HLL

Language Definition

Id: HLL-LDD	Date: September 16, 2026
Version: 4.0	Pages: 128

Legal Notice

Main authors

Lars Helander and Olav Bandmann, Prover Technology AB.

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

Version	Date	Reason for change
4.0	*September 16, 2026*	First version released by the HLL Forum.
2.7	*March 13, 2012*	Version 2.7 released by Prover.
1.0	*March 19, 2008*	Initial Version.

Contents

Foreword

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. The aim of this initiative is to oversee the maintenance and development of the HLL language specifications and promote its use.[1] 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.

In 2026, version 4.0 was released by the HLL Forum.

Contributors

The following organizations participated in the development of version 4.0 of HLL:

- Ansys
- Prover
- RATP
- SafeRiver
- Systerel

Credits

The following people are credited for having participated in the development of version 4.0 of this document:

- Olav Bandmann, Prover
- Sarah Benyagoub, RATP
- Nicolas Breton, Systerel
- Nathan Grosshans, RATP

[1] `https://www.hllforum.org`

- Alexandra Halchin, RATP
- Lars Helander, Prover
- Adja Ndeye Sylla, RATP
- Pierre Nigron, RATP
- Laurent Voisin, Systerel

1 Introduction

This document presents the *syntactical* and *semantical* aspects of the HLL[2] modeling language.

HLL is a synchronous, discrete-time and declarative data-flow stream-based language with a large panel of types and operators. It is suitable for modeling discrete-time sequential behaviors and expressing temporal properties of these behaviors.

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 F.

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, and are structured according to the EBNF syntax definition of the language.
  - 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 expressions.
  - Section 11 describes variable declarations.
  - Section 12 describes variable definitions.
  - Section 13 describes constants.
  - Section 14 describes constraints.
  - Section 15 describes proof obligations.
  - Section 16 describes sections.

[2]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 type system of HLL.
- Appendix D lists the language constructs of HLL that are possible sources of the exceptional value **nil**, and the operations that may absorb **nil**.
- Appendix E lists the non-backwards compatible language changes since previously released versions of this document, and the possible future such changes (deprecated language constructs).
- Appendix F is a glossary of words, terms and abbreviations used within this document.
- Appendix G gives an index of (external) labels exported by this document.
- Appendix H provides information on how to contact the maintainers of this document, the HLL Forum.

2 Preliminaries

2.1 Streams

HLL is a language based on the notion of *Streams*. A stream is a function that maps each natural number, referred to as a *time step*, to a specific value. Time steps represent discrete, sequential instants in time, starting from zero and extending indefinitely:

`s` :	s_0	s_1	s_2	s_3	...	s_n	s_{n+1} ...

Streams are used to model (or give an interpretation to) the expressions of HLL and are suitable for representing an evolution over discrete time steps, in a way that mimics the execution cycles of a reactive program with synchronous time.

There is an important distinction between streams and *Expressions*. The expressions of HLL are defined by the nonterminal `<expr>` in Section 10 whereas streams are simply infinite sequences of values of a given type. For any given model, every expression has a stream as its associated interpretation. To relate these concepts to our example above, we can think of `s` as an expression, the s_i as values, and the sequence "$s_0\ s_1\ s_2\ s_3\ \ldots\ s_n\ s_{n+1}\ \ldots$" as a stream.

Every expression has a *type*. Types represent sets of values, and statements like "a stream of type `T`" simply mean that the stream respects `T` in the sense that every value in the stream belongs to the set represented by `T`. If an expression is of type `T`, then, for any model, its associated interpretation is a stream of type `T`.

For example the Boolean type (written `bool` in HLL) represents the set of values {false, true}. If the expression `s` above has the type `bool`, and the stream "$s_0\ s_1\ s_2\ldots$" is it's interpretation in some model, then each s_i belongs to the set {false, true}. HLL supports a large number of different types, for example Boolean, integer, enumeration, structure and array.

Below follows an example of an interpretation (in some model) of a Boolean expression `x` and an integer expression `y`:

`x` :	true	false	true	true	false	true	...
`y` :	-1	10	42	-5	-5	8	...

2.1.1 Operators

In HLL, there are two fundamentally different types of operators (or functions):

1. combinational operators, which are mappings from values to values, and
2. stream operators, which are mappings from streams to streams.

Any stream of combinational operators (all having the same type signature) "$f_0\ f_1\ f_2\ldots$" can be lifted to a stream operator F by pointwise application: $F(s)_n = f_n(s_n)$ (in general there are several arguments). Note that in each time step such a stream operator only has access to the values *in the current time step* of its operands.

In the case when a constant sequence of functions "$f\ f\ f\ldots$" is lifted, we simply say that the combinational operator f is lifted to a stream operator. For example, if `a` and `b` are expressions of integer type with the interpretation:

`a` :	a_0	a_1	a_2	...	a_n	...
`b` :	b_0	b_1	b_2	...	b_n	...

then the combinational addition operator on integers can be lifted to a stream operator used to define the interpretation of the (syntactical) HLL operator +:

`a + b` :	$a_0 + b_0$	$a_1 + b_1$	$a_2 + b_2$	...	$a_n + b_n$	...

Thus, assuming that expressions `a` and `b` are interpreted as the following streams in some model:

`a` :	2	3	5	7	11	13	...
`b` :	1	1	2	3	5	8	...

Then `a + b` is, in this model, interpreted as:

`a + b` :	3	4	7	10	16	21	...

Stream operators that are not the lift of any stream of combinational operators are called *temporal operators*. For such an operator the resulting value in some time step depends on the value of other time steps. In general, a temporal operator is free to access the values of its operands in any time steps.

The interpretation of the (syntactical) `X` operator in HLL (read “next operator”) is an example of a temporal operator which is usually thought of as returning the “next value” of its operand. To be precise however, the operator shifts the entire stream of its operand one step to the left. To continue our last example:

`X(a + b)` :	4	7	10	16	21	...

2.2 Logic of Exceptions

In HLL, many operations such as division by zero, overflow, array indexing out of bounds, and ill-founded recursions give rise to the exceptional value **nil**.

This exceptional value, **nil**, propagates unhindered through most HLL operations; if one of the arguments to the operation is **nil**, then the result of the operation is also **nil**. However, for a few operations[3], **nil** can be absorbed, such as in the if-then-else construct and Boolean operations with absorbing values (and, or, implies). To preserve the commutativity of the Boolean operators, they are defined to be symmetric, meaning **nil** can be absorbed on either side of the operator[4]. This means that "true or **nil**" and "**nil** or true" will both evaluate to "true".

The general rule is that when **nil** is absorbed, replacing **nil** with any value does not change the resulting value of the operation. The precise evaluation rules can be found in Section 10.

[3] See Section D.2 for the list of operations that may absorb **nil**.
[4] This semantics of three-valued logic corresponds to "Kleene's strong logic of indeterminacy" (Kleene's K_3^S).

2.3 Definitions

2.3.1 Types and Values

Definition 1 (types)**.** *Types* in HLL are defined by the non-terminal <type> in Section 8.[5] The set of all HLL types is denoted by $\mathcal{T}$. There are two fundamentally different categories of types: *Scalar* types and *Composite* types. The composite types are built up from other composite types and scalar types. □

Definition 2 (values)**.** Each type T has a set of *Values* associated to it, denoted by $V(T)$. The set of values associated to each type is given in Section 8. The set of all values is defined as $\mathcal{V} = \bigcup_{T\in\mathcal{T}} V(T)$. Values associated to scalar types are called *scalar* and values associated to composite types are called *composite*. Scalar values are atomic, whereas composite values are composed of *Components* (possibly infinitely many), each associated to a type which in turn can be either composite or scalar. □

There are several composite types defined in Section 8, but their associated values are of only two kinds. The set $V(T)$ of values associated to a composite type T is either

- the set of (ordinary[6] mathematical) functions $V(T_0)^{V(T_1)\times\cdots\times V(T_n)}$ for some type T_0 and some *scalar* types $T_1, \ldots, T_n$, or
- the set of tuples $V(T_1) \times \cdots \times V(T_n)$ for some types $T_1, \ldots, T_n$.

Note that the types T_i above are not uniquely determined by $V(T)$; only the sets of values $V(T_i)$ are uniquely determined. Therefore, the three definitions 4, 5, and 6 below do not depend on the choices of T_i, but only on the sets of values $V(T_i)$.

The structure of a value associated to a type follows the structure of the type. A composite value can be viewed as the root of a tree, having its components as children, where the leaves are scalar values and the internal nodes are other composite values. The leaves are called the *Scalar leaves* of the composite value. This notion is extended to scalar values as a degenerate case, by letting the value itself be its single scalar leaf.

Definition 3 (nil)**. nil** is an exceptional polymorphic scalar value used to define the semantics for exceptional behavior such as division by zero, overflow, array indexing out of bounds, and ill-founded recursions (see Section 2.2). □

Next we add **nil** to the values associated to a type to get the full range of values that defined variables can attain.

Definition 4 (type extension)**.** Given a type T we define $V^*(T)$, the *extension* of $V(T)$, by

$$V^*(T) = \begin{cases} V(T) \cup \{\textbf{nil}\} & \text{if } T \text{ is scalar} \\ V^*(T_0)^{V(T_1)\times\cdots\times V(T_n)} & \text{if } V(T) = V(T_0)^{V(T_1)\times\cdots\times V(T_n)} \\ V^*(T_1) \times \cdots \times V^*(T_n) & \text{if } V(T) = V(T_1) \times \cdots \times V(T_n) \end{cases}$$

The set of all extended values is defined as $\mathcal{V}^* = \bigcup_{T\in\mathcal{T}} V^*(T)$. □

[5] Rigorously speaking, the implicit union type in Section 8.4.3, item (UnionSort), needs to be added to make the set of expressable types complete. However, it would also be sufficient to instead just add a new sort for each possible union of sorts that is not already expressible using the given sorts.

[6] We use the notation B^A to denote the set of functions from A to B. In the context of values, we will consider two functions *equal* if they have the same graph. Thus, given $f : A \to B$ and $g : A \to C$, if $f(a) = g(a)$ for all $a \in A$, then $f = g$, even when $B \neq C$.

Thus, the extension is defined by pushing the $*$ down to the scalar leaves where **nil** is then added.

A variable not introduced within a lambda scope (Section 10.2)[7] or quantifier scope (Section 10.7.10) is considered *free* if it is undefined (Section 12), or *partially free* if it has an undefined initial value (Section 12).

The set of allowed values for free variables of a given type is more restricted than that for defined variables, but they are similarly defined. The only difference is that **nil** is only added to the empty scalar leaves.

Definition 5 (free values)**.** Given a type T, $V^+(T)$ is the non-empty set of values defining the domain for free variables of type T.

$$V^+(T) = \begin{cases} V(T) & \text{if } T \text{ is scalar and } V(T) \neq \emptyset \\ \{\textbf{nil}\} & \text{if } T \text{ is scalar and } V(T) = \emptyset \\ V^+(T_0)^{V(T_1)\times\cdots\times V(T_n)} & \text{if } V(T) = V(T_0)^{V(T_1)\times\cdots\times V(T_n)} \\ V^+(T_1)\times\cdots\times V^+(T_n) & \text{if } V(T) = V(T_1)\times\cdots\times V(T_n) \end{cases}$$

□

Because the way to access the components of a composite value differs, depending on the kind of its composite type, we will also use a more abstract and uniform way to access the individual components.

Definition 6 (component projections)**.** The components of a composite type T are specified by a set of functions $\mathcal{P}(T)$ associated to T called the *Component Projections* of T. There is precisely one component projection for each component of T.

The common domain of the component projections in $\mathcal{P}(T)$ is denoted by $\mathcal{V}_T^*$. If $p \in \mathcal{P}(T)$, then we let $p(T)$ denote the type of the component and $p\colon \mathcal{V}_T^* \to \mathcal{V}^*$ be the function which gives the value of the component, in particular $p(V^*(T)) \subseteq V^*(p(T))$.

The set $\mathcal{P}(T)$ is defined as:

- If $V(T) = V(T_0)^{V(T_1)\times\cdots\times V(T_n)}$, then $\mathcal{V}_T^* = \bigcup_{T_0'\in\mathcal{T}} V^*(T_0')^{V(T_1)\times\cdots\times V(T_n)}$ and

  $$\mathcal{P}(T) = \{p_{(a_1,\ldots,a_n)} \mid (a_1,\ldots,a_n) \in V(T_1)\times\cdots\times V(T_n)\},$$

  where $p_{(a_1,\ldots,a_n)}(v) = v(a_1,\ldots,a_n)$.
- If $V(T) = V(T_1)\times\cdots\times V(T_n)$, then $\mathcal{V}_T^* = \bigcup_{T_1',\ldots,T_n'\in\mathcal{T}} V^*(T_1')\times\cdots\times V^*(T_n')$ and

  $$\mathcal{P}(T) = \{p_1,\ldots,p_n\},$$

  where $p_i(v_1,\ldots,v_n) = v_i$.

□

An immediate but useful property is that if $V^*(T) \cap V^*(T') \neq \emptyset$, then $\mathcal{P}(T) = \mathcal{P}(T')$.

From Definition 6 it follows that, given an arbitrary function $\varphi\colon \mathcal{P}(T) \to \mathcal{V}^*$ that satisfies $\varphi(p) \in V^*(p(T))$, there exists a unique $v \in V^*(T)$ such that $p(v) = \varphi(p)$ for all $p \in \mathcal{P}(T)$.

[7]Note that lambda scopes can also be introduced by case expressions (CaseExpr) and array/function definitions (DefArrayFunction).

I.e., *for any set of prescribed projection values (φ), there exists a unique value (v) that has those projection values*.

Note that since the component projections structure all the values in $V^*(T)$ recursively in a uniform manner (according to the structure of the type T), all values in $V^*(T)$ will have the same tree structure; only the content of the leaves may differ.

Also note that by Definition 4, the values in $V(T)$ and $V^*(T)$ will have the same tree structure. Only the content of the leaves may differ (by allowing **nil** or not).

Values associated to types extended with **nil** have a natural order with respect to how much information they contain.

Definition 7 (information order)**.** The partial order $\preceq$ is defined on the set $\mathcal{V}^*$ as follows: First define the partial order $\preceq_T$ on the set $V^*(T)$ for $T \in \mathcal{T}$. Let $v_1, v_2 \in V^*(T)$.

- If T is a scalar type, then $v_1 \preceq_T v_2$ if and only if $v_1 = \mathbf{nil}$ or $v_1 = v_2$.
- If T is a composite type, then $v_1 \preceq_T v_2$ if and only if $p(v_1) \preceq_{p(T)} p(v_2)$ for every $p \in \mathcal{P}(T)$.

Now we define $\preceq$. If $v_1, v_2 \in \mathcal{V}^*$, then $v_1 \preceq v_2$ if and only if there exists[8] $T \in \mathcal{T}$ such that $v_1 \preceq_T v_2$. □

It follows by induction that

$$\text{if } a \preceq_T b \text{ and } b \in V^*(T'), \text{ then } a \preceq_{T'} b \qquad (*)$$

which can be used to show that $\preceq$ is a partial order (and other properties).

Informally, $v_1 \preceq v_2$ means that v_1 and v_2 are equal except that some **nil**-valued scalar leaves in v_1 may have been replaced by non-**nil** scalar values in v_2, which is interpreted to mean that v_2 contains (at least) all the information that v_1 contains.

The set $V^*(T)$ has a member with minimal information (i.e. the minimum in $V^*(T)$ with respect to $\preceq$), the value that has all its scalar leaves set to **nil**.

Definition 8 (minimal element)**.** We define the value $\mathbf{nil}^\wedge(T)$ to be the unique value in $V^*(T)$ that satisfies:

$$\begin{cases} \mathbf{nil}^\wedge(T) = \mathbf{nil} & \text{if } T \text{ is scalar,} \\ p(\mathbf{nil}^\wedge(T)) = \mathbf{nil}^\wedge(p(T)) \quad \text{for all } p \in \mathcal{P}(T) & \text{if } T \text{ is composite.} \end{cases}$$

□

All the information in the scalar leaves of $\mathbf{nil}^\wedge(T)$ is absent; only the tree structure, the skeleton that all elements in $V^*(T)$ share, remains. It is therefore natural to consider that the minimal value represents the structure of the elements in $V^*(T)$.

Definition 9 (value structure)**.** The *Structure* of a value $v \in \mathcal{V}^*$ is defined as the value $\mathbf{nil}^\wedge(T)$ when $v \in V^*(T)$.

It follows from $(*)$ that all the sets $V^*(T)$ are closed downwards (w.r.t. the order $\preceq$ on $\mathcal{V}^*$). This implies that if $V^*(T) \cap V^*(T') \neq \emptyset$, then $\mathbf{nil}^\wedge(T) = \mathbf{nil}^\wedge(T')$. Hence the definition of the structure of a value does not depend on the choice of T. □

[8]Note that if $T, T' \in \mathcal{T}$ and $V^*(T) \cap V^*(T') \neq \emptyset$, then $\preceq_T$ and $\preceq_{T'}$ will coincide on $V^*(T) \cap V^*(T')$. Hence the choice of $T \in \mathcal{T}$ at the end of Definition 7 doesn't matter: $v_1 \preceq v_2$ if and only if $v_1 \preceq_T v_2$ for any $T \in \mathcal{T}$ such that $v_1, v_2 \in V^*(T)$.

Definition 10 (domain check)**.** Given a type T and a value $v \in \mathcal{V}^*$ having the same structure as the values in $V^*(T)$, the domain check $D(T, v)$ is defined to be the unique value in $V^*(T)$ that satisfies:

$$\begin{cases} D(T,v) = v & \text{if } T \text{ is scalar and } v \in V(T) \\ D(T,v) = \textbf{nil} & \text{if } T \text{ is scalar and } v \notin V(T) \\ p(D(T,v)) = D(p(T), p(v)) \quad \text{for all } p \in \mathcal{P}(T) & \text{if } T \text{ is composite.} \end{cases}$$

□

In other words, $D(T, v)$ is the greatest value $w \in V^*(T)$ such that $w \preceq v$.

2.3.2 Models

In this section we give the semantic model for the set S of HLL expressions (which are defined by the non-terminal <expr> in Section 10). We begin by defining the concept of "approximate model" which is used, together with a limiting procedure, to handle the semantics of recursion.

Definition 11 (approximate model)**.** An *Approximate Model* M_i, at *meta step* $i \in \mathbb{N}$, for the set S of HLL expressions is a binary function $M_i : S \times \mathbb{N} \to \mathcal{V}^*$, where the second argument denotes time steps (with 0 representing the initial time step). Approximate models are required to respect the type T of their argument s in the sense that $M_i(s,k) \in V^*(T)$ for all $k, i \in \mathbb{N}$.

Except for completely or partially free variables, the precise rules defining the approximate models are given throughout the document. In general, for any meta step i and any expression s, which is not a named expression[9], the definition of the stream $k \mapsto M_i(s,k)$ is given in terms of M_i itself (Section 10), and if s is a named expression and $i > 0$, then $k \mapsto M_i(s,k)$ is given in terms of M_{i-1} (Section 12).

For completely or partially free variables (such as inputs or initial inputs), the rules defining the models are as follows: for each variable s of type T and for each time step k where s is undefined and for all $i \geq 0$:

$$M_0(s,k) \in V^+(T)$$
$$M_{i+1}(s,k) = M_i(s,k)$$

□

Only defined variables (see Section 12) can introduce recursion in HLL. In any time step k, for an expression E that contains no defined variables as subexpressions, the rules in Section 10 will imply that for all i, $M_{i+1}(\mathtt{E},k) = M_i(\mathtt{E},k)$ which will then trivially be the limiting value of the sequence $(M_i(\mathtt{E},k))_{i=0}^{\infty}$ in the resulting model.

For an expression E containing defined variables, at any fixed time step k, the sequence $(M_i(\mathtt{E},k))_{i=0}^{\infty}$ is typically not constant but will converge to a limit — becoming eventually constant in the scalar case and converging leaf-wise to a limit in the composite case.

Definition 12 (limit)**.** If T is a scalar type, we say that a sequence $(v_n)_{n=0}^{\infty}$ of values in $V^*(T)$ *converges* whenever there exist $m \in \mathbb{N}$ and $v \in V^*(T)$ such that $v_n = v$ for all $n \geq m$. The value v is called the limit of this sequence and written $\lim_{n\to\infty} v_n$.

If T is a composite type, we say that a sequence $(v_n)_{n=0}^{\infty}$ of values in $V^*(T)$ *converges* whenever the sequences $(p(v_n))_{n=0}^{\infty}$ converge for all $p \in \mathcal{P}(T)$. The limit of the sequence $(v_n)_{n=0}^{\infty}$, written $\lim_{n\to\infty} v_n$, is then the unique value $v \in V^*(T)$ satisfying $p(v) = \lim_{n\to\infty} p(v_n)$ for all $p \in \mathcal{P}(T)$. □

The difference between the approximate models M_i and the resulting limit is that for approximate models the defined variables are initialized by setting their scalar leaves to **nil** and then, for each meta step, assigned to the value of their corresponding right hand side in the previous meta step (see Section 12). For the resulting limit, on the other hand, all left hand sides of definitions will have the same value as their corresponding right hand side.

[9] A *named expression* is an atomic expression that refers to a variable (Section 10.7.3).

Definition 13 (model)**.** Given a sequence of approximate models $(M_i(\mathrm{E}, k))_{i=0}^{\infty}$, the *Model* M for the set S of HLL expressions is the binary function $M : S \times \mathbb{N} \to \mathcal{V}^*$ defined as the limit of the approximate models: for any $s \in S$ and $k \in \mathbb{N}$

$$M(s, k) = \lim_{i \to \infty} M_i(s, k)$$

Note that the sequence $(M_i(s, k))_{i=0}^{\infty}$ is monotonically increasing, i.e. $M_i(s, k) \preceq M_{i+1}(s, k)$ for all i. This is a consequence of the fact that all constructors of expressions (refer to Section 10) satisfy the semantic requirement of being monotonically increasing functions (w.r.t. $\preceq$) of the values of their parts (see also Section 2.2), a property that extends to all HLL expressions since compositions of monotone functions are monotone.

It follows from the definition of the partial order $\preceq$ (Definition 7) and the definition of convergence (Definition 12), that monotonically increasing sequences are convergent. Hence the limit in the definition of $M(s, k)$ is well defined. □

The model M, derived from a sequence of approximate models $(M_i(s, k))_{i=0}^{\infty}$, satisfies all the properties that the M_i do, with the clarification that, in Section 12, all the equalities defining M_{i+1} in terms of M_i remain valid when both M_{i+1} and M_i are replaced with M; in other words, M adheres to the HLL definitions.

This conclusion follows from the observation that all functions used to define the properties of the approximate models — component projections, the domain check function, and the constructions in Section 10 — are continuous (and therefore commute with the limit operator) and independent of the meta step.

2.3.3 Propositions

Definition 14 (proposition)**.** Given a Boolean expression 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 true in all time steps

□

2.3.4 Consequences

Definition 15 (consequence)**.** A proposition q is a *Consequence* of a set of propositions P *iff* for all models M, the following holds:

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

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

1. $M \models_D \mathbb{I}\, s$ *iff* $M(s, 0) \in D$.
2. $M \models_D \Box s$ *iff* $M(s, k) \in D$ for all k.

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 $D = \{\text{true}\}$. □

Note that if the expressions 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.

Also note 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 2.3.4 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).

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" [10], 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".

[10]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.

2.3.5 Static Flag

Definition 16 (static flag)**.** Given an HLL text, the function $\mathcal{SF} : S \to \{0, 1, 2\}$ returns the *Static Flag* of an expression. An expression s is called *Static* if, in every model M, it takes the same value in each time step, *i.e.* $M(s, k) = M(s, 0)$ for all k. The static flag for each type of 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 expressions.

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

2.4 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. expressions,
2. types,
3. user namespaces, and
4. struct components.

Each of the first three types of entities resides in its own namespace. For struct components, there is a separate namespace for each struct type.

A namespace can be further subdivided into *Scopes* which form the nodes of a tree with a unique root node. This unique root node is called the *Global top-level scope*. 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.

At each point in an HLL text, the path from the root node of the scope tree down to the current scope forms a stack $[S_1\ S_2\ \ldots\ S_n]$ where S_1 is the top-level scope of the root node and S_n is the bottom-most scope (the current scope).

Assume an element S_i with $i \in [1, n]$ of such a stack. We will call a scope S_j with $j < i$ an *Ascendant* scope of S_i, and we will call a scope S_k with $k > i$ a *Descendant* scope of S_i. 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[11] 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 E_1 may have the same name as another visible entity E_2 that exists in an ascendant scope (one higher up in the stack). In such a case we say that E_1 *Hides* E_2 in all scopes in which 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 belong to.

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.

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

The declaration or definition of, or in rare cases the reference to, a named entity will cause the entity to exist in the scope in which the declaration, definition or reference occurred (see sections 11, 12, and 10.7.3, respectively). This means that the entity is visible and can be referenced everywhere in the scope, regardless of (and even before) the position of the declaration, definition or reference that caused the existence of the entity within the scope.

2.5 Notation

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

2.5.1 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:`.

2.5.2 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).

We will use standard mathematical notation as much as possible. For example, given sets A and B we use the following standard set notation: $x \in A$ (set membership: x is a member of A), $A \cup B$ (set union) and $|A|$ (set cardinality). Given two integers x and y, we use the range notation $[x, y]$ to denote the set of all integers in the interval between x and y plus x and y themselves, except if $y < x$ in which case $[x, y]$ denotes the empty set, which we write as $\emptyset$. Given a (mathematical) function $f : A \to B$, we use the standard notation $\mathrm{Im}(f)$ to denote the image of f, which is the set $\{f(x) \mid x \in A\}$.

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 construct C_1 and a more basic language construct C_2, we will use the following two relations to relate C_1 to C_2:

Relation	*Type*	*Meaning*
C_1 *is equivalent to* C_2	equivalence relation (reflexive, symmetric transitive)	Either of C_1 and C_2 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[12].
C_1 *is reducible to* C_2	preorder (reflexive, transitive)	C_2 can be used in place of C_1, without change of meaning or correctness[13], anywhere in an HLL text in which all subexpressions have been explicitly grouped.

To avoid redundancy, whenever we say that C_1 is equivalent or reducible to C_2 this means that all relevant restrictions that apply to C_2 (directly or indirectly), also apply to C_1, even if this is not explicitly said. Furthermore, if C_1 and C_2 are expressions they will have the same type even if this is not explicitly said.

As an example, the expression E_1 `!=` E_2 is defined as equivalent to `~`(E_1 `=` E_2). 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 C_1 has additional restrictions compared to another construct C_2, but they are otherwise equivalent, we will typically say that C_1 is reducible to C_2. As an example, the expression E_1 `<->` E_2 is reducible to E_1 `=` E_2 since the type `bool` required of the operands of `<->` satisfies the restrictions on =, but not the other way around.

Note that the purpose of the relation "C_1 is reducible to C_2" is to enable us to define C_1, in terms of (the already defined) C_2, by rewriting C_1 to C_2. When rewriting, the relation between C_1 and C_2 is usually symmetric (i.e., "C_1 is equivalent to C_2"), even if the rewriting goes from left to right, but in some cases it's not symmetric (only reducible). In these cases, either the semantics might change if C_2 is replaced with C_1, or, even though the expression C_2 is legal, the expression C_1 might not be.

[12]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.

[13]Provided any additional restriction on C_1 (compared to C_2) is satisfied.

2.6 Document Structure

This document has been structured around the syntax of HLL[14]. 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.

2.6.1 Language Construct Example

Here we may give a short informal introduction with a few examples as a help for the reader. Below, in the gray 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 G.

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.

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.

[14]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 are not allowed in an HLL text:

Character		*ASCII value*
`'\0'`	(null character)	0

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

Characters		*ASCII values*
`'\t'`	(horizontal tab)	9
`'\n'`	(new line)	10
`'\r\n'`	(carriage return followed by new line)	13, 10
`' '`	(white space)	32

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

3.1 Comments

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 sequence `"\n"` or `"\r\n"`.
2. (CommentSlashStar) Characters present between `"/*"` and `"*/"` are ignored. 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 */
```

Quoted identifiers (see Section 4) also interfere with comments. There are two kinds of quoted identifiers: one is delimited by the single quote character (`"'"`, ASCII 39) and the other is delimited by the double quote character (`""`, ASCII 34). Together with the tokens `"//"`, `"/*"` and `"*/"`, single and double quotes are also considered in the order they appear in the file. Inside a comment, single and double quotes are ignored, and inside a quoted identifier of one kind, quotes of the other kind and all comment tokens are treated as ordinary sequences of characters being part of the identifier.

3.2 Pragmas

(Pragma) All the characters after an "@" are interpreted as the text of a pragma up to, and including, the end of the line character sequence "\n" or "\r\n", except if the "@" is inside a comment.

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

Identifiers in HLL come in three different flavors: unquoted, single-quoted and double-quoted. The unquoted syntax allows only certain characters, whereas the quoted one is more permissive and allows any character except newline (`'\n'`) between the quotes. The quote characters are significant, meaning that the three identifiers `A`, `'A'` and `"A"` are all distinct.

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 a type, a variable, 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* (see (NamedExprImplicitDecl) in Section 10.7.3).
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

User namespaces allow the user to compartmentalize an HLL text with the goal of avoiding variable name clashes between the different compartments. Each user namespace can contain an HLL text in its entirety (defined by the nonterminal <HLL> in Section 16) which means that user namespaces can be nested. Referencing entities (such as variables or named types) of a user namespace from the outside is done using path identifiers, which are defined in Section 5.1.

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 variables, 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> refer to the same user namespace. This means that $\texttt{N \{ HLL}_1\texttt{ \}\ HLL' N \{ HLL}_2\texttt{ \}}$ (where HLL′ represents anything in the text with balanced curly brackets that may come in between the two <namespace>) is equivalent to $\texttt{N \{ HLL}_1\ \texttt{HLL}_2\texttt{ \}\ HLL'}$ (also equivalent to $\texttt{HLL' N \{ HLL}_1\ \texttt{HLL}_2\texttt{ \}}$).

Restrictions

(empty)

5.1 Path Identifiers

Path identifiers allow referencing variables or named types 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
```

Path identifiers (nonterminal `<path_id>`) are normally used to *reference* existing named entities such as variables and types, whereas ordinary identifiers (nonterminal `<id>`) are used to *declare* such entities (causing them to exist in the scope in which they were declared).

The semantics or correctness of a path identifier that does not refer to any existing entity depends on the context in which the path identifier is used. The general rule is that such a path identifier is incorrect (see the restriction (PathIdNoImplicitDecl) below), and the only exception to this rule is when the path identifier is just an ordinary identifier, which implicitly declares the nonexisting variable it refers to, causing it to exist (see (NamedExprImplicitDecl) of Section 10.7.3).

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>`).

   A `<path_id>` with a prefix is called qualified, and one without a prefix is called unqualified.

2. (PathAbsolute) An `<absolute_path>` :: $\texttt{NS}_1$ :: $\texttt{NS}_2$ :: $\cdots$ :: $\texttt{NS}_\texttt{n}$ :: designates a user namespace $\texttt{NS}_\texttt{n}$ that is nested inside user namespaces $\texttt{NS}_1 \ldots \texttt{NS}_{\texttt{n}-1}$ where $\texttt{NS}_1$ is defined on the global top-level.

3. (PathRelative) A `<relative_path>` $\texttt{NS}_1$ :: $\texttt{NS}_2$ :: $\cdots$ :: $\texttt{NS}_\texttt{n}$ :: designates a user namespace $\texttt{NS}_\texttt{n}$ that is nested inside user namespaces $\texttt{NS}_1 \ldots \texttt{NS}_{\texttt{n}-1}$ where $\texttt{NS}_1$ is selected by applying the following rules in order:

   (a) If the `<relative_path>` occurs on global top-level, then the user namespace $\texttt{NS}_1$ defined on the global top-level is selected. If there is none then the `<relative_path>` is incorrect.

   (b) Otherwise, 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.

   (c) Otherwise (and this case is ***deprecated*** and may be removed in future versions of this document), the user namespace $\texttt{NS}_1$ defined on the global top-level is selected. If there is none then the `<relative_path>` is incorrect.

4. (PathIdQualified) A qualified `<path_id> PATH::ID` refers to the entity named `ID` in the top-level scope of the user namespace designated by `PATH`.

5. (PathIdUnqualified) An unqualified `<path_id> ID` refers to the first entity named `ID` encountered when searching upwards in the stack of nested scopes having the scope in which the `<path_id>` occurs bottom-most. The uppermost scope of the stack is the global top-level scope, and is the one searched last.

Restrictions

1. (PathIdNoImplicitDecl) A qualified `<path_id>` (*i.e.* a `<path_id>` with at least one occurrence of `::`) shall refer to an existing entity declared and/or defined in the user namespace designated by the path (no implicit declaration by reference).

6 Syntactic Lists

The syntactic lists defined in this section are just auxiliary grammatical stuff used to define other language constructs in subsequent parts of this document, and have no particular semantics in themselves. They are *not* data structures which an HLL user instead would create using the composite types of Section 8.2.

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:

```
Declarations:
 bool A[8];          // A is an array of 8 bool indexed from 0 to 7
 bool B[4, 3];       // B is a 2-dimensional array of 4*3 bool
 bool f(int);        // f is a function from int to bool
 bool g(int, bool); // g is a Boolean function with two arguments:
                     //   one of type int, the other of type bool
Outputs:
 A[5];
 B[3, 2];            // The last element of B (array-indexing is 0-based)
 f(-47);
 g(12, True);
```

The array and function types themselves are; for the declared `A`, `B`, `f`, and `g` above; `bool^(8)`, `bool^(4, 3)`, `(int -> bool)`, and `(int * bool -> bool)`, respectively (see Sections 8.2.4 and 8.2.3). The same declarations could also be achieved by using these types directly:

```
Declarations:
 bool^(8) A;
 bool^(4, 3) B;
 (int -> bool) f;
 (int*bool -> bool) g;
```

Array and function declarators can be arbitrarily iterated:

```
Declarations:
 bool h[3](bool); // h is an array of 3 functions from bool to bool
 bool C[5][8];    // C is an array of 5 bool arrays of length 8
```

The function `calc_type` on the next page computes the type from an array or function declaration that uses declarators.

Declarators can also be used in type definitions (see Section 8), as shown in the example below, which is an equivalent formulation of the declaration of `C` above:

```
Types:
 bool T[5][8]; // T is the type bool^(8)^(5)
Declarations:
 T C;
```

Syntax

(DeclaratorSyntax)

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

Semantics

1. (Declarator) A *Declarator* (`<declarator>`) consists of an identifier `<id>` and optional declarator suffixes (`<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 [E_1, E_2, ..., E_n] :
      return calc_type(T~(E_1, E_2, ..., E_n), D \ L).
    else if L is (T_1, T_2, ..., T_n) :
      return calc_type((T_1 * T_2 * ··· * T_n -> T), D \ L).
    else (D is empty) :
      return T.
}
```

Please note that the array type notation `T~(E_1, E_2, ..., E_n)` is defined in Section 8.2.4 and the function type notation `(T_1 * T_2 * ··· * T_n -> T)` in Section 8.2.3.

Restrictions

1. (DeclArrayDimInteger) Each E_i of a declarator suffix $[E_1, E_2, \ldots, E_n]$ shall be of integer type (see Section 8.1.2).

2. (DeclArrayDimConstant) Each E_i of a declarator suffix $[E_1, E_2, \ldots, E_n]$ shall satisfy $\mathcal{SF}(E_i) = 2$.

3. (DeclArrayDimNotNil) For each E_i of a declarator suffix $[E_1, E_2, \ldots, E_n]$ there shall exist $j \geq 0$ satisfying $M_j(E_i, 0) \neq$ **nil**.

4. (DeclFunctionParamScalar) Each T_i of a declarator suffix $(T_1, T_2, \ldots, T_n)$ shall be scalar (see Section 8.1).

8 Types

In HLL, every type T represents a set of values $V(T)$, and every value v belongs to $V(T)$ for *at least* one type T. We say that v *is of type* T whenever $v \in V(T)$.[15]

As an example, the Boolean type (written `bool`) represents the set of values $V(\texttt{bool}) = \{\text{false}, \text{true}\}$. Types are assigned to variables 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 expressions 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).

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

Types are classified as either scalar or composite. The scalar types are atomic in nature whereas the composite types are composed of other types, which are called components of the composite type. The scalar types usually have an order defined on their values and the composite types usually have an order defined on their components. There are a few exceptions however.

Syntax

(TypeSyntax)

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

<ordinary_type_def>  ::= <type> <declarator>
                           {"," <declarator>}

<type_def>   ::= <ordinary_type_def> ";"
               | <enum_def> ";"
               | <sort_def> ";"
```

Semantics

1. (Type) *Types* in HLL are defined by the non-terminal `<type>`. Each type `T` represents a (possibly empty[16], possibly infinite) set of values $V(\texttt{T})$.

[15] Note that an HLL expression is of a unique type (by construction), whereas an HLL value can be of several types.

[16] The exceptional value **nil** does not belong to any proper type.

2. (TypeDef) Both <type_def> and <ordinary_type_def> are type definitions (or definitions of named types) that associate 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. A <type_def> also includes definitions of enumerated types (please refer to (EnumDef)) and contributions to the definitions of sort types (please refer to (SortDef)).

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.

5. (InlineMultTypeDef) A single type definition <type> $D_1, D_2, \ldots, D_n$; where each D_i is a <declarator> is equivalent to n type definitions <type> D_1; <type> D_2; ...<type> D_n;.

6. (TypeCompatibility) The *Compatibility* relation $C(T_1, T_2)$ 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* $C(T_1, T_2)$ = true. In Section 10.4 we will extend this relation to include domains (<domain>).

7. (TypeAssignability) The *Assignability* relation $A(T_1, T_2)$ between types is a preorder (reflexive and transitive) defined in the remainder of the document. A type T_1 is said to be *Assignable to* another type T_2 *iff* $A(T_1, T_2)$ = true.

Restrictions

1. (TypeDefUnicity) In any given scope of the namespace of types, a named type is either defined once by an <ordinary_type_def> or an <enum_def>, or by one or several <sort_def>s.

2. (TypeDefWellFounded) A named type may not be defined in terms of itself, either directly or indirectly.

Related Notation

1. A type is said to be *Finite-dimensional iff* it is either scalar, or it is composite with a finite number of scalar components (either directly or indirectly via other composite components).

8.1 Scalar Types

Scalar Types are the subset of HLL types that represent sets of scalar (or atomic) values. The explicit scalar types are the Boolean, integer, enum and sort types. There are also implicit types that are counted among the scalar types, see for example the sort union type defined by (UnionSort) in Section 8.4.3.

8.1.1 Boolean Type

Syntax

(BoolSyntax)

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

Semantics

1. (BoolValues) The type `bool` represents the set of values {false, true}.
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`). An important purpose of finite integer types is 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. All 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> ::= <path_id>
              | <int_literal>
<range>     ::= "[" <expr> "," <expr> "]"
```

Forward References

1. `<expr>` is defined in Section 10.
2. `<int_literal>` is defined in Section 10.7.2.

Semantics

1. (IntValues) The type `int` represents the set of all integers $\mathbb{Z}$.
2. (IntSignedValues) The type `int signed` $\mathtt{E}_1$ represents the set of integers in the interval $[-2^{\mathtt{E}_1-1}, 2^{\mathtt{E}_1-1}-1]$.
3. (IntUnsignedValues) The type `int unsigned` $\mathtt{E}_2$ represents the set of integers in the interval $[0, 2^{\mathtt{E}_2}-1]$.
4. (IntRangeValues) The type `int` $[\mathtt{E}_3, \mathtt{E}_4]$ represents the set of integers in the interval $[\mathtt{E}_3, \mathtt{E}_4]$ (which may be empty if $\mathtt{E}_4$ is less than $\mathtt{E}_3$).
5. (IntCompatibility) All integer types are compatible with each other.
6. (IntAssignability) All integer types are assignable to each other.
7. (IntValueOrder) `i` $<$ `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) $\mathcal{SF}(\mathtt{E}_\mathtt{i}) = 2$ for $\mathtt{i} \in [1, 4]$.

3. (SignedBitsPositive) $\mathtt{E}_1 > 0$.
4. (UnsignedBitsNonNegative) $\mathtt{E}_2 \geq 0$.
5. (IntSizeNotNil) For each $\mathtt{i} \in [1, 4]$ there shall exist $j \geq 0$ such that $M_j(\mathtt{E}_\mathtt{i}, 0) \neq$ **nil**.

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 Enumerated Types

Enumerated (or enum) types in HLL are similar to implementations in other programming languages. Enum values may not be shared between different enum types. Enum types may not be inlined in a variable declaration but have to be defined using a type definition. The following type definition defines an enum type `Color` with the three values (or members) `red`, `green` and `blue`.

```
Types: enum {red, green, blue} Color;
```

Syntax

(EnumSyntax)

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

Semantics

1. (EnumDef) The enum type defined as `enum` $\{\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n\}$ represents the set of values $\{\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n\}$. An `<enum_def>` associates the name given by the `<id>` with the type, as defined in (TypeDef).
2. (EnumValueSpace) The values of an enum type reside in the namespace of expressions.
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 expressions.

8.1.4 Sort Types

Sort types can be seen as an extension of enumerated types that permits to capture the concept of *inheritance* in an object-oriented paradigm. This is done by allowing the creation of hierarchies of sort types, implemented by inclusion of the set of values of one sort type into another's.

As an example, we could classify the types of HLL using sorts in the following way:

```
Types:
 sort HLLTypes;
 sort ScalarTypes, CompositeTypes, ImplicitTypes < HLLTypes;
 sort {Named} < HLLTypes;
 sort {Boolean, Enum, Sort} < ScalarTypes;
 sort IntegerTypes < ScalarTypes;
 sort {Integer, IntegerRange, IntegerImplementation} < IntegerTypes;
 sort {Tuple, Struct, Function, Array} < CompositeTypes;
 sort {Collection, Unsized, Union} < ImplicitTypes;
```

Sorts are defined by one or more contributions, as illustrated in the example above. A contribution is either a list of other sort types (a "subtype contribution"), or a set of values (a "value contribution"). A subtype contribution adds all values of the subsorts to the sort being contributed to (the one on the right hand side of the < character), including those values which the subsorts themselves received from their own subsorts.

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>` declares the name `<id>` of the sort type (as defined in (TypeDef)) and adds a contribution to its definition (an empty contribution if `<sort_contrib>` is not present).
2. (SortContrib) A sort type is defined by one or more contributions, all within the same scope. The values of the sort type is the union of the values of its contributions.
3. (SortValueSpace) The values of a sort type reside in the namespace of expressions.
4. (SortSubTypeContrib) `sort` $S_1, S_2, \ldots, S_k$ < S denotes the contribution of sort types $S_1, S_2, \ldots, S_k$ to sort type S, and the inclusion of their values into S (*i.e.* $V(S_1) \cup V(S_2) \cup \cdots \cup V(S_k) \subseteq V(S)$).

5. (SortValueContrib) `sort` $\{\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n\} < \mathtt{S}$ 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.* $\{\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n\} \subseteq V(\mathtt{S})$).
6. (SortCompatibility) All sort types are compatible with each other.
7. (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 $\mathtt{T}_2$.
8. (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 expressions.
2. (SortSubTypes) $\mathtt{S}_i$ for $\mathtt{i} \in [1, \mathtt{k}]$ of (SortSubTypeContrib) shall refer to sort types.
3. (SortDefWellFounded) A sort may not contribute to its own definition, either directly or indirectly.

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, an expression of composite type is made up by components. The explicit composite types are tuples, structs, arrays, and functions. There are also implicit types that are counted among the composite types, see for example Section 8.4.1.

8.2.1 Tuple Types

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

Syntax

(TupleSyntax)

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

Semantics

1. (TupleType) A tuple is a composition of ordered unnamed components.
2. (TupleCompOrder) The components of a tuple type are ordered according to the order they appear in the text.
3. (TupleValues) The tuple type $\texttt{tuple}\ \{\texttt{T}_1, \texttt{T}_2, \ldots, \texttt{T}_\texttt{n}\}$ represents the set of values given by $V(\texttt{tuple}\ \{\texttt{T}_1, \texttt{T}_2, \ldots, \texttt{T}_\texttt{n}\}) = V(\texttt{T}_1) \times V(\texttt{T}_2) \times \cdots \times V(\texttt{T}_\texttt{n})$.
4. (TupleCompatibility) A tuple type $\texttt{T}_1$ is compatible with another tuple type $\texttt{T}_2$ *iff* they have the same number of components and each component of $\texttt{T}_1$ is compatible with its corresponding component in $\texttt{T}_2$.
5. (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 $\texttt{tuple}\ \{\texttt{T}_1, \texttt{T}_2, \ldots, \texttt{T}_\texttt{n}\}$ and an integer literal K with $0 \leq \texttt{K} < \texttt{n}$, we will write $\texttt{V}_\texttt{T}@\texttt{K}$ 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, distinct from the HLL tuple accessor (.K) that operates on tuple expressions.

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.
2. (StructCompOrder) The components of a struct are ordered according to the order they appear in the text.
3. (StructValues) The type `struct` $\{M_1 : T_1, M_2 : T_2, \ldots, M_n : T_n\}$ represents the same set of values as the corresponding tuple type `tuple` $\{T_1, T_2, \ldots, T_n\}$ (see (TupleValues)).
4. (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.
5. (StructCompatibility) A struct type T_1 is compatible with another struct type T_2 *iff* they have the same number of components and each component of T_1 has the same name as, and is compatible with, its corresponding[17] component in T_2.
6. (StructAssignability) A struct type T_1 is assignable to another struct type T_2 *iff* they have the same number of components and each component of T_1 has the same name as, and is assignable to, its corresponding component in T_2.

Restrictions

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

Related Notation

1. Given a value V_S of struct type `struct` $\{M_1 : T_1, M_2 : T_2, \ldots, M_n : T_n\}$ and an identifier M_i with $1 \leq i \leq n$, we will write $V_S@M_i$ to denote the component of V_S of type T_i with the name M_i. Note that @ is an operation on struct values, distinct from the HLL struct accessor (`.M`) that operates on struct expressions.

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

8.2.3 Function Types

A function type represents a set of function values. In this document, the term *function value* will refer to what we call a (combinational) function, rather than the value of a function when applied to an argument (a more standard use of the term).

For example, the function type `(int [0, 2] -> bool)` has three components (of type `bool`) corresponding to the arguments 0, 1 and 2. The type represents the $2^3 = 8$ function values in the set $V(\texttt{bool})^{V(\texttt{int[0,2]})} = \{\text{true}, \text{false}\}^{\{0,1,2\}}$, of which for example one function value is the function that maps $0 \mapsto$ false, $1 \mapsto$ true, and $2 \mapsto$ true.

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 $(\texttt{T}_1 * \cdots * \texttt{T}_\texttt{n}\ \texttt{->}\ \texttt{T})$ consists of parameter types $\texttt{T}_1$ to $\texttt{T}_\texttt{n}$ and a return type (or component type) T. It represents the set of values given by $V(\texttt{T}_1 * \cdots * \texttt{T}_\texttt{n}\ \texttt{->}\ \texttt{T}) = V(\texttt{T})^{V(\texttt{T}_1)\times\cdots\times V(\texttt{T}_\texttt{n})}$.

   We will use the standard terminology and call the sets $V(\texttt{T}_1)\times\cdots\times V(\texttt{T}_\texttt{n})$ and $V(\texttt{T})$, the *Domain* and *Codomain*, respectively.

   If $\texttt{n} > 1$ 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 given by the lexicographical order on the function's domain, *i.e.* the Cartesian product $V(\texttt{T}_1)\times\cdots\times V(\texttt{T}_\texttt{n})$.
4. (FunctionDomainEquality) Two function parameter types $\texttt{T}_1$ and $\texttt{T}_2$ are said to be *equal iff* they are mutually assignable to each other and their sets of values are equal, *i.e.* $V(\texttt{T}_1) = V(\texttt{T}_2)$.[18]
5. (FunctionCompatibility) A function type $(\texttt{T}_1 * \cdots * \texttt{T}_\texttt{n}\ \texttt{->}\ \texttt{T})$ is compatible with another function type $(\texttt{U}_1 * \cdots * \texttt{U}_\texttt{n}\ \texttt{->}\ \texttt{U})$ *iff* $\texttt{T}_\texttt{i}$ is equal to $\texttt{U}_\texttt{i}$ (according to (FunctionDomainEquality)) for $\texttt{i} \in [1, \texttt{n}]$ and T is compatible with U.
6. (FunctionAssignability) A function type $(\texttt{T}_1 * \cdots * \texttt{T}_\texttt{n}\ \texttt{->}\ \texttt{T})$ is assignable to another function type $(\texttt{U}_1 * \cdots * \texttt{U}_\texttt{n}\ \texttt{->}\ \texttt{U})$ *iff* $\texttt{T}_\texttt{i}$ is equal to $\texttt{U}_\texttt{i}$ (according to (FunctionDomainEquality)) for $\texttt{i} \in [1, \texttt{n}]$ and T is assignable to U.

Restrictions

1. (FunctionDomainScalar) The parameter types $\texttt{T}_\texttt{i}$ for $\texttt{i} \in [1, \texttt{n}]$ of a function type $(\texttt{T}_1 * \texttt{T}_2 * \cdots * \texttt{T}_\texttt{n}\ \texttt{->}\ \texttt{T})$ shall be of scalar type.

[18] This means *e.g.*, that the parameter types `int [0,7]` and `int unsigned 3` are considered equal.

8.2.4 Array Types

Arrays have different restrictions and a different way of accessing components compared to functions, but they share semantics with functions. For example, an array type `bool^(3)` (an array of 3 `bool`) represents the same set of values as the function type (`int[0, 2]` $\rightarrow$ `bool`), but with a different syntax for accessing 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 array type $\texttt{T}\texttt{\^{}}(\texttt{E}_1, \ldots, \texttt{E}_n)$ consists of the array *Dimensions* $\texttt{E}_1$ to $\texttt{E}_n$ and the base type (the component type) `T`. It represents the same set of values as the corresponding function type $(\texttt{int}\ [0, \texttt{E}_1 - 1] * \cdots * \texttt{int}\ [0, \texttt{E}_n - 1]\ \texttt{->}\ \texttt{T})$ (see (FunctionValues)).

   If $\texttt{n} > 1$ then the array type is said to be *Multidimensional*.
3. (ArrayCompatibility) An array type $\texttt{T}_1\texttt{\^{}}(\texttt{D}_1, \texttt{D}_2, \ldots, \texttt{D}_n)$ is compatible with another array type $\texttt{T}_2\texttt{\^{}}(\texttt{E}_1, \texttt{E}_2, \ldots, \texttt{E}_n)$ if each $\texttt{D}_i = \texttt{E}_i$ and $\texttt{T}_1$ is compatible with $\texttt{T}_2$. Note that arrays and functions are not compatible with one another.
4. (ArrayAssignability) An array type $\texttt{T}_1\texttt{\^{}}(\texttt{D}_1, \texttt{D}_2, \ldots, \texttt{D}_n)$ is assignable to another array type $\texttt{T}_2\texttt{\^{}}(\texttt{E}_1, \texttt{E}_2, \ldots, \texttt{E}_n)$ if each $\texttt{D}_i = \texttt{E}_i$ and $\texttt{T}_1$ is assignable to $\texttt{T}_2$. Note that arrays cannot be assigned to functions, nor can functions be assigned to arrays.

Restrictions

1. (ArrayDimInteger) $\texttt{E}_i$ for $\texttt{i} \in [1, \texttt{n}]$ shall be of integer type.
2. (ArrayDimConstant) $\mathcal{SF}(\texttt{E}_i) = 2$ for $\texttt{i} \in [1, \texttt{n}]$.
3. (ArrayDimNotNil) For each $\texttt{i} \in [1, \texttt{n}]$ there shall exist $j \geq 0$ such that $M_j(\texttt{E}_i, 0) \neq$ **nil**.

8.3 Named Types

Syntax

(NamedTypeSyntax)

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

Semantics

1. (NamedType) A named type (`<named_type>`) is a type that represents the same set of values as the type it refers to. Note that a named type is not an alias for another type, it is a type in itself. Hence, by (TypeDef Unicity), the following is not legal:

```
Types:
  sort       S1;
  S1         S2;
  sort {A} < S2; // This is an error.
```

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

The collection type is the type associated with collections (`<collection>`, see Section 12). Expressions of collection type can be used to assign values to variables of composite type.

A variable declared with type `tuple{bool, int}` or `struct{b : bool, i : int}` can, for example, be assigned the collection `{false, 0}`. 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. (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.
2. (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.
3. (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.
4. (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.
5. (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 $T_2\hat{}(E_2, \ldots, E_n)$.
6. (CollFuncAssignability) A collection type T_1 is assignable to a function type $(T_2 \text{ -> } T_3)$ *iff* T_2 is an ordered type and T_1 has $|T_2|$ components and each one is assignable to T_3.

7. (CollMultiVarFuncAssignability) A collection type T_1 is assignable to a multivariate function type $(T_2 * T_3 * \cdots * T_{n-1} \text{ -> } T_n)$ $(n > 3)$ *iff* T_2 is an ordered type, T_1 has $|T_2|$ components and each one is assignable to $(T_3 * \cdots * T_{n-1} \text{ -> } T_n)$.

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 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).

It is worth noting that an unsized composite type has only unsized component types, and the "unsize-operation" does not affect *e.g.* function domain types. To give two examples, the unsized copy of `tuple {int signed` $\mathtt{E}_1$`, int [`$\mathtt{E}_2, \mathtt{E}_3$`]}` is `tuple {int, int}`, and the unsized copy of `(int [`$\mathtt{E}_1, \mathtt{E}_2$`] -> int [`$\mathtt{E}_3, \mathtt{E}_4$`])` is `(int [`$\mathtt{E}_1, \mathtt{E}_2$`] -> int)`.

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.

Restrictions

(empty)

8.4.3 Union Types

Union types provide a means of defining a supertype that encompasses multiple mutually compatible types which might otherwise lack a shared supertype (even though they are compatible).

Syntax

(empty)

Semantics

1. (UnionScalar) The union type of `n` compatible non-sort scalar types $\mathtt{T}_1 \ldots \mathtt{T}_n$ is the unsized copy of $\mathtt{T}_1$.
2. (UnionSort) The union type of `n` sort or sort union types $\mathtt{T}_1 \ldots \mathtt{T}_n$ (*i.e.* each $\mathtt{T}_i$ may be either a sort type or another sort union type) is a special "sort union type" $\mathtt{T}_u$, which represents the set of values $V(\mathtt{T}_1) \cup \cdots \cup V(\mathtt{T}_n)$. We will count the sort union type among the scalar types of HLL.
3. (UnionComposite) The union type of `n` compatible composite types $\mathtt{T}_1 \ldots \mathtt{T}_n$, which are not collection types, is a type $\mathtt{T}_r$, in all respects the same as any of the $\mathtt{T}_i$, but where each component type of $\mathtt{T}_r$ is the union type of the `n` corresponding component types of $\mathtt{T}_1 \ldots \mathtt{T}_n$.
4. (UnionTupleCollection) The union type of a tuple type $\mathtt{T}_1$ and a compatible collection type $\mathtt{T}_2$ is a tuple type $\mathtt{T}_r$, in all respects the same as $\mathtt{T}_1$, but where each component type of $\mathtt{T}_r$ is the union type of the corresponding component types of $\mathtt{T}_1$ and $\mathtt{T}_2$. This union is needed to represent the type of `SELECT` expressions where the default value is a collection, see (QuantSelectType), and explains the need for (CollTupleCompatibility) in Section 8.4.1.
5. (SortUnionCompatibility) A sort union type is compatible with both sort types and other sort union types.
6. (SortUnionAssignability) A sort union type $\mathtt{T}_u$ of the types $\mathtt{T}_1 \ldots \mathtt{T}_n$ is assignable to a sort type $\mathtt{T}_s$ *iff* each $\mathtt{T}_i$ for $\mathtt{i} \in [1, \mathtt{n}]$ is assignable to $\mathtt{T}_s$.[19]

Restrictions

(empty)

[19] Note that this definition relies on a recursion, since the $\mathtt{T}_i$ may themselves be sort unions. The recursion is well-founded since HLL does not allow the creation of named sort union types, meaning that sort union types cannot refer to themselves, meaning in turn that the recursion will eventually stop when encountering a sort type.

9 Accessors

Accessors are used to designate components of expressions of composite type. They are the syntactic equivalent of the semantic concept of component projections (see Definition 6).

Below follow examples of different composite expressions, each with three components, and the corresponding accessors for designating the first, the second, and the third component.

Declaration	*First*	*Second*	*Third*
`tuple {bool, int, bool} T;`	`.0`	`.1`	`.2`
`struct {a: bool, b: int, c: bool} S;`	`.a`	`.b`	`.c`
`bool A[3];`	`[0]`	`[1]`	`[2]`
`bool F(int [4, 6]);`	`(4)`	`(5)`	`(6)`

To actually access, or reference, the components, a projection expression (defined in Section 10.6) is used. For example, accessing the second component of the struct variable `S` would be written `S.b`.

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 steps i and k and relative to an array type $\mathtt{T}\tilde{}(\mathtt{D}_1, \mathtt{D}_2, \ldots, \mathtt{D}_n)$, $[\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_n]$ designates the array component of type `T` at the indices given by $(M_i(\mathtt{E}_1, k), M_i(\mathtt{E}_2, k), \ldots, M_i(\mathtt{E}_n, k))$, unless some index is out of bounds or **nil**, in which case it designates **nil**`^(T)`.
4. (AccFunction) At steps i and k and relative to a function type $(\mathtt{T}_1 * \mathtt{T}_2 * \cdots * \mathtt{T}_n \texttt{ -> } \mathtt{T}_r)$, $(\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_n)$ designates the function component of type $\mathtt{T}_r$ corresponding to the arguments $(M_i(\mathtt{E}_1, k), M_i(\mathtt{E}_2, k), \ldots, M_i(\mathtt{E}_n, k))$, unless some argument is out of the function domain or **nil**, in which case it designates **nil**`^(T)`.

Restrictions

1. (ArrayIndexInteger) Each E_i of an accessor $[E_1, E_2, \ldots, E_n]$ shall be of integer type.
2. (FunctionArgumentScalar) Each E_i of an accessor $(E_1, E_2, \ldots, E_n)$ shall be of scalar type.

10 Expressions

Syntax

(ExprSyntax)

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

Semantics

1. (ExpressionValues) Any expression `E` of type `T` satisfies $M_i(\mathtt{E}, k) \in V^*(\mathtt{T})$ for all $i, k \in \mathbb{N}$. In particular, if some scalar leaf of `T` is an empty scalar type, then the stream of the corresponding scalar leaf of an expression `E` of type `T` will carry the value **nil** in each time step.
2. (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 $\mathtt{E}\langle \mathtt{V}_1 := \mathtt{R}_1, \mathtt{V}_2 := \mathtt{R}_2, \ldots, \mathtt{V}_n := \mathtt{R}_n \rangle$ to denote substitution, *i.e.* the process of replacing all free occurrences of the variables $\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n$ in the expression `E` with expressions $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n$.

   An occurrence of a variable is free to be replaced by a substitution *iff* the scope of the variable is the same as, or an ascendant to, the scope in which the substitution occurs.

   As an example $(\mathtt{x} + \mathtt{y} = \mathtt{z})\langle \mathtt{z} := 5 \rangle$ is equivalent to $\mathtt{x} + \mathtt{y} = 5$. By contrast, $(\mathtt{SOME}\ \mathtt{z} : [0, 4]\ (\mathtt{x} + \mathtt{y} = \mathtt{z}))\langle \mathtt{z} := 5 \rangle$ is equivalent to $\mathtt{SOME}\ \mathtt{z} : [0, 4]\ (\mathtt{x} + \mathtt{y} = \mathtt{z})$ (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 step, `if` $\mathtt{E}_1$ `then` $\mathtt{E}_2$ `else` $\mathtt{E}_3$ of type `T` evaluates to 1) **nil**`^(T)` 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` is the union type of the type of $\mathtt{E}_2$ and the type of $\mathtt{E}_3$. Formally, for all i and k:

$$M_i(\texttt{if E}_1\texttt{ then E}_2\texttt{ else E}_3, k) = \begin{cases} \textbf{nil}\texttt{^(T)} & \text{if } M_i(\mathtt{E}_1, k) = \textbf{nil} \\ M_i(\mathtt{E}_2, k) & \text{if } M_i(\mathtt{E}_1, k) = \text{true} \\ M_i(\mathtt{E}_3, k) & \text{otherwise } (M_i(\mathtt{E}_1, k) = \text{false}) \end{cases}$$

Static Flag

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

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 expressions 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 are – just as any HLL expression of function (or array) type – modelled by streams of combinational 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 combinational function. The combinational function applied to argument values $\mathtt{V}_1$ to $\mathtt{V}_n$ of its domain evaluates to the value of the right hand side expression at time step k with each argument value substituted for its corresponding formal parameter.

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 expressions, 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 of a lambda expression `lambda DS : FP := E`, where `E` is an expression of type $\mathtt{T}_E$, is equal to `calc_type`$(\mathtt{T}_E, \mathtt{DS})$. The function `calc_type` is defined by (DeclaratorTypeCalc) in Section 7.

3. (LambdaMultFormalParam) `lambda` $\mathtt{D}_1 \ldots \mathtt{D}_n : \mathtt{P}_1 \ldots \mathtt{P}_n := \mathtt{E}$ is reducible[20] to `lambda` $\mathtt{D}_1 \ldots \mathtt{D}_{n-1} : \mathtt{P}_1 \ldots \mathtt{P}_{n-1} :=$ (`lambda` $\mathtt{D}_n : \mathtt{P}_n := \mathtt{E}$).

4. (LambdaArray) `lambda`$[\mathtt{E}_1, \ldots, \mathtt{E}_n] : [\mathtt{x}_1, \ldots, \mathtt{x}_n] := \mathtt{E}$, where `E` is of type $\mathtt{T}_E$, and $\mathtt{x}_j$ for $\mathtt{j} \in [1, n]$ is, by definition, of type `int`, is an expression of type $\mathtt{T}_E\hat{}(\mathtt{E}_1, \ldots, \mathtt{E}_n)$ such that, for each step i and k and integers $\mathtt{V}_1, \ldots, \mathtt{V}_n$ with $\forall \mathtt{j} \in [1, n] : 0 \leq \mathtt{V}_j < \mathtt{E}_j$:

$$M_i(\mathtt{lambda}[\mathtt{E}_1, \ldots, \mathtt{E}_n] : [\mathtt{x}_1, \ldots, \mathtt{x}_n] := \mathtt{E}, k)(\mathtt{V}_1, \ldots, \mathtt{V}_n) = M_i(\mathtt{E}\texttt{<}\mathtt{x}_1 := \mathtt{V}_1, \ldots, \mathtt{x}_n := \mathtt{V}_n\texttt{>}, k)$$

[20] `lambda[1] : [i] := (lambda[1] : [i] := i)` is not reducible to `lambda[1][1] : [i][i] := i` (since the latter expression is not legal), so the two are not equivalent.

5. (LambdaFunction) $\texttt{lambda}(\texttt{T}_1, \ldots, \texttt{T}_n) : (\texttt{x}_1, \ldots, \texttt{x}_n) := \texttt{E}$, where E is of type $\texttt{T}_E$, and $\texttt{x}_j$ for $\texttt{j} \in [1, \texttt{n}]$ is, by definition, of type $\texttt{T}_j$, is an expression of type $(\texttt{T}_1 * \cdots * \texttt{T}_n \texttt{ -> } \texttt{T}_E)$ such that, for each step i and k and values $\texttt{V}_1, \ldots, \texttt{V}_n$ with $\forall \texttt{j} \in [1, \texttt{n}] : \texttt{V}_j \in V(\texttt{T}_j)$:

$$M_i(\texttt{lambda}(\texttt{T}_1, \ldots, \texttt{T}_n) : (\texttt{x}_1, \ldots, \texttt{x}_n) := \texttt{E}, k)(\texttt{V}_1, \ldots, \texttt{V}_n) = M_i(\texttt{E<x}_1 := \texttt{V}_1, \ldots, \texttt{x}_n := \texttt{V}_n\texttt{>}, k)$$

Static Flag

1. (LambdaStaticFlag) $\mathcal{SF}$(<lambda_expr>) = 0.
2. (FormalParamStaticFlag) $\mathcal{SF}(\texttt{i}) = 1$ for a lambda parameter i (an <id> of a <formal_param>).

Restrictions

1. (LambdaParamUnicity) Two parameters of a lambda expression may not have the same name.
2. (LambdaParamsBound) |DS| = |FP| (the number of elements of the list DS shall 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>) at the corresponding position in the list DS.

   Furthermore, if D is of the form "[" <expr_list> "]" then F shall be of the form "[" <id_list> "]", and if D is of the form "(" <type_list> ")" then F shall be of the form "(" <id_list> ")".

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) $\texttt{E}_1$ `%` $\texttt{E}_2$ (remainder) is equivalent to $\texttt{E}_1$ `-` ($\texttt{E}_1$ `/` $\texttt{E}_2$) `*` $\texttt{E}_2$.

Core Constructs:

16. (BoolOr) $\texttt{E}_1$ `#` $\texttt{E}_2$ (Boolean or) is an expression of type `bool` for which $M_i(\texttt{E}_1 \texttt{ \# } \texttt{E}_2, k) = M_i(\texttt{E}_1, k) \vee_{\textbf{nil}} M_i(\texttt{E}_2, k)$ holds for all i and k. 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
A	false	false	**nil**	true
	nil	**nil**	**nil**	true
	true	true	true	true

17. (IntLt) $\texttt{E}_1$ `<` $\texttt{E}_2$ (less than) is an expression of type `bool` for which $M_i(\texttt{E}_1 \texttt{ < } \texttt{E}_2, k) = M_i(\texttt{E}_1, k) <_{\textbf{nil}} M_i(\texttt{E}_2, k)$ holds for all i and k. 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) $\texttt{E}_1$ `+` $\texttt{E}_2$ (addition) is an expression of type `int` for which $M_i(\texttt{E}_1 \texttt{ + } \texttt{E}_2, k) = M_i(\texttt{E}_1, k) +_{\textbf{nil}} M_i(\texttt{E}_2, k)$ holds for all i and k. 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) $\texttt{E}_1$ `*` $\texttt{E}_2$ (multiplication) is an expression of type `int` for which $M_i(\texttt{E}_1 \texttt{ * } \texttt{E}_2, k) = M_i(\texttt{E}_1, k) *_{\textbf{nil}} M_i(\texttt{E}_2, k)$ holds for all i and k. 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) $\texttt{E}_1$ `/` $\texttt{E}_2$ (integer division) is an expression of type `int` for which $M_i(\texttt{E}_1 \texttt{ / } \texttt{E}_2, k) = M_i(\texttt{E}_1, k) /_{\textbf{nil}} M_i(\texttt{E}_2, k)$ holds for all i and k. 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) $\mathtt{E}_1 \mathtt{\hat{}} \mathtt{E}_2$ (exponentiation) is an expression of type `int` for which $M_i(\mathtt{E}_1 \mathtt{\hat{}} \mathtt{E}_2, k) = M_i(\mathtt{E}_1, k) \mathbin{\hat{}_{\mathbf{nil}}} M_i(\mathtt{E}_2, k)$ holds for all i and k. The operator $\hat{}_{\mathbf{nil}}$ is defined in the following table:

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

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

22. (OpEq) $\mathtt{E}_1 = \mathtt{E}_2$, where $\mathtt{E}_1$ and $\mathtt{E}_2$ are of (any) compatible types, is an expression of type `bool` for which $M_i(\mathtt{E}_1 = \mathtt{E}_2, k) = (M_i(\mathtt{E}_1, k) =_{\mathbf{nil}} M_i(\mathtt{E}_2, k))$ holds for all i and k.

For any type `T` (the union type of the argument types) the operation $=_{\mathbf{nil}}$ is defined for $V^*(\mathtt{T}) \times V^*(\mathtt{T})$ as follows.

For scalar values, $=_{\mathbf{nil}}$ is defined by the following table:

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

For composite values $=_{\mathbf{nil}}$ is defined as:

$$(A =_{\mathbf{nil}} B) = \begin{cases} A =_{\mathbf{nil}} B & \text{if } A \text{ and } B \text{ are scalars} \\ \text{false} & \text{if } \exists_{p \in \mathcal{P}(\mathtt{T})}(p(A) =_{\mathbf{nil}} p(B)) = \text{false} \\ \mathbf{nil} & \text{if } \exists_{p \in \mathcal{P}(\mathtt{T})}(p(A) =_{\mathbf{nil}} p(B)) = \mathbf{nil} \\ \text{true} & \text{otherwise,} \end{cases}$$

where the cases on the right hand side should be considered in order from top to bottom. $\mathcal{P}(\mathtt{T})$ denotes the set of component projections associated to `T` (see Definition 6).

Precedence and Associativity:

23. (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 `#` and `<->` shall be of type `bool`.
2. (EqOperandsFiniteCompatible) The operands of = shall be of compatible and finite-dimensional types.
3. (IntCoreBinopOperandsInt) The operands of `< + * / ^` shall be of integer type.
4. (SecondShiftOperandStatic) $\mathcal{SF}(\texttt{E}_2) \geq 1$ for an expression $\texttt{E}_1 \circ \texttt{E}_2$ with $\circ \in \{\ll, \gg\}$.
5. (SecondShiftOperandNonNegative) $\texttt{E}_2 \geq 0$ for an expression $\texttt{E}_1 \circ \texttt{E}_2$ with $\circ \in \{\ll, \gg\}$.

10.4 Membership Expressions

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

Syntax

(MembershipSyntax)

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

Semantics

1. (Domain) A *Domain* `D` (`<domain>`) is modelled by a stream of (possibly different) sets of values. We will write $\mathtt{D}_k$ to denote the set of values of `D` at time step k.

2. (DomainAsType) The stream of a `<domain>` which is a `<type_domain>` consists in each time step of the (possibly empty) set of values of the type `T` which the `<type_domain>` refers to. The *Underlying type* of such a `<domain>` is defined to be `T`.

3. (DomainAsRange) The stream of a `<domain>` which is a `<range>` $[\mathtt{E}_1, \mathtt{E}_2]$ is in each time step k the (possibly empty) set of values given by the range $[M_i(\mathtt{E}_1, k), M_i(\mathtt{E}_2, k)]$. The underlying type of such a `<domain>` is defined to be the type `int`.

4. (MembershipType) `E : T`, where `T` is a `<type_domain>` and `E` is of a type which is compatible with `T`, is an expression of type `bool` such that:

$$M_i(\mathtt{E} : \mathtt{T}, k) = \begin{cases} \textbf{nil} & \text{if } M_i(\mathtt{E}, k) = \textbf{nil} \\ \text{true} & \text{if } M_i(\mathtt{E}, k) \in V(\mathtt{T}) \\ \text{false} & \text{otherwise} \end{cases}$$

5. (MembershipRange) $\mathtt{E} : [\mathtt{E}_1, \mathtt{E}_2]$, where `E`, $\mathtt{E}_1$ and $\mathtt{E}_2$ is of integer type, is an expression of type `bool` such that:

$$M_i(\mathtt{E} : [\mathtt{E}_1, \mathtt{E}_2], k) = \begin{cases} \textbf{nil} & \text{if } M_i(\mathtt{E}, k) = \textbf{nil} \text{ or } M_i(\mathtt{E}_1, k) = \textbf{nil} \\ & \text{or } M_i(\mathtt{E}_2, k) = \textbf{nil} \\ \text{true} & \text{if } M_i(\mathtt{E}, k) \in [M_i(\mathtt{E}_1, k), M_i(\mathtt{E}_2, k)] \\ \text{false} & \text{otherwise} \end{cases}$$

6. (MembershipPrecedence) The operator : has the same precedence as the operator =, and is left-associative.

Static Flag

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

Restrictions

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

10.5 Unop Expressions

Syntax

(UnopSyntax)

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

Semantics

1. (BoolNeg) `~E` (Boolean negation) is an expression of type `bool` such that for all i and k

$$M_i(\texttt{~E}, k) = \begin{cases} \textbf{nil} & \text{if } M_i(\texttt{E}, k) = \textbf{nil} \\ \text{false} & \text{if } M_i(\texttt{E}, k) = \text{true} \\ \text{true} & \text{otherwise} \end{cases}$$

2. (IntNeg) `-E` (integer negation) is an expression of type `int` such that for all i and k

$$M_i(-\texttt{E}, k) = \begin{cases} \textbf{nil} & \text{if } M_i(\texttt{E}, k) = \textbf{nil} \\ -M_i(\texttt{E}, k) & \text{otherwise} \end{cases}$$

Static Flag

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

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 expressions of composite type. For example `A[4]` selects the array component at index 4, `f(true)` selects the function component that corresponds to argument `true`, and so on. An important thing to note is that `f(x)`, where `x` is an arbitrary expression, selects, in time step k, the function component that corresponds to the *value* of the stream 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, $M(\mathtt{x}, k) = M(\mathtt{y}, k)$ `->` $M(\mathtt{f(x)}, k) = M(\mathtt{f(y)}, k)$, *regardless of the history (past or future values) of* `x` *and* `y`.

Syntax

(ProjExprSyntax)

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

Forward References

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

Semantics

1. (ProjMultipleAcc) A projection expression $\mathtt{E\ A_1\ A_2 \ldots A_n}$ is equivalent to $(\ldots((\mathtt{E\ A_1})\ \mathtt{A_2})\ldots)\ \mathtt{A_n}$.
2. (ProjTupleStructAcc) `E.M` where `E` is of tuple or struct type $\mathtt{T_E}$ designates the component `M` of `E`. The following formal definition uses an overloading of operator @ defined in Section 8.2.1 for tuples and in Section 8.2.2 for structs. For all steps i and k:
$$M_i(\mathtt{E.M}, k) = M_i(\mathtt{E}, k)@\mathtt{M}$$
3. (ProjArrayAcc) $\mathtt{E}[\mathtt{E_1}, \mathtt{E_2}, \ldots, \mathtt{E_n}]$ where `E` is of array type $\mathtt{T\hat{}}(\mathtt{D_1}, \mathtt{D_2}, \ldots, \mathtt{D_n})$ is an expression of type `T` such that, for all steps i and k:
$M_i(\mathtt{E}[\mathtt{E_1}, \mathtt{E_2}, \ldots, \mathtt{E_n}], k)$
$$= \begin{cases} M_i(\mathtt{E}, k)(M_i(\mathtt{E_1}, k), \ldots, M_i(\mathtt{E_n}, k)) & \text{if } \forall \mathtt{j} \in [1, \mathtt{n}]: \\ & M_i(\mathtt{E_j}, k) \in [0, \mathtt{D_j} - 1] \\ \hline \mathbf{nil}\hat{}(\mathtt{T}) & \text{otherwise} \end{cases}$$
4. (ProjFunctionAcc) $\mathtt{E}(\mathtt{E_1}, \mathtt{E_2}, \ldots, \mathtt{E_n})$ where `E` is of function type $(\mathtt{T_1} * \mathtt{T_2} * \cdots * \mathtt{T_n}$ `->` $\mathtt{T})$ is an expression of type `T` such that, for all steps i and k:
$M_i(\mathtt{E}(\mathtt{E_1}, \mathtt{E_2}, \ldots, \mathtt{E_n}), k)$
$$= \begin{cases} M_i(\mathtt{E}, k)(M_i(\mathtt{E_1}, k), \ldots, M_i(\mathtt{E_n}, k)) & \text{if } \forall \mathtt{j} \in [1, \mathtt{n}]: \\ & M_i(\mathtt{E_j}, k) \in V(\mathtt{T_j}) \\ \hline \mathbf{nil}\hat{}(\mathtt{T}) & \text{otherwise} \end{cases}$$

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`
$[E_1, E_2, \ldots, E_n]$	Array with `n` dimensions
$(E_1, E_2, \ldots, E_n)$	Function ($T_1 * T_2 * \cdots * T_n$ `->` `T`) where the type of E_i is assignable to T_i

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) The expression `(E)` has the same type as `E`, and $M_i((\texttt{E}), k) = M_i(\texttt{E}, k)$ for any steps i and k. 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 constant expression of type `bool` such that $M_i(\texttt{<true>}, k)$ = true for all i and k.
2. (BoolLitFalse) `<false>` is a constant expression of type `bool` such that $M_i(\texttt{<false>}, k)$ = false for all i and k.

Static Flag

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

Restrictions

(empty)

10.7.2 Integer Literals

Syntax

(IntLitSyntax)

```
<dec_literal> ::= regexp: [0-9](_?[0-9])*
<bin_literal> ::= regexp: 0[Bb][0-1](_?[0-1])*
<hex_literal> ::= regexp: 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 constant expression of type `int` such that $M_i($`<int_literal>`$, k) =$ `K` for all i and k.

Static Flag

1. (IntLitStaticFlag) $\mathcal{SF}($`<int_literal>`$) = 2$.

Restrictions

(empty)

10.7.3 Named Expressions

Named expressions are references to variables.

Syntax

(NamedExprSyntax)

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

Semantics

1. (NamedExpr) If the `<path_id>` of a `<named_expr>` refers to an existing variable (in the namespace of expressions), then the `<named_expr>` is that variable.

   Please note that a variable may exist due to language constructs such as *e.g.* declarations and/or definitions (defined in Sections 11 and 12 respectively) appearing *after* the `<named_expr>` in the HLL text.

2. (NamedExprImplicitDecl) Otherwise, if the `<path_id>` does not refer to an existing variable, the `<named_expr>` (which must be unqualified as according to (PathIdNoImplicitDecl)) refers to a unique implicit input variable of type `bool`. The input variable is implicitly declared by the `<named_expr>`. The implicit declaration is made in the top-level scope (of the namespace of expressions) of the inner-most user namespace in which the `<named_expr>` occurs, or else, if the `<named_expr>` does not occur inside a user namespace, on the global top-level. The implicit declaration causes the variable to exist in its scope and to be visible everywhere in its scope regardless of the position of the `<named_expr>`.

   If there are several `<named_expr>` in an HLL text referring to the same nonexisting variable, it is undefined which one of the unqualified references (if there is more than one) is responsible for implicitly declaring the variable. Since the position of the implicit declaration of a `<named_expr>` does not matter, any one of them can be chosen without any difference in semantics.

Static Flag

1. (NamedExprStaticFlag) The static flag of a `<named_expr>` is equal to the static flag of the variable it refers to.

2. (NamedExprUndefinedStaticFlag) $\mathcal{SF}(\mathtt{v}) = \mathtt{0}$ for a variable `v` which is either explicitly declared (as according to (Declaration) and the other semantic items of Section 11) or implicitly declared (as according to (NamedExprImplicitDecl) above), but not defined (as according to (Declaration) and the other semantic items of Section 12).

Restrictions

(empty)

10.7.4 Next Expressions

Syntax

(NextExprSyntax)

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

Semantics

1. (NextExpr) $M_i(\texttt{X(E)}, k) = M_i(\texttt{E}, k+1)$ for all steps i and k. `X(E)` and `E` have the same type.

Static Flag

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

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>(E1, E2)` is an initialized memory of type `T`, where `E2` defines the initial value for the initial time step and `E1` defines the memorized value for all other time steps. Formally, the value of the expression at steps i and k is defined as follows:

$$M_i(\texttt{pre<T>}(\texttt{E}_1, \texttt{E}_2), k) = \begin{cases} D(\texttt{T}, M_i(\texttt{E}_2, 0)) & k = 0 \\ D(\texttt{T}, M_i(\texttt{E}_1, k-1)) & k > 0 \end{cases}$$

3. (PreTyped) `pre<T>(E)` is an uninitialized memory of type `T` which takes the value **nil** in the initial time step. Formally, the value of the expression at steps i and k is defined as follows:

$$M_i(\texttt{pre<T>}(\texttt{E}), k) = \begin{cases} \textbf{nil}\hat{}(\texttt{T}) & k = 0 \\ D(\texttt{T}, M_i(\texttt{E}, k-1)) & k > 0 \end{cases}$$

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(E1, E2)` is equivalent to `pre<T>(E1, E2)` where `T` is the union type (see 8.4.3) of the types of `E1` and `E2`.

Static Flag

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

Restrictions

1. (PreOperandsAssignable) The types of the operands `E1` and `E2` of `pre<T>(E1, E2)` and the type of the operand `E` of `pre<T>(E)`, shall be assignable to `T`.
2. (PreUntypedOperandsCompatible) The types of the operands `E1` and `E2` of `pre(E1, E2)` shall be compatible.

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 identifier `i`.
   The `SUM` quantifier is defined in Section 10.7.10.
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 fresh identifiers `i` and `bit_is_one`, 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`, ..., `E_n`, K)` of `n` Boolean expressions E_i and an integer expression `K` is true exactly when less than `K` of the Boolean expressions 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`, ..., `E_n`, K)` is equivalent to `~population_count_lt(`E_1`, `E_2`, ..., `E_n`, K + 1)`.

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

11. (OpBitwiseOr) `$or(`E_1`, `E_2`)` (bitwise or) satisfies for all i and k,

    $$M_i(\texttt{\$or}(E_1, E_2), k) = \textbf{nil} \text{ whenever } M_i(E_1, k) = \textbf{nil} \text{ or } M_i(E_2, k) = \textbf{nil}.$$

    Otherwise the expression `$or(`E_1`, `E_2`)` is equivalent to

    `bin2s((lambda[C] : [i] := s2bin(`E_1`, C)[i] # s2bin(`E_2`, C)[i]), C)`

    for a fresh identifier `i` and a constant `C` big enough to represent all significant bits of E_1 and E_2, *i.e.*

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

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

12. (OpBitwiseXor) $\texttt{\$xor}(\texttt{E}_1, \texttt{E}_2)$ (bitwise xor) is equivalent to

$$\texttt{bin2s}((\texttt{lambda[C] : [i] := s2bin}(\texttt{E}_1, \texttt{C})[\texttt{i}] \texttt{ \#! s2bin}(\texttt{E}_2, \texttt{C})[\texttt{i}]), \texttt{C})$$

for a fresh identifier `i` and a constant `C` big enough to represent all significant bits of $\texttt{E}_1$ and $\texttt{E}_2$, *i.e.*

$$\texttt{C} \geq \log_2(\max(\text{ub}(\texttt{E}_1) + 1, \text{ub}(\texttt{E}_2) + 1, |\text{lb}(\texttt{E}_1)|, |\text{lb}(\texttt{E}_2)|)) + 1$$

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

13. (OpBitwiseNot) $\texttt{\$not}(\texttt{E})$ (bitwise not) is equivalent to

$$\texttt{bin2s}((\texttt{lambda[C] : [i] := \~{}s2bin}(\texttt{E}, \texttt{C})[\texttt{i}]), \texttt{C})$$

for a fresh identifier `i` and a constant `C` big enough to represent all significant bits of `E`, *i.e.*

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

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

14. (OpBitwiseAnd) $\texttt{\$and}(\texttt{E}_1, \texttt{E}_2)$ (bitwise and) is equivalent to

$$\texttt{\$not}(\texttt{\$or}(\texttt{\$not}(\texttt{E}_1), \texttt{\$not}(\texttt{E}_2))).$$

Static Flag

1. (FunopUnaryStaticFlag) $\mathcal{SF}(\circ(\texttt{E}_1)) = \mathcal{SF}(\texttt{E}_1)$ for $\circ \in \{\texttt{\$abs}, \texttt{\$not}\}$.
2. (FunopBinaryStaticFlag) $\mathcal{SF}(\circ(\texttt{E}_1, \texttt{E}_2)) = \min(\mathcal{SF}(\texttt{E}_1), \mathcal{SF}(\texttt{E}_2))$ for $\circ \in \{\texttt{\$min}, \texttt{\$max}, \texttt{\$and}, \texttt{\$xor}, \texttt{\$or}\}$.
3. (FunopNaryStaticFlag) $\mathcal{SF}(\circ(\texttt{E}_1, \ldots, \texttt{E}_n)) = 0$ 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. (SecondBin2IntOperandStatic) $\mathcal{SF}(\texttt{E}_2) = 2$ for an expression $\circ(\texttt{E}_1, \texttt{E}_2)$ with $\circ \in \{$`bin2u`, `bin2s`$\}$.
4. (PopCountNumberStatic) $\mathcal{SF}(\texttt{K}) \geq 1$ for an expression $\circ(\texttt{E}_1, \ldots, \texttt{E}_n, \texttt{K})$ 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)`.
4. (CastNamedType) `cast<T>(E)` where `T` is a named type defined, directly or indirectly, as the type `int S` (where `S` must be of the form `<sign>`, see Section 8.1.2) is equivalent to `cast<int S>(E)`.

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 expression, where a single component has been arbitrarily modified. For example, if `A` is a Boolean array, then "`(A with [0] := ~A[0])`" is a Boolean array with the same components as `A` except for the one at index 0, which has been negated. With expressions are particularly useful when translating assignments of an imperative language into HLL.

It is worth noting that modification of components out of array bounds or function domains, does not result in **nil**, but instead simply means no modification is made. As an example, if `A` is our array from above, then "`(A with [-1] := x)`" is an array with the same components as `A`. This behaviour is formalised by (WithArrayAcc) and (WithFunctionAcc) below.

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

$$(\texttt{A with [0] := } \{1, 2, 3\})$$

is expanded, assuming the projection expression `A[0]` is of array type (*i.e.* `A` is an array of arrays), into the equivalent formula

$$(((\texttt{A with [0][0] := } 1) \texttt{ with [0][1] := } 2) \texttt{ with [0][2] := } 3).$$

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. (WithMultipleAcc) $(\mathtt{E_1\ with\ A_1\ A_2 \ldots A_n := E_2})$ is equivalent to

$$(\mathtt{E_1\ with\ A_1\ A_2 \ldots A_{n-1} := (E_1\ A_1\ A_2 \ldots A_{n-1}\ with\ A_n := E_2)})$$

2. (WithArrayAcc) $(\mathtt{E_1\ with\ [E_3,\ E_4, \ldots, E_n] := E_2})$ where $\mathtt{E_1}$ is of type $\mathtt{T\tilde{}(C_3,\ C_4, \ldots, C_n)}$ is equivalent to

$$\begin{aligned}&(\mathtt{lambda[C_3,\ C_4, \ldots, C_n] : [i_3,\ i_4, \ldots, i_n] :=}\\ &\quad \mathtt{if\ i_3 = E_3\ \&\ i_4 = E_4\ \& \ldots \&\ i_n = E_n\ then\ E_2\ else\ E_1[i_3,\ i_4, \ldots, i_n]})\end{aligned}$$

   where $\mathtt{i_j}$ with $3 \le \mathtt{j} \le \mathtt{n}$ are fresh variables.

3. (WithFunctionAcc) $(\mathtt{E_1\ with\ (E_3,\ E_4, \ldots, E_n) := E_2})$ where $\mathtt{E_1}$ is of type $(\mathtt{T_3} * \mathtt{T_4} * \cdots * \mathtt{T_n}\ \mathtt{->}\ \mathtt{T})$ is equivalent to

$$\begin{aligned}&(\mathtt{lambda(T_3,\ T_4, \ldots, T_n) : (i_3,\ i_4, \ldots, i_n) :=}\\ &\quad \mathtt{if\ i_3 = E_3\ \&\ i_4 = E_4\ \& \ldots \&\ i_n = E_n\ then\ E_2\ else\ E_1(i_3,\ i_4, \ldots, i_n)})\end{aligned}$$

   where $\mathtt{i_j}$ with $3 \le \mathtt{j} \le \mathtt{n}$ are fresh variables.

4. (WithTupleStructAcc) $(\mathtt{E_1\ with\ .M := E_2})$, where $\mathtt{E_1}$ is of tuple or struct type, is identical to $\mathtt{E_1}$ for all components except the one designated by M, which is equal to $\mathtt{E_2}$. The following formal definition uses an overloading of operator @ defined in Section 8.2.1 for tuples and in Section 8.2.2 for structs. For all steps i and k, and for all accessors K:

$$M_i((\mathtt{E_1\ with\ .M := E_2}), k)@\mathtt{K} = \begin{cases} M_i(\mathtt{E_2}, k) & \text{if } \mathtt{K} = \mathtt{M} \\ M_i(\mathtt{E_1}, k)@\mathtt{K} & \text{otherwise} \end{cases}$$

5. (WithCollectionRhsUniDim) $(\mathtt{E\ with\ A := \{R_0, R_1, \ldots, R_{n-1}\}})$ where E A is of type $\mathtt{T_{E_A}}$, which is neither a multidimensional array type nor a multivariate function type, is equivalent to:

   $(\ldots((\mathtt{E\ with\ A\ A_0 := R_0})\ \mathtt{with\ A\ A_1 := R_1}) \ldots\ \mathtt{with\ A\ A_{n-1} := R_{n-1}})$ where:

$$\mathtt{A_i} = \begin{cases} \mathtt{.i} & \text{if } \mathtt{T_{E_A}} \text{ is } \mathtt{tuple\ \{T_0, \ldots, T_{n-1}\}} \\ \mathtt{.M_i} & \text{if } \mathtt{T_{E_A}} \text{ is } \mathtt{struct\ \{M_0 : T_0, \ldots, M_{n-1} : T_{n-1}\}} \\ \mathtt{[i]} & \text{if } \mathtt{T_{E_A}} \text{ is } \mathtt{T\tilde{}(n)} \\ \mathtt{(V_i)} & \text{if } \mathtt{T_{E_A}} \text{ is } (\mathtt{T_1\ ->\ T}) \text{ and } V(\mathtt{T_1}) = \{\mathtt{V_0}, \ldots, \mathtt{V_{n-1}}\} \text{ with } \mathtt{V_i} < \mathtt{V_{i+1}} \end{cases}$$

6. (WithCollectionRhsMultiDim) (E `with A :=` $\{R_0, R_1, \ldots, R_{m_1}\}$) where E A is of type T_{E_A}, which is either a multidimensional array type or a multivariate function type, is equivalent to:

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

where:

$$
A_{i_1\ldots i_k} = \begin{cases} [i_1, \ldots, i_k] & \text{if } T_{E_A} \text{ is } T\tilde{}(m_1 + 1, \ldots, m_k + 1) \\ (V^1_{i_1}, \ldots, V^k_{i_k}) & \text{if } T_{E_A} \text{ is } (T_1 * \cdots * T_k \texttt{ -> } T), \\ & \quad V(T_{j\in[1,k]}) = \{V^j_0, \ldots, V^j_{m_j}\} \text{ and } V^j_i < V^j_{i+1} \end{cases}
$$

and $R_{i_1\ldots i_k}$ are constructed inductively:

(base) R_{i_1} for $i_1 \in [0, m_1]$ are given above (as the elements of the collection).

(step) $R_{i_1\ldots i_j}$

$$
= \begin{cases} R'_{i_j} & \text{if } R_{i_1\ldots i_{j-1}} \text{ is } \{R'_0, \ldots, R'_{m_j}\} \\ R_{i_1\ldots i_p}[i_{p+1}, \ldots, i_k] & \text{if } j = k \text{ and } R_{i_1\ldots i_{p-1}} \text{ is <collection> and} \\ & \quad R_{i_1\ldots i_p} \text{ is <expr> of type } T\tilde{}(m_{p+1} + 1, \ldots, m_k + 1) \\ R_{i_1\ldots i_p}(V^{p+1}_{i_{p+1}}, \ldots, V^k_{i_k}) & \text{if } j = k \text{ and } R_{i_1\ldots i_{p-1}} \text{ is <collection> and} \\ & \quad R_{i_1\ldots i_p} \text{ is <expr> of type } (T_{p+1} * \cdots * T_k \texttt{ -> } T) \end{cases}
$$

Static Flag

1. (WithExprStaticFlag) $\mathcal{SF}$(<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_list> ::= <pattern> { "," <pattern> }
<pattern>      ::= <expr>
                 | <named_type> ( <id> | "_" )
                 | "_"
```

Semantics

1. (CasePatternMatch) In order to formally define the pattern matching of case expressions, we define the binary function "match", with signature match : <expr_list> × <pattern_list> → <expr>, as follows:

   match($\{E_1, \ldots, E_n\}, \{P_1, \ldots, P_n\}$)

   $$= \begin{cases} \texttt{match}(\{E_1\}, \{P_1\}) \,\&\, \ldots \,\&\, \texttt{match}(\{E_n\}, \{P_n\}) & \text{if } n > 1, \text{ else} \\ \hline E_1 = P_1 & \text{if } P_1 \text{ is of the form <expr>, else} \\ \hline E_1 \texttt{ : } T & \text{if } P_1 \text{ is of the form T x where T is a <named\_type>, else} \\ \hline E_1 = E_1 & (P_1 \text{ is of the form "\_"}) \end{cases}$$

2. (CaseExpr) A case expression

   $$\begin{array}{llllll} (E_1, & E_2, & \ldots, & E_n & & \\ \mid P_{11}, & P_{12}, & \ldots, & P_{1n} & \texttt{=>} & R_1 \\ \mid P_{21}, & P_{22}, & \ldots, & P_{2n} & \texttt{=>} & R_2 \\ \vdots & \vdots & \ddots & \vdots & & \\ \mid P_{m1}, & P_{m2}, & \ldots, & P_{mn} & \texttt{=>} & R_m) \end{array}$$

   is reducible to

   $$\texttt{if } \mu_1 \texttt{ then } R'_1 \texttt{ elif } \mu_2 \texttt{ then } R'_2 \texttt{ ... elif } \mu_m \texttt{ then } R'_m \texttt{ else } N$$

   where, for $i = 1, \ldots, m$:

   - μ_i = match($\{E_1, \ldots, E_n\}$,$\{P_{i1} \ldots P_{in}\}$),
   - R'_i = (lambda $(T_1, \ldots, T_n)$: $(y_1, \ldots, y_n)$:= $R_i)(E_1, \ldots, E_n)$, where T_j is the type of E_j, and y_j is a fresh identifier — except when P_{ij} is of the form <named_type> <id>, in which case y_j is the same as <id> — and
   - N = (lambda [0] : [a] := R'_1)[0] (any expression that evaluates to **nil**ˆ(T), where T is the type of R'_1).

3. (CasePatternType) The type of a <pattern> E of the form <expr> is the same as the type of E. The type of a <pattern> of the form <named_type> (<id>|"_") is the same as the <named_type>.

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 pairwise compatible. Wildcard pattern items (_) are for this purpose considered compatible with any scalar type.
3. (CaseBranchesCompatible) The types of $\mathtt{R}_1, \ldots, \mathtt{R}_\mathtt{m}$ shall be compatible.
4. (CasePatternExprConstant) $\mathcal{SF}(\mathtt{E}) = 2$ for a <pattern> E which is an <expr>.
5. (CasePatternNotNil) For each $\mathtt{i} \in [1, \mathtt{m}]$ and $\mathtt{j} \in [1, \mathtt{n}]$, there shall exist $k \geq 0$ such that $M_k(\mathtt{P}_{\mathtt{ij}}, 0) \neq$ **nil**.
6. (CasePatternTypeSort) A pattern item <pattern> which is a <named_type> shall refer to a valid sort type.
7. (CaseCapturingVarUnicity) A <pattern_list> may not contain two <pattern>s of the form <named_type> <id> that have the same <id>.

10.7.10 Quantifier Expressions

Quantifiers in HLL operate on static domains (sets of values – much like types) to build formulas over populations of expressions. 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 quantifications 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 the result is **nil**.

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

The operator may be provided with a *default* value in case there is no value in the domain satisfying the predicate. For example, with a default value of -1, we get:

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

The `SELECT` operator can also select unique N-tuples by specifying N domains to select from. Here is an example with $N = 2$:

```
SELECT i:[0,2], j:[0,2]
  (A[i] = 1 & A[j] = 4):  (nil nil),  (0 2), (nil nil), (nil nil), ...
```

The default value in the case of multiple domains can be given as a collection:

```
SELECT i:[0,2], j:[0,2]
  (A[i] = 1 & A[j] = 4,
              {-1, -1}):  (-1 -1),  (0 2), (nil nil), (-1 -1), ...
```

The type of SELECT expressions deserves to be mentioned. In the case of a single domain it is a scalar type compatible with the underlying type of the domain. In the case of *N* domains the type is a tuple with *N* components where each component is a scalar type compatible with the underlying type of the corresponding domain. Finally, in the presence of a default value, the type is the union type between the previously mentioned type and the type of the default value. In the case where the default value is a collection the union type is defined by the special case (UnionTupleCollection) in Section 8.4.3.

Quantification may also be performed over (the components of) array and function expressions, using the operator `$items`. As an example, `ALL a:$items(A) (a < 4)` is equivalent to `ALL i:[0,2] (A[i] < 4)` above. Semantically, `$items` behaves as a *multiset*. In the example

```
SUM a:$items(A) (a): 0,     5,   nil,     7, ...
```

note that the value at time step 3 is 7 (not 4) since the value 3 has been added twice.

When applying the SELECT operator to a domain of the form `$items(A)`, it selects, for each time step, (the value of) one of the components (or items) of `A`. This means that `SELECT a:$items(A) (a = 1)` is equivalent to `A[SELECT i:[0,2] (A[i] = 1)]`:

```
SELECT a:$items(A) (a = 1): nil,     1,   nil,     1, ...
```

The multiset semantics can require a bit of extra care when combining SELECT with `$items`. For example:

```
SELECT a:$items(A) (a = 3): nil,     nil,   nil,     nil, ...
```

At time step 3, even though the array values are only 1 and 3, the result is **nil** since the value 3 is repeated twice in the array.

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" "(" <path_id> ")"

<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 expressions, 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.[21]

   Although the quantifier scope includes the default value (a <rhs>) of a SELECT expression, the default value is prohibited from referencing any of the quantifier variables due to the restriction (SelectQuantDefaultGround) below.

2. (QuantDomainDomain) A quantifier domain D which is a <domain> corresponds for each time step k to the set of values 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.

3. (QuantDomainItems) QTF v : $items(V) (E), where QTF is a <quantifier> other than SELECT and V is of array type T^($D_1, \ldots, D_n$) is equivalent to: QTF i_1 : [0, $K_1 - 1$], ..., i_n : [0, $K_n - 1$] (E<v := W[$i_1, \ldots, i_n$]>) where W is a fresh variable referring to V, the K_i are fresh constants with $K_i = D_i$, and the i_j are fresh identifiers. If, on the other hand, V is of function type ($T_1 * \cdots * T_n$ -> T) then the expression is equivalent to: QTF $i_1 : N_1, \ldots, i_n : N_n$ (E<v := W($i_1, \ldots, i_n$)>) where W is a fresh variable referring to V, the N_i are fresh named types using the corresponding T_i as the defining type, and the i_j are fresh identifiers.

[21] 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.

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 of the form `$items(E)`.
5. (QuantMultVar) `QTF` $\mathtt{i}_1 : \mathtt{D}_1$, $\mathtt{i}_2 : \mathtt{D}_2, \ldots, \mathtt{i}_n : \mathtt{D}_n$ `(E)`, where `QTF` is a `<quantifier>` other than `SELECT` and the $\mathtt{i}_i : \mathtt{D}_i$ are `<quantif_var>`, is reducible to `QTF` $\mathtt{j}_1 : \mathtt{D}_1$ (`QTF` $\mathtt{j}_2 : \mathtt{D}_2 \ldots$ (`QTF` $\mathtt{j}_n : \mathtt{D}_n$ (`E<`$\mathtt{i}_1 := \mathtt{j}_1, \mathtt{i}_2 := \mathtt{j}_2, \ldots \mathtt{i}_n := \mathtt{j}_n$`>`)) $\ldots$) 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 x:D (E)`, where `D` is a `<domain>`, is an expression of type `bool` that evaluates to true in steps i and k *iff* there exists some value `V` in `D` such that `E`, with all occurrences of `x` replaced by `V`, evaluates to true. If `D` is empty the quantification evaluates to false. Formally:

$$M_i(\mathtt{SOME\ x : D\ (E)}, k) = \begin{cases} \text{true} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \text{true} \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \text{false} & \text{otherwise} \end{cases},$$

   where the cases on the right hand side should be considered in order from top to bottom.
9. (QuantAll) `ALL x:D (E)`, where `D` is a `<domain>`, is an expression of type `bool` that evaluates to true in steps i and k *iff* for all values `V` in `D`, `E`, with all occurrences of `x` replaced by `V`, evaluates to true. If `D` is empty the quantification evaluates to true. Formally:

$$M_i(\mathtt{ALL\ x : D\ (E)}, k) = \begin{cases} \text{false} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \text{false} \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \text{true} & \text{otherwise} \end{cases},$$

   where the cases on the right hand side should be considered in order from top to bottom.
10. (QuantSum) `SUM x:D (E)`, where `D` is a `<domain>`, is an expression of type `int` that evaluates to the sum of all $\mathtt{E}_\mathtt{x}$. Formally:

$$M_i(\mathtt{SUM\ x : D\ (E)}, k) = \begin{cases} 0 & \text{if } \mathtt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \sum_{\mathtt{V} \in \mathtt{D}_k} M_i(\mathtt{E<x := V>}, k) & \text{otherwise} \end{cases}$$

11. (QuantProd) `PROD x:D (E)`, where `D` is a `<domain>`, is an expression of type `int` that evaluates to the product of all $\mathtt{E}_\mathtt{x}$. Formally:

$$M_i(\mathtt{PROD\ x : D\ (E)}, k) = \begin{cases} 1 & \text{if } \mathtt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \prod_{\mathtt{V} \in \mathtt{D}_k} M_i(\mathtt{E<x := V>}, k) & \text{otherwise} \end{cases}$$

12. (QuantMin) `$min x:D (E)`, where `D` is a `<domain>`, is an expression of type `int` that selects the minimum of all $\mathtt{E}_x$. Formally:

    $M_i(\$\mathtt{min}\ \mathtt{x} : \mathtt{D}\ (\mathtt{E}), k)$

    $$= \begin{cases} \textbf{nil} & \text{if } \mathtt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \min_{\mathtt{V} \in \mathtt{D}_k} M_i(\mathtt{E<x := V>}, k) & \text{otherwise} \end{cases}$$

13. (QuantMax) `$max x:D (E)`, where `D` is a `<domain>`, is an expression of type `int` that selects the maximum of all $\mathtt{E}_x$. Formally:

    $M_i(\$\mathtt{max}\ \mathtt{x} : \mathtt{D}\ (\mathtt{E}), k)$

    $$= \begin{cases} \textbf{nil} & \text{if } \mathtt{D}_k = \emptyset \\ \textbf{nil} & \text{if } \exists \mathtt{V} \in \mathtt{D}_k\ M_i(\mathtt{E<x := V>}, k) = \textbf{nil} \\ \max_{\mathtt{V} \in \mathtt{D}_k} M_i(\mathtt{E<x := V>}, k) & \text{otherwise} \end{cases}$$

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 `T` of `SELECT` $\mathtt{i}_1 : \mathtt{D}_1,\ \mathtt{i}_2 : \mathtt{D}_2, \ldots, \mathtt{i}_n : \mathtt{D}_n$ `(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:

    $$\mathtt{T}_i = \begin{cases} \text{unsized copy of } \mathtt{D}_i & \text{if } \mathtt{D}_i \text{ is a } \mathtt{<type\_domain>} \\ \mathtt{int} & \text{if } \mathtt{D}_i \text{ is a } \mathtt{<range>} \\ \text{unsized copy of } \mathtt{T}' & \text{if } \mathtt{D}_i \text{ is } \$\mathtt{items}(\mathtt{E}), \text{ where } \mathtt{T}' \\ & \quad \text{is the component type of } \mathtt{E} \end{cases}$$

    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. (QuantSelectImage) The *image* $\Delta_i(k)$ of `SELECT` $\mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ `(E [,R])`, at meta step i and time step k, is defined as[22] $\Delta_i(k) = V_i(\mathtt{D}_1, k) \times \cdots \times V_i(\mathtt{D}_n, k)$, where the $V_i(\mathtt{D}_j, k)$ are sets of values defined by

    $$V_i(\mathtt{D}_j, k) = \begin{cases} V(\mathtt{T}) & \text{if } \mathtt{D}_j \text{ is a } \mathtt{<type\_domain>} \text{ with} \\ & \quad \text{underlying type } \mathtt{T} \\ [\mathtt{a}, \mathtt{b}] & \text{if } \mathtt{D}_j \text{ is a range } [\mathtt{a}, \mathtt{b}] \\ \mathrm{Im}(M_i(\mathtt{F}_j, k)) & \text{if } \mathtt{D}_j \text{ has the form } \$\mathtt{items}(\mathtt{F}_j) \end{cases}$$

    Given $\mathtt{D}_1, \ldots, \mathtt{D}_n$, the image is the set of values that a select expression of the form `SELECT` $\mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ `( ... [,...])` can attain.

If $v = (v_1, \ldots, v_n) \in \Delta_i(k)$, then $(\mathtt{W}_1, \ldots, \mathtt{W}_n)$ is a *preimage* of v if it has the form

$$\mathtt{W}_j = \begin{cases} v_j & \text{if } \mathtt{D}_j \text{ is a <domain>} \\ \mathtt{F}'_j(u_{j1}, \ldots, u_{jm_j}) & \text{if } \mathtt{D}_j \text{ has the form \$items}(\mathtt{F}_j) \\ & \quad \text{with } \mathtt{F}_j \text{ of function type} \\ \mathtt{F}'_j[u_{j1}, \ldots, u_{jm_j}] & \text{if } \mathtt{D}_j \text{ has the form \$items}(\mathtt{F}_j) \\ & \quad \text{with } \mathtt{F}_j \text{ of array type} \end{cases}$$

where $\mathtt{F}'_j$ is a fresh variable referring to $\mathtt{F}_j$ and the tuple $(u_{j1}, \ldots, u_{jm_j})$ belongs to the domain of $M_i(\mathtt{F}_j, k)$, and satisfies $M_i(\mathtt{W}_j, k) = v_j$ for $j = 1, \ldots, n$.

17. (QuantSelect) At each time step, the selection expression `SELECT` $\mathtt{x}_1 : \mathtt{D}_1,\ \mathtt{x}_2 : \mathtt{D}_2, \ldots, \mathtt{x}_n : \mathtt{D}_n$ (`E` [, `R`]) of type `T`, as given according to (QuantSelectType), selects the unique value (if `n` = 1), or tuple value (if `n` > 1), for which the Boolean expression `E` evaluates to true, given that such a value exists and is unique. If no such value exists, then the value of the select expression is given by the default (if present). In all other cases, the value of the select expression is **nil**$\hat{}$(`T`).

    If the default `R` is present in the `SELECT` expression, let `R`′ denote[23] the expression ((`lambda[1] : [i] := Q`) `with [0] := R`)`[0]`, where `Q` is a fresh variable of type `T`. Also, let

$$\sigma_i(k) = M_i(\texttt{SUM } \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n \texttt{ (if E then 1 else 0)}, k).$$

    Then

$$M_i(\texttt{SELECT } \mathtt{x}_1 : \mathtt{D}_1,\ \mathtt{x}_2 : \mathtt{D}_2, \ldots, \mathtt{x}_n : \mathtt{D}_n \texttt{ (E [, R])}, k)$$
$$= \begin{cases} \mathbf{nil}\hat{}(\mathtt{T}) & \text{if } \sigma_i(k) \notin \{0, 1\} \text{ or } \mathtt{R} \text{ is } \textit{not} \text{ present and } \sigma_i(k) = 0 \\ M_i(\mathtt{R}', k) & \text{if } \mathtt{R} \text{ is present and } \sigma_i(k) = 0 \\ v & \text{for the unique } v = (v_1, \ldots, v_n) \in \Delta_i(k) \text{ satisfying} \\ & \quad M_i(\mathtt{E}\texttt{<}\mathtt{x}_1\texttt{:=}\ \mathtt{W}_1, \ldots, \mathtt{x}_n\texttt{:=}\ \mathtt{W}_n\texttt{>}, k) = \text{true}, \\ & \quad \text{where } (\mathtt{W}_1, \ldots, \mathtt{W}_n) \text{ is a preimage of } v. \end{cases}$$

    The cases on the right hand side should be considered in order from top to bottom (even if the first condition is true, there can still be a unique v satisfying the last condition due to multiset semantics).

Static Flag

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

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

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. (QuantDomainNotNil) Quantification is not allowed over a domain taking the value **nil** in any time step (see (DomainAsRange)).
4. (QuantDomainStatic) $\mathcal{SF}(\texttt{D}) \geq 1$ for a `<quantif_var> i : D`, where `D` is a `<domain>`.
5. (ItemsOperandArrayOrFunction) The operand `V` of `$items(V)` must be of finite-dimensional array or function type.
6. (BoolQuantOperandBool) The operand `E` of `QTF V (E)` shall be of type `bool` if `QTF` $\in$ {`SOME`, `ALL`}.
7. (IntQuantOperandInt) The operand `E` of `QTF V (E)` shall be of integer type if `QTF` $\in$ {`SUM`, `PROD`, `$min`, `$max`}.
8. (SelectQuantOperandBool) The operand `E` of `SELECT` $\texttt{V}_1, \ldots, \texttt{V}_\texttt{n}$ (`E` [, `R`]) shall be of type `bool`.
9. (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).
10. (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 for example not allowed: `SELECT i:[0,10] (false, i)`.

[22]Here we adopt the mathematical standard convention that if $n = 1$, then the Cartesian product $A_1 \times \cdots \times A_n$ is defined to be A_1 (instead of the set of 1-tuples with elements in A_1).

[23]This is to handle the case when `R` is a collection. In that case `R'` is the corresponding tuple.

11 Declarations

Declarations declare variables with a name and a type. The declared variable 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[24] `<input>`) declares a variable with name and type (causing it to exist in the scope of the declaration) and makes it visible everywhere in the scope (of the namespace of expressions) of the declaration, regardless of the position of the declaration.

2. (DeclMultInline) `T` D_1, $D_2, \ldots, D_n$; is equivalent to the `n` declarations `T` D_1; `T` D_2; ...`T` D_n; regardless of whether the D_i are of the form `<declarator>` or `<input_declarator>`.

3. (DeclNormal) A *Normal declaration* (`<declaration>`) of the form `[T] D` declares a 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>`) of 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>`) of the form `[T] I(D)` is called an *Initial input declaration* and is reducible to a 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 variable may not be declared more than once per scope of the namespace of expressions. This also means that the same variable cannot be declared once with a normal declaration and once with an input declaration, in the same scope.

2. (InputsFinite) The resulting type (as calculated by `calc_type` from the base type and the declarator) of an input declaration shall be finite-dimensional.

[24] 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).

3. (InputsNonEmpty) The resulting type `T` (as calculated by `calc_type` from the base type and declarator) of an input declaration shall be one of the following:
   (a) a non-empty scalar type (`|T| > 0`),
   (b) a composite type with no components (*e.g.* due to an empty function domain), or
   (c) a composite type with only non-empty component types.
4. (InputsUndefined) A variable declared using an input declaration may not be defined, except for initial inputs as specified in (DeclInitialInputDefNext).
5. (DeclInitialInputDefNext) A variable declared using an initial input declaration shall be defined with, and only with, a next definition (see Section 12).
6. (UndefinedSized) A variable 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 variables on the left hand side of the definition symbol := using an expression or a collection on the right hand side. If the variables are undeclared they become implicitly declared by the definition, but only if the type of the right hand side is scalar.

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
```

Multiple definitions of the same variable are not allowed (even though latch definitions can be split into two parts, see semantics below). Definitions are not required to be well-founded (even circular definitions are allowed), but ill-founded definitions are handled using the value **nil**. Implementations of HLL may choose to forbid circular definitions. Example:

```
Definitions:
 z := x + y; // x and z are not well-founded and will carry nil
 x := z - y; // in each time step
 y := 10;
 y := 4;     // Not allowed to define y more than once
 X(w) := w;  // Allowed: w keeps its value in the next cycle
```

Recursion is allowed in HLL, and ill-founded recursions will result in the value **nil**:

```
Declarations:
 int fibonacci(int);
 int infinite(int);
 int circular(int);
Definitions:
 // fibonacci(i) is a well-founded recursion:
 fibonacci(i) := if i <= 2 then 1 else fibonacci(i - 1) +
                                       fibonacci(i - 2);

 // infinite(i) is a non-terminating recursion, hence ill-founded:
 infinite(i) := infinite(i + 1);

 // circular(i) is a circular recursion, hence ill-founded:
 circular(i) := circular(i);
```

To handle recursion and give a clear semantics of defined variables in case of circularity and other cases of ill-founded sets of definitions, we will rely upon the meta step *i* mentioned in Section 2.3.2. The idea is to initialize all defined variables to the value **nil** at the initial

meta step, for all time steps. Then, during a sequence of meta steps, the correct values for the defined variables are inferred and eventually a fixed point, with a final value for each time step, is reached, except in the case of non-terminating recursions: in those cases the impacted variable (or components) and time steps will keep the value **nil** forever.

The use of the meta step *i* can be observed in the semantic items (DefAlways), (DefInit) and (DefNext) below.

Syntax

(DefinitionSyntax)

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

Semantics

1. (Definition) A *Definition* (`<definition>`) defines one or several variables on the *Left hand side* (`<lhs>`) using an expression or collection on the *Right hand side* (`<rhs>`). Latches may be defined using a comma-separated pair of `<rhs>`, see (DefLatch).

2. (DefUndeclared) If a 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. The implicit declaration causes the variable to exist and to be visible everywhere in the scope (of the namespace of expressions) of the definition, regardless of the position of the definition. The type of the variable is inferred according to (DefUndeclaredType) below.

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

Definition	**Condition**	`T`
`X(V) := E;`	(none)	`bool`
`V :=` E_1`,` E_2`;`	(none)	`bool`
`V := E;`	`V` is defined (directly or indirectly) in terms of itself in a previous time step	`bool`
	otherwise	T_E

4. (DefAlways) `V := E;` (always definition) where `V` is a variable of type `T` and `E` is an expression of a type assignable to `T`, defines the value of `V` in all time steps, using the expression `E`. Formally, for all meta steps i and time steps k:

$$M_0(\mathtt{V}, k) = \mathbf{nil}\hat{}(\mathtt{T})$$
$$M_{i+1}(\mathtt{V}, k) = D(\mathtt{T}, M_i(\mathtt{E}, k))$$

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 a type assignable to `T`. Formally, for all meta steps i:

$$M_0(\mathtt{V}, 0) = \mathbf{nil}\hat{}(\mathtt{T})$$
$$M_{i+1}(\mathtt{V}, 0) = D(\mathtt{T}, M_i(\mathtt{E}, 0))$$

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 a type assignable to `T`. Formally, for all meta steps i and time steps k:

$$M_0(\mathtt{V}, k+1) = \mathbf{nil}\hat{}(\mathtt{T})$$
$$M_{i+1}(\mathtt{V}, k+1) = D(\mathtt{T}, M_i(\mathtt{E}, k))$$

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;` together with a declaration for `V'` which 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;` together with a declaration for `V'` which 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 (except in the more complex case, handled by the formal definition below, where the `<lhs>` is of multidimensional array type or multivariate function type). 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 $\mathtt{V} := ((\mathtt{lambda[1]} : \mathtt{[i]} := \mathtt{V'})\ \mathtt{with}\ \mathtt{[0]} := \{\mathtt{R_1}, \ldots, \mathtt{R_n}\})\mathtt{[0]};$ together with a declaration for `V'` which is a fresh variable with the same type as `V`.

    The purpose of this rather cryptic translation is to reuse the semantic items (WithCollectionRhsUniDim) and (WithCollectionRhsMultiDim) of Section 10.7.8 for the definition at hand.

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

12. (DefWildcard) `_ := <rhs>;` is reducible to the empty string.

13. (DefUnfolding) An *Unfolding definition* is a definition where several variables or wildcards ("_") occur comma-separated on the left hand side (using an <unfolding>). If the composite type of the right hand side has n ordered components, then such a definition, with k variables and l wildcards where $n = k + l$, on the left hand side is equivalent to k ordinary definitions of the (DefAlways) kind, 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. The restrictions on the left- and right-hand sides are specified in (DefUnfoldingCompatibleRhs).

Static Flag

1. (CollectionStaticFlag) $\mathcal{SF}$(<collection>) = 0
2. (DefAlwaysStaticFlag) $\mathcal{SF}(\texttt{V}) = \texttt{min}(1, \mathcal{SF}(\texttt{E}))$ for `V := E;`
3. (DefLatchStaticFlag) $\mathcal{SF}(\texttt{V}) = 0$ for `X(V) := E;` or `V := E1, E2;`

Restrictions

1. (DefUnicity) A variable may not have its value be defined more than once per time step.

   This means that it is allowed to combine an initial definition with a next definition, but not to combine an always definition with either an initial or a next definition.

2. (DefCompleteness) A 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.

3. (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.

4. (DefRhsTypeAssignableToLhsType) The type of the <rhs> shall be assignable to the type of the <lhs>. For this purpose, the type of a <lhs> of 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).

5. (LatchesSized) A variable 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).

6. (DefUnfoldingCompatibleRhs) When the left hand side (<lhs>) of a definition is of the form <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.
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) $\mathcal{SF}$(`C`) = 2 for a `<constant> T C := E;`.

Restrictions

1. (ConstantDefRhsConstant) $\mathcal{SF}$(`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

Constraints can be used to remove (or disallow) models of an HLL text. For example, the contraint `x` would remove all models where `x` is not always true, and the constraint `I(x)` would remove all models where `x` is not true in the initial time step. Please refer to Sections 2.3.3 and 2.3.4 for a better understanding of constraints.

Syntax

(ConstraintSyntax)

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

Semantics

1. (ConstraintAlways) An *Always constraint* `E;` corresponds to the proposition $\square$`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;` where the expression `E` is of type `bool` corresponds to the proposition `□E`.
2. (PoComposite) A proof obligation `E;` where the expression `E` is 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).[25] 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) An allowed type for a proof obligation is either `bool`, or an array or finite-dimensional function type with `bool` as its component type.

   A proof obligation shall be of an allowed type.

[25] See Sections 2.3.3 and 2.3.4 for the definition of the consequence relation.

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 this 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 of finite-dimensional type.

A Syntax Overview

This appendix contains the complete grammar for HLL.

```
<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> ";"

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

```
                  | <integer>
                  | <tuple>
                  | <structure>
                  | <array>
                  | <function>
                  | <named_type>
<bool>          ::= "bool"
<integer>       ::= "int"
                  | "int" <sign>
                  | "int" <range>
<sign>          ::= "signed" <id_or_int>
                  | "unsigned" <id_or_int>
<id_or_int>     ::= <path_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> ";"
                  | <unfolding> ":=" <rhs> ";"
                  | "_" ":=" <rhs> ";"
                  | "I" "(" <lhs> ")" ":=" <rhs> ";"
                  | "X" "(" <lhs> ")" ":=" <rhs> ";"
                  | <lhs> ":=" <rhs> "," <rhs> ";"

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

```
<rhs>           ::= <expr>
                  | <collection>
<collection>    ::= "{" <rhs> {"," <rhs>} "}"

<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>
                  | <integer>
<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>   ::= regexp: [0-9](_?[0-9])*
```

```
<bin_literal>   ::= regexp: 0[Bb][0-1](_?[0-1])*
<hex_literal>   ::= regexp: 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_list>  ::= <pattern> { "," <pattern> }
<pattern>       ::= <expr>
                  | <named_type> ( <id> | "_" )
                  | "_"

<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" "(" <path_id> ")"

<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"
```

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

The operator `:` (`<membership_expr>`) has the same precedence as the operator `=`, and is left-associative, as according to (MembershipPrecedence).

B Reserved Words

The following words cannot be used as identifiers:

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

C Type System Overview

The table below summarizes the types of all expressions and their operands.

Expression	*Operands*	*Type*
`if` E_1 `then` E_2 `else` E_3	E_1 : `bool` E_2 : any type T_2 E_3 : any type T_3 compatible with T_2	The union type of T_2 and T_3
`lambda DS : FP := E`	`E` : any type	As defined by (LambdaType)
E_1 `#` E_2	E_1 : `bool` E_2 : `bool`	`bool`
E_1 `&` E_2	E_1 : `bool` E_2 : `bool`	`bool`
E_1 `#!` E_2	E_1 : `bool` E_2 : `bool`	`bool`
E_1 `->` E_2	E_1 : `bool` E_2 : `bool`	`bool`
E_1 `<->` E_2	E_1 : `bool` E_2 : `bool`	`bool`
E_1 `>` E_2	E_1 : any integer type E_2 : any integer type	`bool`
E_1 `>=` E_2	E_1 : any integer type E_2 : any integer type	`bool`
E_1 `<` E_2	E_1 : any integer type E_2 : any integer type	`bool`
E_1 `<=` E_2	E_1 : any integer type E_2 : any integer type	`bool`
E_1 `=` E_2	E_1 : any finite-dimensional type `T` E_2 : any type compatible with `T`	`bool`
E_1 `==` E_2	E_1 : any finite-dimensional type `T` E_2 : any type compatible with `T`	`bool`
E_1 `!=` E_2	E_1 : any finite-dimensional type `T` E_2 : any type compatible with `T`	`bool`
E_1 `<>` E_2	E_1 : any finite-dimensional type `T` E_2 : any type compatible with `T`	`bool`
E_1 `+` E_2	E_1 : any integer type E_2 : any integer type	`int`
E_1 `-` E_2	E_1 : any integer type E_2 : any integer type	`int`
E_1 `*` E_2	E_1 : any integer type E_2 : any integer type	`int`

Expression	*Operands*	*Type*
E_1 ^ E_2	E_1 : any integer type E_2 : any integer type	int
E_1 « E_2	E_1 : any integer type E_2 : any integer type	int
E_1 » E_2	E_1 : any integer type E_2 : any integer type	int
E_1 / E_2	E_1 : any integer type E_2 : any integer type	int
E_1 /> E_2	E_1 : any integer type E_2 : any integer type	int
E_1 /< E_2	E_1 : any integer type E_2 : any integer type	int
E_1 % E_2	E_1 : any integer type E_2 : any integer type	int
E : D	E : any scalar type T D : any type compatible with T	bool
~E	E : bool	bool
-E	E : any integer type	int
E A	E : any composite type T A : an accessor compatible with T	The type of the component of T designated by A
(E)	E : any type T	T
<bool_literal>		bool
<int_literal>		int
X(E)	E : any type T	T
pre(E)	E : any type T	The unsized copy of T
pre(E_1, E_2)	E_1 : any type T_1 E_2 : any type T_2 compatible with T_1	The union type of T_1 and T_2
pre<T>(E)	E : any type assignable to T	T
pre<T>(E_1, E_2)	E_1 : any type assignable to T E_2 : any type assignable to T	T
$min($E_1$, E_2)	E_1 : any integer type E_2 : any integer type	int
$max($E_1$, E_2)	E_1 : any integer type E_2 : any integer type	int
$abs(E)	E : any integer type	int
$or($E_1$, E_2)	E_1 : any integer type E_2 : any integer type	int

Expression	*Operands*	*Type*
`$and(`E_1`, `E_2`)`	E_1 : any integer type E_2 : any integer type	`int`
`$xor(`E_1`, `E_2`)`	E_1 : any integer type E_2 : any integer type	`int`
`$not(E)`	`E` : any integer type	`int`
`bin2u(`E_1`, `E_2`)`	E_1 : `bool^(N)` for an integer `N` E_2 : any integer type	`int`
`u2bin(`E_1`, `E_2`)`	E_1 : any integer type E_2 : any integer type	`bool^(`E_2`)`
`bin2s(`E_1`, `E_2`)`	E_1 : `bool^(N)` for an integer `N` E_2 : any integer type	`int`
`s2bin(`E_1`, `E_2`)`	E_1 : any integer type E_2 : any integer type	`bool^(`E_2`)`
`population_count_lt(`$E_1, \ldots, E_n, K$`)`	E_i : `bool` `K` : any integer type	`bool`
`population_count_gt(`$E_1, \ldots, E_n, K$`)`	E_i : `bool` `K` : any integer type	`bool`
`population_count_eq(`$E_1, \ldots, E_n, K$`)`	E_i : `bool` `K` : any integer type	`bool`
`cast<T>(E)`	`E` : any integer type	`int`

`(E with A :=` `R)`	`E` : any composite type `T`
	`A` : an accessor compatible with `T`
	`R` : any type assignable to the type of `E A`
		`T`

`(`E_1`, `E_2`, ..., `E_n `|`P_{11}`, `P_{12}`, ..., `P_{1n}` => `R_1 `|`P_{21}`, `P_{22}`, ..., `P_{2n}` => `R_2 $\vdots \quad \ddots \quad \vdots$ `|`P_{m1}`, `P_{m2}`, ..., `P_{mn}` => `R_m`)`	E_j : any scalar type T_j P_{ij} : any type compatible with T_j R_i : any type U_i	The union type of $U_1 \ldots U_m$
`SOME` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : `bool`	`bool`
`ALL` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : `bool`	`bool`
`DISJ` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : `bool`	`bool`
`CONJ` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : `bool`	`bool`
`SUM` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : any integer type	`int`

Expression	***Operands***	***Type***
`PROD` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : any integer type	`int`
`$min` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : any integer type	`int`
`$max` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : any integer type	`int`
`SELECT` $i_1 : D_1, \ldots, i_n : D_n$ `(E)`	D_i : any finite scalar type `E` : `bool`	As defined by (QuantSelectType)
`SELECT` $i_1 : D_1, \ldots, i_n : D_n$ `(E, R)`	D_i : any finite scalar type `E` : `bool` `R` : any type compatible with T_{val} of (QuantSelectType)	As defined by (QuantSelectType)
`$items(E)`	`E` : any finite-dimensional array or function type	*Not applicable*

D Sources and Absorption of Nil

D.1 Sources of Nil

The following table lists the possible sources of **nil**, together with the condition necessary for its appearance in each case. Some language constructs which are very closely related ("syntactic sugar") to one of the constructs listed in the table may have been omitted.

In the table below, if v is a value, we use the notation $v \ni$ **nil** to express that at least one scalar leaf in v has the value **nil** (if v happens to be scalar, it simply means that $v =$ **nil**).

Language Construct	*Condition*	*References*
$\mathtt{E_1 / E_2}$	$M(\mathtt{E_1 / E_2}, k) =$ **nil** if $M(\mathtt{E_2}, k) = 0$	(IntDiv)
$\mathtt{E_1 /> E_2}$	$M(\mathtt{E_1 /> E_2}, k) =$ **nil** if $M(\mathtt{E_2}, k) = 0$	(IntFloorDiv) (IntDiv)
$\mathtt{E_1 /< E_2}$	$M(\mathtt{E_1 /< E_2}, k) =$ **nil** if $M(\mathtt{E_2}, k) = 0$	(IntCeilDiv) (IntDiv)
$\mathtt{E_1 \% E_2}$	$M(\mathtt{E_1 \% E_2}, k) =$ **nil** if $M(\mathtt{E_2}, k) = 0$	(IntRem) (IntDiv)
$\mathtt{E_1 \hat{} E_2}$	$M(\mathtt{E_1 \hat{} E_2}, k) =$ **nil** if $M(\mathtt{E_1}, k) = 0$ and $M(\mathtt{E_2}, k) < 0$	(IntExp)
$\mathtt{E[E_1, \ldots, E_n]}$	$M(\mathtt{E[E_1, \ldots, E_n]}, k) =$ **nil**$\hat{}(\mathtt{T})$ if $\mathtt{T_E} = \mathtt{T\tilde{}(D_1, \ldots, D_n)}$ and $\exists \mathtt{i}\ M(\mathtt{E_i}, k) \notin [\mathtt{0}, \mathtt{D_i} - \mathtt{1}]$	(ProjArrayAcc)
$\mathtt{E(E_1, \ldots, E_n)}$	$M(\mathtt{E(E_1, \ldots, E_n)}, k) =$ **nil**$\hat{}(\mathtt{T})$ if $\mathtt{T_E} = (\mathtt{T_1} * \cdots * \mathtt{T_n}$ -> $\mathtt{T})$ and $\exists \mathtt{i}\ M(\mathtt{E_i}, k) \notin V(\mathtt{T_i})$	(ProjFunctionAcc)
`pre<T>(E)`	$M($`pre<T>(E)`$, k) =$ **nil**$\hat{}(\mathtt{T})$ if $k = 0$	(PreTyped)
`pre(E)`	$M($`pre(E)`$, k) =$ **nil**$\hat{}(\mathtt{T_E})$ if $k = 0$	(PreUntyped) (PreTyped)

Language Construct	***Condition***	***References***
<case_expr> = (E$_1$, E$_2$, ..., E$_n$ |P$_{11}$, P$_{12}$, ..., P$_{1n}$ => R$_1$ |P$_{21}$, P$_{22}$, ..., P$_{2n}$ => R$_2$ ⋮ ⋱ ⋮ |P$_{m1}$, P$_{m2}$, ..., P$_{mn}$ => R$_m$)	$M(\text{<case\_expr>}, k) = \textbf{nil}\hat{}(\mathtt{T})$ if $\forall \mathtt{i}\ M(\text{match}(\{\mathtt{E}_1, \ldots, \mathtt{E}_n\},$ $\{\mathtt{P}_{i1}, \ldots, \mathtt{P}_{in}\}), k) = \text{false}$	(CaseExpr) (CasePattern-Match)
SELECT x$_1$: D$_1$, ..., x$_n$: D$_n$ (E)	$M(\mathtt{SELECT}\ \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n\ (\mathtt{E}), k) = \textbf{nil}\hat{}(\mathtt{T})$ if $M(\mathtt{SUM}\ \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ $(\mathtt{if\ E\ then\ 1\ else\ 0}), k) \neq 1$	(QuantSelect)
SELECT x$_1$: D$_1$, ..., x$_n$: D$_n$ (E, R)	$M(\mathtt{SELECT}\ \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n\ (\mathtt{E}, \mathtt{R}), k) = \textbf{nil}\hat{}(\mathtt{T})$ if $M(\mathtt{SUM}\ \mathtt{x}_1 : \mathtt{D}_1, \ldots, \mathtt{x}_n : \mathtt{D}_n$ $(\mathtt{if\ E\ then\ 1\ else\ 0}), k) \notin \{0, 1\}$	(QuantSelect)
V := E;	$M(\mathtt{V}, k) \ni \textbf{nil}$ if $M(\mathtt{E}, k) \notin V^*(\mathtt{T})$	(DefAlways)
I(V) := E;	$M(\mathtt{V}, k) \ni \textbf{nil}$ if $k = 0$ and $M(\mathtt{E}, 0) \notin V^*(\mathtt{T})$	(DefInit)
X(V) := E;	$M(\mathtt{V}, k) \ni \textbf{nil}$ if $k > 0$ and $M(\mathtt{E}, k-1) \notin V^*(\mathtt{T})$	(DefNext)
V := E$_1$, E$_2$;	$M(\mathtt{V}, k) \ni \textbf{nil}$ if $k = 0$ and $M(\mathtt{E}_1, 0) \notin V^*(\mathtt{T})$ or $k > 0$ and $M(\mathtt{E}_2, k-1) \notin V^*(\mathtt{T})$	(DefLatch) (DefInit) (DefNext)

D.2 Absorption of Nil

The following table lists the expressions which may absorb **nil**.

Expression	*Operands Which May Be (Partially or Wholly) Absorbed*	*Operands Which May Cause Nil To Be Absorbed*
if E_1 then E_2 else E_3	E_2, E_3	E_1
E_1 # E_2	E_1, E_2	E_1, E_2
E_1 & E_2	E_1, E_2	E_1, E_2
E_1 -> E_2	E_1, E_2	E_1, E_2
X(E)	E	
(E with $A_1, \ldots, A_n$:= R)	E, R	$A_1, \ldots, A_n$
(E_1, E_2, ..., E_n |P_{11}, P_{12}, ..., P_{1n} => R_1 |P_{21}, P_{22}, ..., P_{2n} => R_2 ⋮ ⋱ ⋮ |P_{m1}, P_{m2}, ..., P_{mn} => R_m)	$R_1, R_2, \ldots, R_m$	$E_1, E_2, \ldots, E_n$ $P_{11}, P_{12}, \ldots, P_{(m-1)n}$
SOME $i_1 : D_1, \ldots, i_n : D_n$ (E)	E	E
ALL $i_1 : D_1, \ldots, i_n : D_n$ (E)	E	E
DISJ $i_1 : D_1, \ldots, i_n : D_n$ (E)	E	E
CONJ $i_1 : D_1, \ldots, i_n : D_n$ (E)	E	E

E Breaking Changes and Deprecations

The following appendix lists the non-backwards compatible changes (breaking changes) to HLL since previously released versions of this document, and the possible future such changes (deprecations).

Deprecated language constructs in this version of the document:

1. A reference to a top-level user namespaces from within another user namespace using a path identifier with a relative path is ***deprecated***. See (PathRelative).

Breaking changes since version 2.7 of the document:

1. Undeclared variables which are recursively defined will be interpreted as being of type `bool`, as defined by (DefUndeclaredType), whereas in version 2.7 their type, if scalar, was inferred from the defining expression, when possible.
2. The semantics of (DefAlways) has changed so that the value **nil** is taken whenever the value of the right hand side does not belong to to the type of the left hand side (for example in the case of integer overflow).
3. The compatibility and assignability relations on functions have changed to take into consideration the function domains (they must now be equal). See (FunctionCompatibility), (FunctionAssignability) and (FunctionDomainEquality).
4. The static flag of case capturing variables has changed from 0 to 1. See (CaseExpr) and (FormalParamStaticFlag).
5. An "overhang" of the declaration suffix of a lambda expression is no longer allowed, by a modification of the restriction (LambdaParamsBound) enforcing $|\texttt{DS}| = |\texttt{FP}|$ where previously was allowed $|\texttt{DS}| \geq |\texttt{FP}|$ for a lambda expression `lambda DS : FP := E`. This change makes the outermost lambda expression in the below example illegal:

```
lambda[4][3]:[i] := (lambda[3]:[j] := 0); // No longer legal
```

F Glossary

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

1. **approximate model**
   an approximation to a *model* indexed with a *meta step*, see Section 2.3.2
2. **ascendant scope**
   an enclosing scope, *i.e.* one higher up in the scope stack; see Section 2.4
3. **ASCII**
   American Standard Code for Information Interchange, a character encoding standard
4. **assignable / assignability**
   refers to the assignability relation between types, see Section 8; an expression $\texttt{E}_1$ is *assignable* to an expression $\texttt{E}_2$ *iff* the type of $\texttt{E}_1$ is assignable to the type of $\texttt{E}_2$
5. **combinational function**
   a mapping from values to values (usually a mapping from one or several values to a single value)
6. **combinational operator**
   see *combinational function*
7. **compatible / compatibility**
   (sometimes preceded by *type*) refers to the compatibility relation between types, see Section 8; an expression $\texttt{E}_1$ is *compatible* with an expression $\texttt{E}_2$ *iff* the type of $\texttt{E}_1$ is compatible with the type of $\texttt{E}_2$
8. **component**
   a component (type) of a composite type, a component of an expression of composite type, or a component of a composite value; similar terms used elsewhere would be *e.g.* struct field, tuple member or array element – all those entities are collectively called components in this document
9. **component projection**
   a function that extracts the value of a specific component of a composite value (the semantical equivalent to an accessor); for a given composite type `T`, the set of all component projections for values of type `T` is denoted $\mathcal{P}(\texttt{T})$, see Section 2.3.1
10. **composite**
    equivalent to *non-scalar*;
    (noun) an expression or value of composite type
    (adj.) of composite type
11. **composite type**
    equivalent to *non-scalar type*; any type defined in Section 8.2
12. **consequence**
    see Section 2.3.4

13. **constant**
    refers to any expression with a static flag of 2, which includes expressions defined using the nonterminal `<constant>` of Section 13

14. **declared type**
    see Section 11

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

16. **descendant scope**
    an enclosed scope, *i.e.* one further down in the scope stack; see Section 2.4

17. **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*

18. **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

19. **EBNF**
    Extended Backus-Naur Form, see also Section 2.5.1

20. **equivalent to**
    see Section 2.5.2

21. **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 explicitly grouped into respectively "`((a & b) & c)`" and "`(a + (b * c))`"

22. **extended values**
    for any given type `T`, the set of values $V(\texttt{T})$ it represents is extended to the set $V^*(\texttt{T})$ by allowing **nil** as a value in scalar leaves; see Section 2.3.1

23. **finite-dimensional**
    (adj.) a type is said to be finite-dimensional *iff* it is either scalar, or it is composite with a finite number of components, each of finite-dimensional type; an expression is finite-dimensional *iff* it is of finite-dimensional type.

24. **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

25. **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)

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

27. **function value**
    see *combinational function*

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

29. **hide / hiding**
    refers to variable hiding see Section 2.4

30. **HLL**
    High Level Language

31. **HLL function**
    an expression (of function type) modelled by a stream of combinational functions (function values) which, together with a function accessor, maps streams to streams by point-wise application of the combinational function in each time step on the values of the stream of the operand in the same time step; HLL functions are distinct from *temporal functions* since HLL functions, in each time step, only have access to the values in the current time step of its operands

32. **iff**
    if and only if

33. **implicit grouping**
    the process of mapping a text into its equivalent explicitly grouped text, governed by syntax, precedence, and associativity rules.

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

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

36. **information order**
    a partial order $\preceq$ on the set of all extended values: $v_1 \preceq v_2$ means that v_1 can be obtained from v_2 by replacing a subset of the scalar leaves in v_2 with **nil**-values (i.e., removing information)

37. **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)

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

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

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

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

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

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

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

45. **meta step**
    a natural number indicating a position in the sequence of approximate models used to define the semantics of HLL definitions, see Section 2.3.2

46. **model**
    see Section 2.3.2

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

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

49. **namespace**
    see Section 2.4; not to be confused with *user namespace*

50. **nil**
    see Section 2.2

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

52. **proposition**
    see Section 2.3.3

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

54. **reducible to**
see Section 2.5.2

55. **scalar**
(noun) an expression or value of scalar type
(adj.) of scalar type

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

57. **scope**
see Section 2.4

58. **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

59. **static flag**
see Section 2.3.5

60. **stream**
a stream of values of a certain type, see Section 2.1

61. **temporal operator**
a mapping from streams to streams where the resulting value in some time step depends on the value of other time steps; `X` and `PRE` are examples of temporal operators

62. **time step**
a natural number indicating a position in a stream, see Section 2.1

63. **type**
a property of expressions which represents a set of values, see Section 8; types are used to constrain the possible values an expression may take and restrict which operations can be performed on it.

64. **undefined variable**
see *free variable*

65. **underlying type**
the underlying type of a `<domain>` which is a `<type_domain> TD` is the type to which `TD` refers; the underlying type of a `<domain>` which is a `<range_domain>` is the type `int`; see Section 10.4

66. **unsized copy**
see Section 8.4.2

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

68. **value**
    a mathematical object that can be assigned a type. The set of values that can be assigned a type `T` is denoted $V(\texttt{T})$. There are two kinds of values: scalar and composite, which can be assigned scalar and composite types, respectively. Scalar values are atomic, such as the Boolean value “true”, the integer value “1234”, or the enum value “`blue`”. Composite values are either finite tuples of (scalar or composite) values or functions from one or more scalar values to (scalar or composite) values.

    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

69. **variable**
    a variable (a named expression; that can be referenced if it is *visible*)

70. **visible / visibility**
    refers to variable visibility, see Section 2.4

G Label Index

H Contact Us

If you have questions or comments concerning this document, or want to report an error or omission, you are welcome to contact HLL Forum at:

`support@hllforum.org`

For other inquiries please refer to the contact details on our Web page:

`https://www.hllforum.org/about-us/contact-us/`

We welcome your comments, so please do not hesitate to contact us.

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