\}, \{\texttt{P}_1\})\ \texttt{\&} \ldots \texttt{\&} & \text{if } \texttt{n} > 1, \text{ else} \\ \text{match}(\{\texttt{E}_n\}, \{\texttt{P}_n\}) & \\ \texttt{E}_1\ \texttt{=}\ \texttt{P}_1 & \text{if } \texttt{P}_1 \text{ is on the form } \texttt{<expr>}, \text{ else} \\ \texttt{E}_1\ \texttt{:}\ \texttt{T} & \text{if } \texttt{P}_1 \text{ is on the form } \texttt{T x} \text{ where } \texttt{T} \\ & \text{is a } \texttt{<named\_type>}, \text{ else} \\ \texttt{true} & (\texttt{P}_1 \text{ is on the form } \texttt{"\_"}) \end{cases}$$

Now, given the case expression of (CaseExpr) above, it will evaluate to **nil** in time step k if one of the following conditions is true:

(a) There is no matching `<pattern_list>` $\texttt{P}_{i1}, \ldots \texttt{P}_{in}$, *i.e.* one for which $M_{\text{bool}}(\text{match}(\{\texttt{E}_1, \ldots \texttt{E}_n\}, \{\texttt{P}_{i1} \ldots \texttt{P}_{in}\}), k) = \text{true}$, or

(b) There is a `<pattern_list>` $\texttt{P}_{j1}, \ldots \texttt{P}_{jn}$ above the first matching `<pattern_list>` $\texttt{P}_{i1}, \ldots \texttt{P}_{in}$ (*i.e.* with $\texttt{j} < \texttt{i}$) such that $M_{\text{bool}}(\text{match}(\{\texttt{E}_1, \ldots \texttt{E}_n\}, \{\texttt{P}_{j1} \ldots \texttt{P}_{jn}\}), k) = \textbf{nil}$.

(Note that the case expression will of course also evaluate to **nil** in time step k in case the matching branch $\texttt{R}_i$ evaluates to **nil** in time step k.)

Static Flag

1. (CaseExprStaticFlag) $\mathcal{SF}(\texttt{<case\_expr>}) = \texttt{0}$.
2. (CaseCapturingVarStaticFlag) $\mathcal{SF}(\texttt{x}) = \texttt{1}$ for a capturing variable `x` corresponding to the `<id>` of a `<pattern>` on the form `<named_type> <id>`.

Restrictions

1. (CaseSwitchesScalar) The switches in the `<expr_list>` of a case expression shall be of scalar type.
2. (CasePatternsCompatible) The number of pattern items (`<pattern>`) in each `<case_item>` shall match the number of switches in the `<expr_list>`, and their types shall be pair-wise compatible, except for wildcards (`_`).
3. (CaseBranchesCompatible) The types of the branches ($\texttt{R}_i$) shall be compatible.
4. (CasePatternExprConstant) $\mathcal{SF}(\texttt{E}) = \texttt{2}$ for a `<pattern> E` which is an `<expr>`.
5. (CasePatternTypeSort) A pattern item `<pattern>` which is a `<named_type>` shall refer to a valid sort type.
6. (CaseCapturingVarUnicity) Two capturing variables of the same branch may not have the same name.

10.7.10 Quantifier Expressions

Quantifiers in HLL usually operate on static domains (sets of values – much like types) to create properties over populations of streams. For example, if we have an array `A` of 3 integers, declared as `int A[3]`, with the components taking the following values:

```
A[0]     :    0,     1,     2,     3, ...
A[1]     :    0,     0,   nil,     1, ...
A[2]     :    0,     4,     4,     3, ...
```

Then the following HLL quantifiers over `A` take the following values:

```
ALL    i:[0,2] (A[i] < 4): true, false, false,  true, ...
SOME   i:[0,2] (A[i] = 0): true,  true,   nil, false, ...
SUM    i:[0,2] (A[i])    :    0,     5,   nil,     7, ...
PROD   i:[0,2] (A[i])    :    0,     0,   nil,     9, ...
$min   i:[0,2] (A[i])    :    0,     0,   nil,     1, ...
$max   i:[0,2] (A[i])    :    0,     4,   nil,     3, ...
```

The `SELECT` operator selects the unique value from the domain that satisfies the given predicate. If zero or two (or more) values from the domain satisfy the predicate, then **nil** is selected instead:

```
SELECT i:[0,2] (A[i] = 1):  nil,     0,   nil,     1, ...
```

Quantification is extended to several domains in the natural way. For example:

```
ALL i:[0,2], j:[0,2] (i=j # A[i] != A[j]): false, true, nil, false, ...
```

Quantifiers can also be nested, and the domains of nested quantifiers may depend on the quantifier variables of the enclosing ones. So the previous formulation could be optimized to reduce the number of iterations:

```
ALL i:[0,2] ALL j:[0, i-1] (A[i] != A[j]): false, true, nil, false, ...
```

Quantification was extended in HLL version 3.0 to allow quantification over (the components of) array and function streams, using the new operator `$items`. As an example, `ALL a:$items(A) (a < 4)` is equivalent to `ALL i:[0,2] (A[i] < 4)` above. As two components of an array or function may be equal, this corresponds to quantification over a domain which is a **multiset of streams (of values)**. To fit easier with the historical quantifiers in the formal definition below, we will use an alternative (but equivalent) point of view and instead define the domain as a **stream of multisets (of values)**. To connect this with our running example, it means that the domain `$items(A)` corresponds to the following stream of multisets:

```
$items(A): {0, 0, 0}, {1, 0, 4}, {2, nil, 4}, {3, 1, 3}, ...
```

The use of multisets instead of sets is important only when considering the `SUM` and `PROD` quantifers. For example:

```
SUM a:$items(A) (1)      :    3,     3,     3,     3, ...
```

Note that the `SELECT` operator is not allowed in the combination with a domain on the form `$items(A)`.

Syntax

(QuantExprSyntax)

```
<quantif_expr>   ::= <quantifier> <quantif_vars>
                       ( "(" <expr> ")" | <quantif_expr> )
                   | "SELECT" <quantif_vars>
                       ( "(" <expr> ["," <rhs>] ")" |
                         <quantif_expr> )

<quantif_vars>   ::= <quantif_var> {"," <quantif_var>}
<quantif_var>    ::= <id> ":" <quantif_domain>
<quantif_domain> ::= <domain>
                   | "$items" "(" <expr> ")"

<quantifier>     ::= "SOME" | "ALL"  | "SUM"  | "PROD"
                   | "CONJ" | "DISJ" | "$min" | "$max"
```

Semantics

1. (QuantScope) A *Quantifier expression* (`<quantif_expr>`) introduces a local scope in the namespace of streams, called a *Quantifier scope* that starts *right after* the last `<quantif_var>` and continues to the end of the expression. The quantifier variables (`<quantif_var>`) exist, and only exist, within this scope.[20]

2. (QuantDomainDomain) A quantifier domain `D` which is a `<domain>` corresponds for each time step k to the set of values $\mathtt{D}_k$ (see Section 10.4). Due to restriction (QuantDomainStatic), such a quantifier domain is guaranteed to be static and not to change from one time step to another.

[20] This means that upper bound `i` of the domain of `j` in the expression `ALL i : [0, 10], j : [0, i] (i >= j)` does *not* refer to the quantifier variable `i` introduced just before `j`. On the other hand, in the expression `ALL i : [0, 10] ALL j : [0, i] (i >= j)`, the upper bound of `j` does refer to the quantifier variable `i` in the enclosing quantifier expression.

3. (QuantDomainItems) A quantifer domain `$items(E)` where `E` is of array or function type $\mathtt{T_E}$ with component type `T` corresponds for each time step k to the following **multiset** of values:

$$\begin{array}{llll}
\{M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots00}}, k), & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots01}}, k), & \ldots & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots0m_n}}, k), \\
M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots10}}, k), & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots11}}, k), & \ldots & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots1m_n}}, k), \\
\vdots & \vdots & \ddots & \vdots \\
M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots m_{n-1}0}}, k), & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots m_{n-1}1}}, k), & \ldots & M_{\mathtt{T}}(\mathtt{E\ A_{0\ldots m_{n-1}m_n}}, k), \\
\vdots & \vdots & \ddots & \vdots \\
M_{\mathtt{T}}(\mathtt{E\ A_{m_1m_2\ldots0}}, k), & M_{\mathtt{T}}(\mathtt{E\ A_{m_1m_2\ldots1}}, k), & \ldots & M_{\mathtt{T}}(\mathtt{E\ A_{m_1m_2\ldots m_n}}, k)\ \}
\end{array}$$

where:

$$\mathtt{A_{i_1\ldots i_n}} = \\ = \begin{cases} [\mathtt{i_1}, \ldots \mathtt{i_n}] & \text{if } \mathtt{T_E} \text{ is } \mathtt{T\hat{}(m_1\ +\ 1, \ldots m_n\ +\ 1)} \\ (\mathtt{V_{i_1}}, \ldots \mathtt{V_{i_n}}) & \text{if } \mathtt{T_E} \text{ is } \mathtt{(T_1 * \ldots T_n\ ->\ T)}, \mathtt{T_{j \in [1,n]}} \text{ is } \{\mathtt{V_0}, \ldots \mathtt{V_{m_j}}\} \end{cases}$$

4. (QuantVarType) The type of a quantifer variable `i : D` is:
   (a) The type compatible with `D` if `D` is a `<domain>`, as according to (DomainAsType) and (DomainAsRange).
   (b) The component type of the array or function type of `E`, if `D` is on the form `$items(E)`.

5. (QuantMultVar) $\mathtt{QTF\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E)}$, where `QTF` is a `<quantifier>` other than `SELECT` and the $\mathtt{i_i : D_i}$ are `<quantif_var>`, is reducible to $\mathtt{QTF\ j_1 : D_1}$ $(\mathtt{QTF\ j_2 : D_2}$ $\ldots$ $(\mathtt{QTF\ j_n : D_n}$ $(\mathtt{E\langle i_1 := j_1, i_2 := j_2, \ldots i_n := j_n\rangle}))$ $\ldots)$ where $\mathtt{j_1}$ to $\mathtt{j_n}$ are fresh identifiers.

6. (QuantDisj) `DISJ` is equivalent to `SOME`.

7. (QuantConj) `CONJ` is equivalent to `ALL`.

8. (QuantSome) `SOME i:D (E)`, of type `bool`, evaluates to true in time step k iff there exists some value `j` in `D` such that `E`, with all occurrences of `i` replaced by `j`, evaluates to true. Formally:

$$M_{\text{bool}}(\mathtt{SOME\ i : D\ (E)}, k) = \begin{cases} \text{true} & \text{if } \exists \mathtt{j} \in \mathtt{D}_k\ M_{\text{bool}}(\mathtt{E\langle i := j\rangle}, k) = \text{true} \\ \textbf{nil} & \text{if } \exists \mathtt{j} \in \mathtt{D}_k\ M_{\text{bool}}(\mathtt{E\langle i := j\rangle}, k) = \textbf{nil} \\ \text{false} & \text{otherwise} \end{cases}$$

   (Where the cases on the right hand side should be considered in order from top to bottom.)

9. (QuantAll) `ALL i:D (E)`, of type `bool`, evaluates to true in time step k iff for all values `j` in `D`, `E`, with all occurrences of `i` replaced by `j`, evaluates to true. Formally:

$$M_{\text{bool}}(\mathtt{ALL\ i : D\ (E)}, k) = \begin{cases} \text{false} & \text{if } \exists \mathtt{j} \in \mathtt{D}_k\ M_{\text{bool}}(\mathtt{E\langle i := j\rangle}, k) = \text{false} \\ \textbf{nil} & \text{if } \exists \mathtt{j} \in \mathtt{D}_k\ M_{\text{bool}}(\mathtt{E\langle i := j\rangle}, k) = \textbf{nil} \\ \text{true} & \text{otherwise} \end{cases}$$

   (Where the cases on the right hand side should be considered in order from top to bottom.)

10. (QuantSum) `SUM i:D (E)`, of type `int`, evaluates to the sum of all $\texttt{E}_\texttt{i}$. Formally:

    $M_{\text{int}}(\texttt{SUM i : D (E)}, k) =$

    $$= \begin{cases} 0 & \text{if } \texttt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \texttt{j} \in \texttt{D}_k \ M_{\text{int}}(\texttt{E<i := j>}, k) = \textbf{nil} \\ \sum_{\texttt{j} \in \texttt{D}} M_{\text{int}}(\texttt{E<i := j>}, k) & \text{otherwise} \end{cases}$$

    (Where $\texttt{D}_k = \emptyset$ means that the domain `D` is empty at time step k.)

11. (QuantProd) `PROD i:D (E)`, of type `int`, evaluates to the product of all $\texttt{E}_\texttt{i}$. Formally:

    $M_{\text{int}}(\texttt{PROD i : D (E)}, k) =$

    $$= \begin{cases} 1 & \text{if } \texttt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \texttt{j} \in \texttt{D}_k \ M_{\text{int}}(\texttt{E<i := j>}, k) = \textbf{nil} \\ \prod_{\texttt{j} \in \texttt{D}} M_{\text{int}}(\texttt{E<i := j>}, k) & \text{otherwise} \end{cases}$$

    (Where $\texttt{D}_k = \emptyset$ means that the domain `D` is empty at time step k.)

12. (QuantMin) `$min i:D (E)`, of type `int`, selects the minimum of all $\texttt{E}_\texttt{i}$. Formally:

    $M_{\text{int}}(\texttt{\$min i : D (E)}, k) =$

    $$= \begin{cases} \textbf{nil} & \text{if } \texttt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \texttt{j} \in \texttt{D}_k \ M_{\text{int}}(\texttt{E<i := j>}, k) = \textbf{nil} \\ \min_{\texttt{j} \in \texttt{D}_k} M_{\text{int}}(\texttt{E<i := j>}, k) & \text{otherwise} \end{cases}$$

    (Where $\texttt{D}_k = \emptyset$ means that the domain `D` is empty at time step k.)

13. (QuantMax) `$max i:D (E)`, of type `int`, selects the maximum of all $\texttt{E}_\texttt{i}$. Formally:

    $M_{\text{int}}(\texttt{\$max i : D (E)}, k) =$

    $$= \begin{cases} \textbf{nil} & \text{if } \texttt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \texttt{j} \in \texttt{D}_k \ M_{\text{int}}(\texttt{E<i := j>}, k) = \textbf{nil} \\ \max_{\texttt{j} \in \texttt{D}_k} M_{\text{int}}(\texttt{E<i := j>}, k) & \text{otherwise} \end{cases}$$

    (Where $\texttt{D}_k = \emptyset$ means that the domain `D` is empty at time step k.)

14. (QuantSelectDefault) The optional `<rhs>`, separated by a comma from the `<expr>`, of a `<quantif_expr>` with the `SELECT` operator, denotes the default value, of scalar, tuple or collection type, to select.

15. (QuantSelectType) The type $\mathtt{T_{val}}$ of **the selected value of** $\mathtt{SELECT\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E\ [, R])}$ is defined as follows:

$$\mathtt{T_{val}} = \begin{cases} \mathtt{T_1} & \text{if } \mathtt{n} = 1 \\ \mathtt{tuple}\ \{\mathtt{T_1}, \ldots \mathtt{T_n}\} & \text{otherwise } (\mathtt{n} > 1) \end{cases}$$

where:

$$\mathtt{T_i} = \begin{cases} \text{unsized copy of } \mathtt{D_i} & \text{if } \mathtt{D_i} \text{ is a } \texttt{<type\_domain>} \\ \mathtt{int} & \text{if } \mathtt{D_i} \text{ is a } \texttt{<range>} \end{cases}$$

The type `T` of the `SELECT` expression itself is equal to $\mathtt{T_{val}}$ if there is no default value `R`. Otherwise, it is defined as the union type (see 8.4.3) of $\mathtt{T_{val}}$ and the type of `R`

16. (QuantSelectDomain) The domain `D` of $\mathtt{SELECT\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E\ [, R])}$ is the `n`-fold Cartesian product $\mathtt{D = D_1 \times D_2 \times \ldots \times D_n}$.

    Note that the domain of a selection is a static subset of the type $\mathtt{T_{val}}$ defined by (QuantSelectType) above.

17. (QuantSelectWithoutDefault) At each time step, the selection expression $\mathtt{SELECT\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E)}$ of type `T` and domain `D` as given according to (QuantSelectType) and (QuantSelectDomain) respectively, selects the unique value (if $\mathtt{n} = 1$) or tuple value (if $\mathtt{n} > 1$) from the domain `D` for which the Boolean predicate `E` is true. To simplify the formal definition below, we will extend the tuple notation to scalar values so that `V@0` will mean the same thing as `V` if the value `V` is scalar. Formally:

$$M_{\mathtt{T}}(\mathtt{SELECT\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E)}, k) =$$

$$= \begin{cases} & \text{if } \mathtt{V} \in \mathtt{D} & \text{and} \\ \mathtt{V} & M_{\text{bool}}(\texttt{E<}\mathtt{i_1} \texttt{:= V@0}, \ldots, \mathtt{i_n} \texttt{:= V@(n} - 1\texttt{)>}, k) = \text{true} & \text{and} \\ & \neg\exists(\mathtt{V'} \in \mathtt{D}) : \mathtt{V'} \neq \mathtt{V} \wedge & \\ & M_{\text{bool}}(\texttt{E<}\mathtt{i_1} \texttt{:= V'@0}, \ldots, \mathtt{i_n} \texttt{:= V'@(n} - 1\texttt{)>}, k) \neq \text{false} & \\ \\ \textbf{nil} & \text{otherwise} & \end{cases}$$

18. (QuantSelectWithDefault) The selection expression with a default value is similar to the selection without default (see (QuantSelectWithoutDefault)), except in the case where no value in the domain `D` makes the predicate true at time step k. In that case the default value is selected for time step k. A default value which is a `<collection>` is then interpreted as a tuple value. Formally:

$$M_{\mathtt{T}}(\mathtt{SELECT\ i_1 : D_1,\ i_2 : D_2, \ldots i_n : D_n\ (E, R)}, k) =$$

$$= \begin{cases} & \text{if } \mathtt{V} \in \mathtt{D} \quad \text{and} \\ \mathtt{V} & M_{\text{bool}}(\mathtt{E<i_1 := V@0, \ldots, i_n := V@(n-1)>}, k) = \text{true} \quad \text{and} \\ & \neg\exists(\mathtt{V'} \in \mathtt{D}) : \mathtt{V'} \neq \mathtt{V} \wedge \\ & M_{\text{bool}}(\mathtt{E<i_1 := V'@0, \ldots, i_n := V'@(n-1)>}, k) \neq \text{false} \\ M_{\mathtt{T}}(\mathtt{R}, k) & \text{if } \forall(\mathtt{V} \in \mathtt{D}) : \\ & M_{\text{bool}}(\mathtt{E<i_1 := V@0, \ldots, i_n := V@(n-1)>}, k) = \text{false} \\ \textbf{nil} & \text{otherwise} \end{cases}$$

(Where the cases should be considered in order from top to bottom.)

Static Flag

1. (QuantExprStaticFlag) $\mathcal{SF}(\mathtt{<quantif\_expr>}) = \mathtt{0}$.
2. (QuantVarStaticFlag) The static flag of a quantifier variable `i : D` (the `<id>` of a `<quantif_var>`) is:

$$\mathcal{SF}(\mathtt{i}) = \begin{cases} \mathtt{1} & \text{if D is a <domain>} \\ \mathtt{0} & \text{otherwise (D is \$items(E))} \end{cases}$$

Restrictions

1. (QuantVarUnicity) Two quantifier variables of a quantifier expression may not have the same name.
2. (QuantDomainFinite) Quantification is not allowed over an infinite domain.
3. (QuantDomainStatic) $\mathcal{SF}(\mathtt{D}) \geq \mathtt{1}$ for a `<quantif_var> i : D`, where `D` is a `<domain>`.
4. (ItemsOperandArrayOrFunction) The operand `E` of `$items(E)` must be either of array type, or of function type (with a finite domain).
5. (BoolQuantOperandBool) The operand `E` of `QTF V (E)` where $\mathtt{QTF} \in \{\mathtt{SOME}, \mathtt{ALL}\}$ shall be of type `bool`.
6. (IntQuantOperandInt) The operand `E` of `QTF V (E)` where $\mathtt{QTF} \in \{\mathtt{SUM}, \mathtt{PROD}, \mathtt{\$min}, \mathtt{\$max}\}$ shall be of integer type.
7. (SelectQuantOperandBool) The operand `E` of $\mathtt{SELECT\ V_1, \ldots V_n\ (E\ [, R])}$ shall be of type `bool`.

8. (SelectQuantDefaultCompatible) The operand `R` of `SELECT` $\texttt{V}_1, \ldots \texttt{V}_\texttt{n}$ (`E` [, `R`]) shall be of a type compatible to the type $\texttt{T}_{\texttt{val}}$ of the selected value as defined by (QuantSelectType).

9. (SelectQuantDefaultGround) The operand `R` of `SELECT` $\texttt{V}_1, \ldots \texttt{V}_\texttt{n}$ (`E`, `R`) shall not make reference to any of the quantifier variables $\texttt{V}_1$ to $\texttt{V}_\texttt{n}$. (The following is not allowed: `SELECT i:[0,10] (false, i)`.)

10. (SelectQuantNoItemsDomain) A quantifier expression with operator `SELECT` is not allowed in combination with a domain on the form `$items(E)`. In other words, the quantifier variables $\texttt{V}_\texttt{i}$ of `SELECT` $\texttt{V}_1, \ldots \texttt{V}_\texttt{n}$ (`E` [, `R`]) may not be on the form `i : $items(E)`.

11 Declarations

Declarations declare streams with a name and a type. The declared stream is visible everywhere (except where it is hidden) in the scope where the declaration appears, regardless of the position of the declaration within the scope. An example, where all occurrences of `x` refer to the same variable:

```
Outputs:
 x;
Definitions:
 x := true;
Declarations:
 bool x; // declares x of type bool in the global top-level scope
```

Another, more elaborate, example, which illustrates the hiding mechanism:

```
Outputs:
 x;
Definitions:
 x := true;
Declarations:
 bool x;
Namespaces:
 NS1 {
  Outputs:
   x;  // Refers to the local x
  Declarations:
   bool x;  // Hides the global x in the scope of NS1
 }
 NS2 {
  Outputs:
   x; // Refers to the global x
 }
```

Syntax

(DeclarationSyntax)

```
<declaration>      ::= [<type>] <declarator> {"," <declarator>}
<input>            ::= [<type>] <input_declarator>
                           {"," <input_declarator>}
<input_declarator> ::= <declarator>
                     | "I" "(" <declarator> ")"
```

Semantics

1. (Declaration) A *Declaration* (`<declaration>` or[21] `<input>`) declares a stream with name and type and makes it visible everywhere in its scope (of the namespace of streams), regardless of the position of the declaration.
2. (DeclMultInline) `T` $\texttt{D}_1$, $\texttt{D}_2, \ldots \texttt{D}_\texttt{n}$ is equivalent to the `n` declarations `T` $\texttt{D}_1$, `T` $\texttt{D}_2$, `...T` $\texttt{D}_\texttt{n}$, regardless of whether the $\texttt{D}_\texttt{i}$ denote `<declarator>` or `<input_declarator>`.
3. (DeclNormal) A `<declaration> [T] D` declares a (normal) stream variable. The name of the variable corresponds to the first `<id>` of the declarator `D` and the *Declared Type* is calculated according to procedure `calc_type` in Section 7 using the base type `T` if given and the declarator `D`. If no base type `T` is given, the base type used to calculate the declared type is the type `bool`.
4. (DeclInput) An *Input declaration* (`<input>`) on the form `[T] D` (*i.e.* not an initial input declaration as defined in (DeclInitialInput)) is reducible to a normal declaration (`<declaration>`) `[T] D`. (Note the extra restrictions (InputsFinite) and (InputsUndefined) which apply to input declarations.)
5. (DeclInitialInput) An input declaration (`<input>`) on the form `[T] I(D)` is called an *Initial input declaration* and is reducible to the normal declaration (`<declaration>`) `[T] D`. (Note the extra restriction (DeclInitialInputDefNext) which applies to initial input declarations in addition to those for ordinary input declarations.)

Restrictions

1. (DeclUnicity) A stream variable may not be declared more than once per scope of the namespace of streams.
2. (InputsFinite) The resulting type (as calculated by `calc_type` from the base type and the declarator) of an input declaration shall be either scalar or composite with a finite number of components (either directly or indirectly via other composite components).
3. (InputsUndefined) A stream declared using an input declaration may not be defined, except for initial inputs as specified in (DeclInitialInputDefNext).

[21] Whether the text is parsed as a `<declaration>` or as an `<input>` depends on the context, *i.e.* whether it occurs in a "declarations section" (`<decl_section>`) or in an "inputs section" (`<inputs_section>`) (see Section 16).

4. (DeclInitialInputDefNext) A stream variable declared using an initial input declaration shall be defined with, and only with, a next definition (see Section 12).

5. (UndefinedSized) A stream declared (using a normal or an input declaration) but not defined shall be declared using a type that is neither `int` (without a size restriction), nor a composite type with a component of type `int` (either directly or indirectly via other composite components).

12 Definitions

Definitions define one or several stream variables on the left hand side of the definition symbol `:=` using an expression or a collection on the right hand side. If the stream variables are undeclared they become implicitly declared by the definition, but only if the type of the right hand side is scalar.

Multiple definitions and circular (non-causal) definitions are not allowed. For example:

```
Definitions:
 z := x + y;
 x := z - y; // Not allowed to define x or z in terms of themselves
 y := 10;
 y := 4;     // Not allowed to define y more than once
 X(w) := w;  // Allowed: w keeps its value in the next cycle
```

If the definition is paired with a declaration, the pair must appear in the same scope. An example:

```
Declarations:
 bool x;  // Global and undefined x
Namespaces: N {
  Definitions: x := true; // Implicitly declares and defines a local x
 }
Proof Obligations:
 x;  // The PO is falsifiable
```

Syntax

(DefinitionSyntax)

```
<definition>    ::= <lhs> ":=" <rhs>
                  | "I" "(" <lhs> ")" ":=" <rhs>
                  | "X" "(" <lhs> ")" ":=" <rhs>
                  | <lhs> ":=" <rhs> "," <rhs>
<lhs>           ::= <unfolding>
                  | <id> {<formal_param>}+
<rhs>           ::= <expr>
                  | <collection>
<collection>    ::= "{" <rhs> {"," <rhs>} "}"
<unfold_lhs>    ::= <id> | "_"
<unfolding>     ::= <unfold_lhs> {"," <unfold_lhs>}
```

Semantics

1. (Definition) A *Definition* (`<definition>`) defines one or several stream variables on the *Left hand side* (`<lhs>`) using an expression or collection on the *Right hand side* (`<rhs>`).

2. (DefUndeclared) If a stream variable on the left hand side of a `<definition>` is undeclared in the scope (excluding ascendant and descendant scopes) of the definition, then it is implicitly declared by the definition and becomes visible everywhere in the scope (of the namespace of streams) of the definition, regardless of the position of the definition. The type of the variable is inferred according to (DefUndeclaredType) below.

3. (DefUndeclaredType) A stream variable `V` which is implicitly declared by a definition according to (DefUndeclared) above is assigned the type `T` according to the following table. $\texttt{T}_\texttt{E}$ denotes the type of the right hand side `E`.

Definition	**Condition**	`T`
`X(V) := E`	(none)	`bool`
`V` := $\texttt{E}_1$, $\texttt{E}_2$	(none)	`bool`
`V := E`	`V` is defined (directly or indirectly) in terms of itself (recursive definition)	`bool`
	otherwise	$\texttt{T}_\texttt{E}$

4. (DefAlways) `V := E` (always definition) where `E` is a stream expression of type $\texttt{T}_\texttt{E}$ and `V` is a stream variable of type `T`, defines the value of `V` in all time steps, using the expression `E`. Formally, for all time steps k:

$$M_\texttt{T}(\texttt{V}, k) = \begin{cases} M_{\texttt{T}_\texttt{E}}(\texttt{E}, k) & \text{if } M_{\texttt{T}_\texttt{E}}(\texttt{E}, k) \in \texttt{T} \\ \textbf{nil} & \text{otherwise} \end{cases}$$

5. (DefInit) `I(V) := E` (initial definition) defines the value of `V` of type `T`, in the first time step only, using the expression `E` of type $\texttt{T}_\texttt{E}$. Formally:

$$M_\texttt{T}(\texttt{V}, 0) = \begin{cases} M_{\texttt{T}_\texttt{E}}(\texttt{E}, 0) & \text{if } M_{\texttt{T}_\texttt{E}}(\texttt{E}, 0) \in \texttt{T} \\ \textbf{nil} & \text{otherwise} \end{cases}$$

6. (DefNext) `X(V) := E` (next definition) defines the value of `V` of type `T`, in the next time step, using the expression `E` of type $\mathtt{T_E}$. Formally, for all time steps k:
$$M_{\mathtt{T}}(\mathtt{V}, k+1) = \begin{cases} M_{\mathtt{T_E}}(\mathtt{E}, k) & \text{if } M_{\mathtt{T_E}}(\mathtt{E}, k) \in \mathtt{T} \\ \textbf{nil} & \text{otherwise} \end{cases}$$

7. (DefLatch) `<lhs> := E1, E2` (latch definition) is equivalent to the two definitions `I(<lhs>) := E1` and `X(<lhs>) := E2`.

8. (DefFunctionInit) `I(V FP):= E` is equivalent to the two definitions: `I(V):= V'` and `V' FP:= E` where `V'` is a fresh variable of same type as `V`.

9. (DefFunctionNext) `X(V FP):= E` is equivalent to the two definitions: `X(V):= V'` and `V' FP:= E` where `V'` is a fresh variable of same type as `V`.

10. (DefArrayFunction) An *Array or Function definition* `V FP := E` is equivalent to `V := lambda DS : FP := E` where `DS` is obtained by solving the equation $\mathtt{T_V} = \mathtt{calc\_type}(\mathtt{T_E}, \mathtt{DS})$ where $\mathtt{T_V}$ is the declared type of `V` and $\mathtt{T_E}$ is the type of `E`.

11. (DefCollectionRhs) A `<lhs>` of ordered composite type may be defined using a collection (`<collection>`) on the right hand side. In such a case component number k of `<lhs>` is defined by the `<rhs>` number k of the collection. Such components may themselves be recursively defined by collections, if they are of ordered composite type. Formally, a definition $\mathtt{V} := \{\mathtt{R_1}, \ldots \mathtt{R_n}\}$ is equivalent to $\mathtt{V} := ((\mathtt{lambda}[\mathtt{1}] : [\mathtt{i}] := \mathtt{V'})\ \mathtt{with}\ [\mathtt{0}] := \{\mathtt{R_1}, \ldots \mathtt{R_n}\})[\mathtt{0}]$ where `V'` is a fresh variable with the same type as `V`.

    The type of a `<rhs>` which is a `<collection>` is a collection type as defined in Section 8.4.1.

12. (DefUnfolding) An *Unfolding definition* is a definitions where several variables or wildcards (`"_"`) occur comma-separatedly on the left hand side (using an `<unfolding>`). If the right hand side is of composite type (including collections) with n ordered components then such a definition with k variables and l wildcards, with $n = k+l$ on the left hand side is equivalent to k ordinary definitions (*i.e.* with a single variable on the left hand side) of the appropriate kind ("always", "initial" or "next") where the variable at index i (with $1 \leq i \leq n$) on the left hand side is defined using the component (or collection element) number i on the right hand side.

Static Flag

1. (CollectionStaticFlag) $\mathcal{SF}(\mathtt{<collection>}) = \mathtt{0}$.

2. (DefAlwaysStaticFlag) $\mathcal{SF}(\mathtt{V}) = \mathtt{min}(\mathtt{1}, \mathcal{SF}(\mathtt{E}))$ for `V := E`.

3. (DefLatchStaticFlag) $\mathcal{SF}(\mathtt{V}) = \mathtt{0}$ for `X(V) := E` or `V := E1, E2`.

Restrictions

1. (DefCausality) A scalar stream variable or component may not have its value in time step n be defined, directly or indirectly, in terms of its own value in any time step $k \geq n$.

   The cyclicity criterion, *i.e.* the precise criterion for when such a variable or component is considered defined in terms of its own value will depend on the reasoning power of tools implementing HLL – especially when it comes to recursive array and function definitions – and is thus outside the scope of this document.

2. (DefUnicity) A stream variable may not have its value in time step n be defined more than once.

3. (DefCompleteness) A stream variable with an initial definition shall also have a next definition. (The converse is not required however, *i.e.* the initial value may be left undefined/free.)

4. (DefUndeclaredLhsScalarRhs) If the variable on the `<lhs>` is undeclared, then there shall be no `<formal_param>` and the type of the `<rhs>` shall be scalar.

5. (DefRhsTypeAssignableToLhsType) The type of the `<rhs>` shall be assignable to the type of the `<lhs>`. For this purpose, the type of an `<lhs>` on the form `V FP` (where `FP` corresponds to the `{<formal_param>}+`) is derived in the same way as for the corresponding projection expression (see Section 10.6).

6. (LatchesSized) A stream defined with a latch or next definition shall not be declared with a type that is either `int` (without a size restriction), or a composite type with a component of type `int` (either directly or indirectly via other composite components).

7. (DefUnfoldingCompatibleRhs) When the left hand side (`<lhs>`) of a definition is on the form of an `<unfolding>`, then the right hand side (`<rhs>`) shall be of ordered composite type, but shall not be of multidimensional array type nor of multivariate function type, and the number of variables plus the number of wildcards on the left hand side shall equal the number of components of the type of the right hand side.

13 Constants

Syntax

(ConstantSyntax)

```
<constant> ::= "bool" <id> ":=" <expr>
             | "int" <id> ":=" <expr>
```

Semantics

1. (ConstantDef) A *Constant definition* (`<constant>`) declares and defines a named constant stream.
2. (ConstantDefIsaDeclDef) Semantically, `T C := E` is equivalent to:

   ```
   Declarations:
    T C;
   Definitions:
    C := E;
   ```

   Furthermore, all restrictions that apply to the latter language construct also apply to the former (as according to (ConstantDefInheritedRestrictions)). However, there are two minor differences between the language constructs:

   (a) The static flag of `C` as defined by the former construct is 2 (as according to (ConstantStaticFlag)), whereas it is at most 1 for the `C` defined by the latter construct (as according to (DefAlwaysStaticFlag) of Section 12).

   (b) The additional restriction (ConstantDefRhsConstant) applies only to the former construct.

Static Flag

1. (ConstantStaticFlag) $\mathcal{SF}(\texttt{C}) = 2$ for a `<constant> T C := E`.

Restrictions

1. (ConstantDefRhsConstant) $\mathcal{SF}(\texttt{E}) = 2$ for a `<constant> T C := E`.
2. (ConstantDefInheritedRestrictions) All restrictions that apply to the language construct:

   ```
   Declarations:
    T C;
   Definitions:
    C := E;
   ```

   also apply to a `<constant> T C := E`.

14 Constraints

Syntax

(ConstraintSyntax)

```
<constraint> ::= <expr>
               | "I" "(" <expr> ")"
```

Semantics

1. (ConstraintAlways) An *Always constraint* `E` (an `<expr>`) corresponds to the proposition $\Box$`E`.
2. (ConstraintInitial) An *Initial constraint* `I(E)` corresponds to the proposition $\mathbb{I}$ `E`.

Restrictions

1. (ConstraintBool) Constraints shall be expressions of type `bool`.

15 Proof Obligations

Syntax

(PoSyntax)

```
<po> ::= <expr>
```

Semantics

1. (PoBool) A *Proof obligation* (PO) `E` (an `<expr>`) of type `bool` corresponds to the proposition □`E`.
2. (PoComposite) A proof obligation `E` of type `T`, which is an array or function type with `bool` as component type, corresponds to the proposition □(`ALL e : $items(E) (e)`).
3. (PoValid) A proof obligation is *Valid* iff it is a consequence of all the constraints within the same (global) HLL text (regardless of whether the constraints appear in the same user namespace or not). Note that if there is no model M which does not falsify the constraints (for example if the constraints are contradictory), then any proof obligation is trivially valid.
4. (PoFalsifiable) A proof obligation which is not valid takes the value **nil** or the value `false` at some time step k in some model M which does not falsify the constraints. If the proof obligation takes the value **nil** we say that the proof obligation is not well-defined. Otherwise, if it takes the value `false`, we say that the proof obligation is *Falsifiable*.

Restrictions

1. (PoType) Proof obligations shall be expressions of type `bool`, or of array or function type with `bool` as component type. (This means that the types `bool^(N)` or `bool^(N, M)` are accepted but not the type `bool^(N)^(M)`.)

16 Sections

Syntax

(SectionSyntax)

```
<HLL>                ::= {<section>}
<section>            ::= <constants_section>
                       | <types_section>
                       | <inputs_section>
                       | <decl_section>
                       | <def_section>
                       | <outputs_section>
                       | <constr_section>
                       | <po_section>
                       | <namespaces_section>

<constants_section>  ::= <constants> ":"    {<constant> ";"}
<types_section>      ::= <types> ":"        {<type_def> ";"}
<inputs_section>     ::= <inputs> ":"       {<input> ";"}
<decl_section>       ::= <declarations> ":" {<declaration> ";"}
<def_section>        ::= <definitions> ":"  {<definition> ";"}
<outputs_section>    ::= <outputs> ":"      {<expr> ";"}
<constr_section>     ::= <constraints> ":"  {<constraint> ";"}
<po_section>         ::= <proof> <obligations> ":" {<po> ";"}
<namespaces_section> ::= <namespaces> ":"   {<namespace>}

<constants>          ::= "Constants"    | "constants"
<types>              ::= "Types"        | "types"
<inputs>             ::= "Inputs"       | "inputs"
<declarations>       ::= "Declarations" | "declarations"
<definitions>        ::= "Definitions"  | "definitions"
<constraints>        ::= "Constraints"  | "constraints"
<proof>              ::= "Proof"        | "proof"
<obligations>        ::= "Obligations"  | "obligations"
<outputs>            ::= "Outputs"      | "outputs"
<namespaces>         ::= "Namespaces"   | "namespaces"
```

Semantics

1. (HLLText) An HLL text (`<HLL>`) is a (possibly empty) list of sections.
2. (GlobalTopLevelScopes) The *Global top-level* scopes of an HLL text `<HLL>` that is not nested within a `<namespace>` encompass the entire text.
3. (SectionOrderIrrelevant) The order of the sections of an HLL text and the order of items within the sections have no impact on the semantics of the text.
4. (SectionsReopen) Sections may be opened any number of times.

Restrictions

1. (OutputsFinite) The `<expr>` in an `<outputs>` section shall be either of scalar type or of a composite type with a finite number of scalar components (either directly or indirectly via other composite components).

A Syntax Overview

```
<HLL>           ::= {<section>}
<section>       ::= <constants_section>
                  | <types_section>
                  | <inputs_section>
                  | <decl_section>
                  | <def_section>
                  | <outputs_section>
                  | <constr_section>
                  | <po_section>
                  | <namespaces_section>

<constants_section>  ::= <constants> ":"    {<constant> ";"}
<types_section>      ::= <types> ":"        {<type_def> ";"}
<inputs_section>     ::= <inputs> ":"       {<input> ";"}
<decl_section>       ::= <declarations> ":" {<declaration> ";"}
<def_section>        ::= <definitions> ":"  {<definition> ";"}
<outputs_section>    ::= <outputs> ":"      {<expr> ";"}
<constr_section>     ::= <constraints> ":"  {<constraint> ";"}
<po_section>         ::= <proof> <obligations> ":" {<po> ";"}
<namespaces_section> ::= <namespaces> ":"   {<namespace>}

<constants>     ::= "Constants"    | "constants"
<types>         ::= "Types"        | "types"
<inputs>        ::= "Inputs"       | "inputs"
<declarations>  ::= "Declarations" | "declarations"
<definitions>   ::= "Definitions"  | "definitions"
<constraints>   ::= "Constraints"  | "constraints"
<proof>         ::= "Proof"        | "proof"
<obligations>   ::= "Obligations"  | "obligations"
<outputs>       ::= "Outputs"      | "outputs"
<namespaces>    ::= "Namespaces"   | "namespaces"

<namespace>     ::= <id> "{" <HLL> "}"

<constant>      ::= "bool" <id> ":=" <expr>
                  | "int" <id> ":=" <expr>

<type_def>      ::= <type> <declarator> {"," <declarator>}
                  | <enum_def>
                  | <sort_def>
<declarator>    ::= <id> {<declarator_suffix>}
<declarator_suffix> ::= "[" <expr_list> "]"
                  | "(" <type_list> ")"
<type>          ::= "bool"
                  | <integer>
                  | <tuple>
                  | <structure>
```

```
                  | <array>
                  | <function>
                  | <named_type>
<integer>       ::= "int"
                  | "int" <sign>
                  | "int" <range>
<sign>          ::= "signed" <id_or_int>
                  | "unsigned" <id_or_int>
<id_or_int>     ::= <id>
                  | <int_literal>
<range>         ::= "[" <expr> "," <expr> "]"
<enum_def>      ::= <enumerated> <id>
<enumerated>    ::= "enum" "{" <id_list> "}"
<tuple>         ::= "tuple" "{" <type_list> "}"
<structure>     ::= "struct" "{" <member_list> "}"
<sort_def>      ::= "sort" [ <sort_contrib> "<" ] <id>
<sort_contrib>  ::= <path_id_list>
                  | "{" <id_list> "}"
<array>         ::= <type> "^" "(" <expr_list> ")"
<function>      ::= "(" <type> {"*" <type>} "->" <type> ")"
<named_type>    ::= <path_id>
<type_list>     ::= <type> {"," <type>}
<member_list>   ::= <id> ":" <type> {"," <id> ":" <type>}

<input>         ::= [<type>] <input_declarator>
                        {"," <input_declarator>}
<input_declarator> ::= <declarator>
                     | "I" "(" <declarator> ")"

<declaration>   ::= [<type>] <declarator> {"," <declarator>}

<po>            ::= <expr>
<constraint>    ::= <expr>
                  | "I" "(" <expr> ")"

<definition>    ::= <lhs> ":=" <rhs>
                  | "I" "(" <lhs> ")" ":=" <rhs>
                  | "X" "(" <lhs> ")" ":=" <rhs>
                  | <lhs> ":=" <rhs> "," <rhs>

<lhs>           ::= <unfolding>
                  | <id> {<formal_param>}+
<rhs>           ::= <expr>
                  | <collection>

<collection>    ::= "{" <rhs> {"," <rhs>} "}"

<unfold_lhs>    ::= <id> | "_"
<unfolding>     ::= <unfold_lhs> {"," <unfold_lhs>}
```

```
<formal_param>  ::= "[" <id_list> "]"
                  | "(" <id_list> ")"

<accessor>      ::= "." <id>
                  | "." <int_literal>
                  | "[" <expr_list> "]"
                  | "(" <expr_list> ")"

<expr>          ::= <ite_expr>
                  | <lambda_expr>
                  | <binop_expr>
                  | <membership_expr>
                  | <unop_expr>
                  | <proj_expr>
<ite_expr>      ::= "if"   <expr> "then" <expr>
                    {"elif" <expr> "then" <expr>}
                    "else" <expr>
<lambda_expr>   ::= "lambda" {<declarator_suffix>}+ ":"
                    {<formal_param>}+ ":=" <expr>
<binop_expr>    ::= <expr> <binop> <expr>
<membership_expr> ::= <expr> ":" <domain>
<domain>        ::= <range>
                  | <type_domain>
<type_domain>   ::= <named_type>
                  | "bool"
                  | "int"
<unop_expr>     ::= <unop> <expr>
<proj_expr>     ::= <closed_expr> { <accessor> }

<closed_expr>   ::= <bool_literal>
                  | <int_literal>
                  | <named_expr>
                  | <next_expr>
                  | <pre_expr>
                  | <fun_expr>
                  | <cast_expr>
                  | <with_expr>
                  | <case_expr>
                  | <quantif_expr>
                  | "(" <expr> ")"

<bool_literal>  ::= <true>
                  | <false>

<dec_literal>   ::= [0-9](_?[0-9])*
<bin_literal>   ::= 0[Bb][0-1][_?[0-1])*
<hex_literal>   ::= 0[Xx][0-9A-Fa-f](_?[0-9A-Fa-f])*
<int_literal>   ::= <dec_literal>
                  | <hex_literal>
                  | <bin_literal>
```

```
<named_expr>    ::= <path_id>
<next_expr>     ::= "X" "(" <expr> ")"
<pre_expr>      ::= ("pre" | "PRE") ["<" <type> ">"]
                    "(" <expr> ["," <expr>] ")"
<fun_expr>      ::= <fop> "(" <expr_list> ")"
<cast_expr>     ::= "cast" "<" <type> ">" "(" <expr> ")"
<with_expr>     ::= "(" <expr> "with" {<accessor>}+ ":=" <rhs> ")"

<case_expr>     ::= "(" <expr_list> {<case_item>}+ ")"
<case_item>     ::= "|" <pattern_list> "=>" <expr>
<pattern>       ::= <expr>
                  | <named_type> ( <id> | "_" )
                  | "_"
<pattern_list>  ::= <pattern> { "," <pattern> }

<quantif_expr>  ::= <quantifier> <quantif_var> {"," <quantif_var>}
                        ( "(" <expr> ["," <rhs>] ")" | <quantif_expr> )
<quantif_var>   ::= <id> ":" <quantif_domain>
<quantif_domain> ::= <domain>
                   | "$items" "(" <expr> ")"

<binop>         ::= "#" | "&"  | "#!" | "->" | "<->"
                  | ">" | ">=" | "<"  | "<="
                  | "=" | "==" | "!=" | "<>"
                  | "+" | "-"  | "*"  | "%"  | "^"   | "<<" | ">>"
                  | "/" | "/>" | "/<"
<unop>          ::= "~" | "-"
<fop>           ::= "$min"
                  | "$max"
                  | "$abs"
                  | "$or"
                  | "$and"
                  | "$xor"
                  | "$not"
                  | "bin2u"
                  | "u2bin"
                  | "bin2s"
                  | "s2bin"
                  | "population_count_eq"
                  | "population_count_lt"
                  | "population_count_gt"
<expr_list>     ::= <expr> {"," <expr>}

<id_list>       ::= <id> {"," <id>}
<path_id_list>  ::= <path_id> {"," <path_id>}
<path_id>       ::= <relative_path> <id>
                  | <absolute_path> <id>
<relative_path> ::= { <id> "::" }
<absolute_path> ::= "::" { <id> "::" }
```

```
<id>            ::= regexp: [a-zA-Z_][a-zA-Z0-9_]*
                  | regexp: '[^\n']+'
                  | regexp: "[^\n"]+"
<true>          ::= "true"  | "TRUE"  | "True"
<false>         ::= "false" | "FALSE" | "False"
<quantifier>    ::= "SOME" | "ALL"  | "SUM"  | "PROD"
                  | "CONJ" | "DISJ" | "$min" | "$max"
                  | "SELECT"
```

A.1 Operator Precedence and Associativity

The precedence of expressions is given below in order from lowest to highest (same line means same precedence). The information below is a copy of (ExprPrecedence).

1. `<ite_expr>`, `<lambda_expr>`
2. `<binop_expr>`, `<membership_expr>`
3. `<unop_expr>`
4. `<proj_expr>`

The precedence of the binary operators is given below in order from lowest to highest, together with their associativity. Same line means same precedence. The information below is a copy of (BinopGrouping).

Precedence	*Associativity*
`<-> #!`	left
`->`	right
`#`	left
`&`	left
`> >= < <= = == != <>`	left
`<< >>`	left
`+ -`	left
`* / /< /> %`	left
`^`	right

B Reserved Words

```
ALL
assumptions
Assumptions
bin2s
bin2u
block
blocks
Blocks
bool
cast
CONJ
constants
Constants
constraints
Constraints
declarations
Declarations
definitions
Definitions
DISJ
elif
else
enum
false
False
FALSE
guarantees
Guarantees
I
if
inputs
Inputs
int
lambda
namespaces
Namespaces
new
obligations
Obligations
outputs
Outputs
population_count_eq
population_count_gt
population_count_lt
pre
PRE
PROD
proof
Proof
s2bin
SELECT
signed
SOME
sort
struct
SUM
then
true
True
TRUE
tuple
types
Types
u2bin
unsigned
with
X
```

C Restrictions Overview

Below are listed the direct and indirect restrictions which apply to each language construct of HLL. The indirect restrictions, as explained in Section 2.4.2, are due to language constructs being defined by translation or reduction to other language constructs which in turn have restrictions on them. These restrictions also apply, indirectly, to the language constructs defined by the translation or reduction, and are thus listed in the table below.

On the other hand, some language constructs contain sub-constructs which have specific restrictions applied to them. The restrictions which apply to sub-constructs are not listed as applicable to the parent construct (to avoid duplication). An example is projections, `<proj_expr>`, which are defined using the sub-construct `<accessor>` where specific restrictions apply.

When a restriction applies only to a certain part of a language construct, we will occasionally, for the benefit of the reader and especially if the restriction is indirect, prefix the restriction with the part of the language construct to which it applies.

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 4		
<id>		(ReservedWords)
Section 5		
<path_id>		(PathIdNoImplicitDecl)
Section 7		
<declarator_suffix>		(DeclArrayDimInteger) (DeclArrayDimConstant) (DeclFunctionParamScalar)
Section 8		
<type> <declarator>	<type_def>	(TypeDefUnicity) (TypeDefCausality)
<enum_def>	<type_def>	(TypeDefUnicity) (TypeDefCausality)
<sort_def>	<type_def>	(TypeDefCausality)
int signed E	<integer>	(IntSizeInteger) (IntSizeConstant) (SignedBitsPositive) (IntSizeNotNil)
int unsigned E	<integer>	(IntSizeInteger) (IntSizeConstant) (UnsignedBitsNonNegative) (IntSizeNotNil)
int $[E_1, E_2]$	<integer>	(IntSizeInteger) (IntSizeConstant) (IntSizeNotNil)
<enumerated>	<enum_def>	(EnumValueUnicity)
$\{V_1, V_2, \ldots V_n\}$	<sort_contrib>	(SortValueUnicity)
sort $S_1, S_2, \ldots S_k$ < S	<sort_def>	(SortSubTypes)
<member_list>	<structure>	(StructCompUnicity)
<function>		(FunctionDomainScalar)
<array>		(ArrayDimConstant) (ArrayDimNotNil)
<named_type>		(NamedTypeRef)
Section 9		
$[E_1,\ E_2, \ldots E_n]$	<accessor>	(ArrayIndexInteger)
$(E_1,\ E_2, \ldots E_n)$	<accessor>	(FunctionInputScalar)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 10		
<ite_expr>		(IteCondBool) (IteBranchesCompatible)
<lambda_expr>		(LambdaParamUnicity) (LambdaParamsBound) (LambdaParamsMatch) (LambdaTypeCheck)
E_1 # E_2	<binop_expr>	(BoolOrEquivOperandsBool)
E_1 & E_2	<binop_expr>	(BoolOrEquivOperandsBool)
E_1 #! E_2	<binop_expr>	(BoolOrEquivOperandsBool)
E_1 -> E_2	<binop_expr>	(BoolOrEquivOperandsBool)
E_1 <-> E_2	<binop_expr>	(BoolOrEquivOperandsBool)
E_1 > E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 >= E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 < E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 <= E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 = E_2	<binop_expr>	(EqOperandsFiniteCompatible)
E_1 == E_2	<binop_expr>	(EqOperandsFiniteCompatible)
E_1 != E_2	<binop_expr>	(EqOperandsFiniteCompatible)
E_1 <> E_2	<binop_expr>	(EqOperandsFiniteCompatible)
E_1 + E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 - E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 * E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 ^ E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 << E_2	<binop_expr>	(IntCoreBinopOperandsInt) (SecondShiftOperandStatic) (SecondShiftOperandNonNegative)
E_1 >> E_2	<binop_expr>	(IntCoreBinopOperandsInt) (SecondShiftOperandStatic) (SecondShiftOperandNonNegative)
E_1 / E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 /> E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 /< E_2	<binop_expr>	(IntCoreBinopOperandsInt)
E_1 % E_2	<binop_expr>	(IntCoreBinopOperandsInt)
<domain>		(DomainScalar)
<membership_expr>		(MembershipDomainCompatible)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 10 (continued)		
`~E`	`<unop_expr>`	(BoolNegOperandBool)
`-E`	`<unop_expr>`	(IntNegOperandInt)
`<proj_expr>`		(ProjAccCompatible)
`<pre_expr>`		(PreOperandsAssignable)
`$min(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) (IntCoreBinopOperandsInt)
`$max(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) (IntCoreBinopOperandsInt)
`$abs(E)`	`<fun_expr>`	(FunopUnaryCard) (IntCoreBinopOperandsInt)
`$or(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) (ProjAccCompatible)
`$and(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) (ProjAccCompatible)
`$xor(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) (ProjAccCompatible)
`$not(E)`	`<fun_expr>`	(FunopUnaryCard) (ProjAccCompatible)
`bin2u(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) E_1: (ProjAccCompatible) E_1: (IteCondBool) E_2: (IntCoreBinopOperandsInt) E_2: (QuantDomainStatic)
`u2bin(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) E_1: (ProjAccCompatible) E_2: (DeclArrayDimInteger) E_2: (DeclArrayDimConstant)
`bin2s(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) E_1: (ProjAccCompatible) E_1: (IteCondBool) E_2: (IntCoreBinopOperandsInt) E_2: (ArrayIndexInteger) E_2: (QuantDomainStatic)
`s2bin(`E_1`, `E_2`)`	`<fun_expr>`	(FunopBinaryCard) E_1: (ProjAccCompatible) E_2: (DeclArrayDimInteger) E_2: (DeclArrayDimConstant)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 10 (continued)		
population_count_lt($E_1, \ldots E_n$, K)	<fun_expr>	(PopCountNumberStatic) E_i: (IteCondBool) K: (IntCoreBinopOperandsInt)
population_count_gt($E_1, \ldots E_n$, K)	<fun_expr>	(PopCountNumberStatic) E_i: (IteCondBool) K: (IntCoreBinopOperandsInt)
population_count_eq($E_1, \ldots E_n$, K)	<fun_expr>	(PopCountNumberStatic) E_i: (IteCondBool) K: (IntCoreBinopOperandsInt)
<cast_expr>		(CastTargetIntImpl) (ProjAccCompatible)
<with_expr>		(WithAccCompatible) (WithRhsAssignable)
<case_expr>		(CaseSwitchesScalar) (CasePatternsCompatible) (CaseBranchesCompatible) (CasePatternExprConstant) (CasePatternTypeSort) (CaseCapturingVarUnicity)
SOME $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (BoolQuantOperandsBool)
ALL $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (BoolQuantOperandsBool)
DISJ $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (BoolQuantOperandsBool)
CONJ $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (BoolQuantOperandsBool)
SUM $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (IntQuantOperandsInt)
PROD $i_1 : D_1, \ldots i_n : D_n$ (E)	<quantif_expr>	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (IntQuantOperandsInt)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 10 (continued)		
`$min` $\texttt{i}_1 : \texttt{D}_1, \ldots \texttt{i}_n : \texttt{D}_n$ (`E`)	`<quantif_expr>`	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (IntQuantOperandsInt)
`$max` $\texttt{i}_1 : \texttt{D}_1, \ldots \texttt{i}_n : \texttt{D}_n$ (`E`)	`<quantif_expr>`	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (IntQuantOperandsInt)
`SELECT` $\texttt{i}_1 : \texttt{D}_1, \ldots \texttt{i}_n : \texttt{D}_n$ (`E`)	`<quantif_expr>`	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (SelectQuantOperandBool) (SelectQuantNoItemsDomain)
`SELECT` $\texttt{i}_1 : \texttt{D}_1, \ldots \texttt{i}_n : \texttt{D}_n$ (`E`, `R`)	`<quantif_expr>`	(QuantVarUnicity) (QuantDomainFinite) (QuantDomainStatic) (SelectQuantOperandBool) (SelectQuantDefaultCompatible) (SelectQuantDefaultGround) (SelectQuantNoItemsDomain)
`$items`(`E`)	`<quantif_domain>`	(ItemsOperandArrayOrFunction)
Section 11		
`<declaration>`		(DeclUnicity) (UndefinedSized)
`<input>`		(DeclUnicity) (InputsFinite) (InputsUndefined) (DeclInitialInputDefNext) (UndefinedSized)
Section 12		
`V:= R`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType)
`I(V):= R`	`<definition>`	(DefCausality) (DefUnicity) (DefCompleteness) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 12 (continued)		
`X(V):= R`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized)
`V:= R`$_1$`,` `R`$_2$	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized)
`V FP:= R`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LambdaParamUnicity)
`I(V FP):= R`	`<definition>`	(DefCausality) (DefUnicity) (DefCompleteness) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LambdaParamUnicity)
`X(V FP):= R`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized) (LambdaParamUnicity)
`V FP:= R`$_1$`,` `R`$_2$	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized) (LambdaParamUnicity)
`V`$_1$`,...` `V`$_n$`:= R`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (DefUnfoldingCompatibleRhs)
`I(V`$_1$`,...` `V`$_n$`):= R`	`<definition>`	(DefCausality) (DefUnicity) (DefCompleteness) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (DefUnfoldingCompatibleRhs)

Language Construct	*Parent Construct*	*Applicable Restrictions*
Section 12 (continued)		
`X(V1,...Vn):= E`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized) (DefUnfoldingCompatibleRhs)
`V1,...Vn:= E1,E2`	`<definition>`	(DefCausality) (DefUnicity) (DefUndeclaredLhsScalarRhs) (DefRhsTypeAssignableToLhsType) (LatchesSized) (DefUnfoldingCompatibleRhs)
Section 13		
`<constant>`		(ConstantDefRhsConstant) (ConstantDefInheritedRestrictions) (DeclUnicity) (DefUnicity) (DefRhsTypeAssignableToLhsType)
Section 14		
`<constraint>`		(ConstraintBool)
Section 15		
`<po>`		(PoType)
Section 16		
`E;`	`<outputs_section>`	(OutputsFinite)

D Glossary

For the purposes of this document, the following terms and abbreviations are used.

1. **ascendant scope**
   an enclosing scope, *i.e.* one higher up in the scope stack; see Section 2.3
2. **ASCII**
   American Standard Code for Information Interchange, a character encoding standard
3. **assignable / assignability**
   refers to the assignability relation between types, see Section 8; a stream expression `E1` is *assignable* to a stream expression `E2` iff the type of `E1` is assignable to the type of `E2`
4. **combinatorial function**
   a mapping from values to values (usually a mapping from one or several values to a single value)
5. **combinatorial operator**
   see *combinatorial function*
6. **compatible / compatibility**
   (sometimes preceded by *type*) refers to the compatibility relation between types, see Section 8; a stream expression `E1` is *compatible* with a stream expression `E2` iff the type of `E1` is compatible with the type of `E2`
7. **component**
   either a component (type) of a composite type, or a component (stream) of a stream of composite type; similar terms used elsewhere would be *e.g.* struct field, array element, function output – all those entities are collectively called components in this document
8. **composite**
   equivalent to *non-scalar*;
   (noun) a stream of composite type
   (adj.) applied to a stream it means a composite stream; applied to a type it means a composite type; applied to a value it means a value of a composite type
9. **composite type**
   equivalent to *non-scalar type*; any type defined in Section 8.2
10. **consequence**
    see Section 2.1.4
11. **constant**
    refers either to a stream defined using the nonterminal `<constant>` of Section 13, or to any stream with a static flag of 2.
12. **declared type**
    see Section 11

13. **defined variable**
    a variable with a definition, *i.e.* that occurs on the left hand side of a definition; see Section 12

14. **descendant scope**
    an enclosed scope, *i.e.* one further down in the scope stack; see Section 2.3

15. **directly defined**
    (adj.) refers to an entity $\texttt{E}_1$ which is defined in terms of another entity $\texttt{E}_2$ without there being any intermediate entity $\texttt{E}_3$ in between $\texttt{E}_1$ and $\texttt{E}_2$; see also *indirectly defined*

16. **domain**
    depending on the context, refers to one of:
    (a) the nonterminal `<domain>`, defined in Section 10.4,
    (b) the nonterminal `<quantif_domain>`, defined in Section 10.7.10,
    (c) the parameter types of a function type, see Section 8.2.3,
    (d) the domain of an array type, which amounts to the parameter types of the equivalent function type, see Section 8.2.4, or
    (e) the domain of the `SELECT` operator, which is defined by (QuantSelectDomain).

17. **EBNF**
    Extended Backus-Naur Form, see also Section 2.4.1

18. **equivalent to**
    see Section 2.4.2

19. **explicit grouping**
    the process of adding parentheses around all subexpressions (or groups) in a text; for example, expressions such as "`a & b & c`" and "`a + b * c`" are explictly grouped into respectively "`((a & b) & c)`" and "`(a + (b * c))`"

20. **free**
    (adj.) applied to a variable it means a variable that is free to take any value of its type in each time step; input variables are free; quantifier variables or defined variables are not free

21. **fresh**
    (adj). applied to the identifier of an entity it means that another entity with the same identifier is not visible within the same namespace; (an entity with a fresh name does not hide another entity)

22. **function accessor**
    see (AccFunction) of Section 9

23. **function value**
    see *combinatorial function*

24. **group / grouping**
    see *implicit grouping* or *explicit grouping*

25. **hide / hiding**
    refers to variable hiding see Section 2.3

26. **HLL**
    High Level Language

27. **HLL function**
    a stream of combinatorial functions (function values) which, together with a function accessor, maps streams to streams by point-wise application of the combinatorial function in each time step on the values of the stream operands in the same time step; HLL functions are distinct from *temporal functions* which are general mappings of streams to streams

28. **iff**
    if and only if

29. **implicit grouping**
    the process by which a parser uses syntax, precedence and associativity rules to map a text into an abstract syntax tree (AST) where each group (or subexpression) is represented by its own vertex

30. **implicit input**
    (sometimes followed by *variable*) a free variable that is not an (explicit) input

31. **indirectly defined**
    (adj.) refers to an entity $\texttt{E}_1$ which is defined in terms of another entity $\texttt{E}_2$ via an intermediate entity $\texttt{E}_3$; *e.g.* $\texttt{E}_1$ `:=` $\texttt{E}_3$; $\texttt{E}_3$ `:=` $\texttt{E}_2$;

32. **initial input**
    (sometimes followed by *variable*) a memory variable which is undefined in the first time step (*i.e.* it has only a next definition, see Section 12)

33. **input**
    (sometimed preceded by *explicit*; sometimes followed by *variable*) refers to variables declared using the nonterminal `<input>`, see Section 11

34. **integer implementation type**
    see Related Notation 1 of Section 8.1.2

35. **integer range type**
    see Related Notation 2 of Section 8.1.2

36. **integer type**
    any type defined in Section 8.1.2

37. **latch**
    (sometimes followed by *variable*) a variable defined using a latch or a next definition, see Section 12

38. **literal**
    (sometimes followed by *value*) refers to an immediate value such as the integer literal `1234` or the Boolean literal `true`.

39. **memory**
    either a pre expression or a latch; if followed by *variable* it means a latch.

40. **model**
    (sometimes preceded by *stream*) see Section 2.1.1.

41. **multidimensional**
    (adj). applied to an array type it means that the type has more than one dimension, see Section 8.2.4

42. **multivariate**
    (adj). applied to a function type it means that the type has more than one parameter type, see Section 8.2.3

43. **namespace**
    see Section 2.3; not to be confused with *user namespace*

44. **nil**
    see Section 2.1.2

45. **parameter**
    refers to the nonterminal `<formal_param>` defined in Section 10.2

46. **proposition**
    see Section 2.1.3

47. **qualified**
    (adj). applied to a path identifier (`<path_id>`) it means a path identifier with at least one occurrence of `"::"`. See Section 5.1

48. **reducible to**
    see Section 2.4.2

49. **scalar**
    (noun) a stream of scalar type
    (adj.) applied to a stream it means a scalar stream; applied to a type it means a scalar type; applied to a value it means a value of a scalar type

50. **scalar type**
    any type defined in Section 8.1

51. **scope**
    see Section 2.3

52. **significant bit**
    a significant bit of an integer encoded in two's complement is a bit which cannot be removed from the encoding without changing the encoded value; leading 0s and 1s are not significant, meaning that the numbers `000` and `0` both encode 0, `001` and `01` both encode +1, and `111` and `1` both encode −1

53. **static flag**
    see Section 2.1.5

54. **stream**
    a stream of values of a certain type, see Section 10

55. **temporal function**
    a mapping from streams to streams

56. **temporal operator**
    see *temporal function*

57. **type**
    a tool for the classification of streams which can be understood simply as a set of values, see Section 8

58. **undefined variable**
    see *free variable*

59. **unsized copy**
    see Section 8.4.2

60. **user namespace**
    see Section 5; not to be confused with *namespace*

61. **value**
    a mathematical object that is the element or member of a type, for example the Boolean value "true", the integer value "1234", or the enum value "`blue`"; composite values (*i.e.* values of composite types) can be represented by n-tuples where n corresponds to the number of components of the composite type, for example "$(\text{true}, 1234)$"; if a value `V` is used in a context where a stream is expected, it is interpreted as a constant stream that takes the value `V` in each time step

62. **variable**
    (often preceded by *stream*) a stream variable, *i.e.* a named stream (that can be referenced if it is *visible*)

63. **visible / visibility**
    refers to variable visibility, see Section 2.3

E Label Index

References

[1] Julien Ordioni, Nicolas Breton, Jean-Louis Colaço, HLL v.2.7 Modelling Language Specification, [Other] STF-16-01805, RATP. 2018. ffhal-01799749f.
URL: `https://hal.archives-ouvertes.fr/hal-01799749`