High Level Language - Syntax and Semantics - Logical Foundation Document

Nicolas Breton

High Level Language pr4.0rc1

Syntax and Semantics
Logical Foundation Document

This is one of two documents that are published together.

- The “HLL Language Definition Document” describes the syntax and semantics of the formal modelling language HLL.
- The “HLL Logical Foundation Document” also describes these syntax and semantics, but in a form closer to that of the previously published version 2.7, allowing to clearly see how the new version of the language has evolved.

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

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

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

High Level Language pr4.0rc1

Syntax and Semantics
Logical Foundation Document

Contents

1 Preamble **5**
1.1 Purpose . . . 5
1.2 History . . . 5
1.3 Reference documents . . . 5
1.4 Terms and abbreviations . . . 6

2 Introduction **7**
2.1 How to read this document . . . 7
2.2 Requirement identification . . . 7
2.3 Overview . . . 7

3 Syntax **9**
3.1 Notation . . . 9
3.2 HLL syntax specification . . . 9
3.3 Comments . . . 13
3.4 Pragmas . . . 14
3.5 Operator precedence and associativity . . . 14
3.6 Identifiers . . . 14

4 Sections in HLL **15**

5 Namespaces and scoping rules **16**
5.1 HLL namespaces . . . 16
5.2 User namespaces . . . 18

6 HLL **types semantics** **20**
6.1 Basic types . . . 20
6.2 Enumerated types . . . 20
6.3 Sorts . . . 21
6.4 Tuples and structures . . . 21
6.5 Arrays . . . 21
6.6 Function types . . . 22
6.7 Named types . . . 22
6.8 Collection types . . . 22
6.9 HLL types . . . 22
6.10 Definitions on types . . . 23
6.11 Order on types . . . 27
6.11.1 Booleans . . . 27
6.11.2 Integers . . . 27
6.11.3 Enumerated types . . . 27
6.11.4 Sorts . . . 28
6.12 Order on composite domains . . . 28
6.12.1 Tuples . . . 28
6.12.2 Structures . . . 28
6.12.3 Arrays and Functions . . . 28
6.12.4 Nested composites . . . 29

7 Type checking rules **30**
7.1 Preliminary definitions . . . 30

7.2 Typing rules . . . 31
7.2.1 Typing expressions . . . 32
7.2.2 Typing definitions . . . 37
7.2.3 Typing declarations . . . 39
7.2.4 Typing types . . . 39
7.2.5 Typing type definitions . . . 40
7.2.6 Typing the entire model . . . 41

8 Additional static checks **43**
8.1 Partial stream definition . . . 43
8.2 Unicity of stream definitions . . . 43
8.3 Unfolding stream definitions . . . 43
8.4 Function Assignability . . . 44
8.5 Named type references and definitions . . . 44
8.6 Scoping rules (or namespaces) . . . 44
8.7 Well-definedness of constants . . . 44

9 Sorts: a hierarchy of enumerations **46**
9.1 Specifying a sort hierarchy . . . 46
9.2 Sorts and the switch-case expression . . . 47

10 Mapping semantics **49**
10.1 Arrays . . . 49
10.2 Function . . . 51
10.3 Making recursive definitions terminate . . . 52
10.4 Note about causality in the presence of mappings . . . 52

11 Defining composites using collections **53**
11.1 Tuples . . . 53
11.2 Structures . . . 53
11.3 Arrays . . . 53
11.4 Functions . . . 53
11.5 Example . . . 54

12 Local binders **55**
12.1 Quantifying over scalar types (built-in and user-defined) . . . 55
12.2 Quantifying over integer ranges . . . 56
12.3 Quantifying over enumerations . . . 56
12.4 Quantifying over mapping streams . . . 57
12.5 Arithmetic extensions . . . 57
12.6 Selection operator . . . 58
12.7 Summary of quantifier semantics . . . 59
12.8 Anonymous function and array definition (`lambda`) . . . 59
12.9 The *pigeon-hole* example . . . 61
12.10 The *sudoku* example . . . 61

13 Arithmetics in HLL **62**

14 Stream semantics **63**
14.1 Input streams . . . 63
14.2 Throw-away definition . . . 63

14.3 Combinatorial definition . . . 63
14.4 Memory definition . . . 64
14.5 Initial and next definitions . . . 64
14.6 Next definition only . . . 64
14.7 Next expression : `X(e)` . . . 64
14.8 Unit delay expression : `pre(e)` . . . 64
14.9 Definition of a data memory . . . 65
14.10 Array definitions . . . 65
14.11 Determinism, sanity, and nil values in HLL . . . 65

15 Causality **67**
15.1 Temporal dependencies between scalar streams . . . 67
15.2 Composite types, mappings and causality . . . 68

16 Predefined combinatorial operator semantics **69**
16.1 Logical operators . . . 69
16.2 Population count . . . 70
16.3 Polymorphic comparison operators `=`, `==`, `!=`, `<>` . . . 70
16.4 Shift operators `«`, `»` . . . 71
16.5 Arithmetic operators `+`, `-`, `*` and unary minus `-` . . . 71
16.6 Integer comparison operators `>`, `>=`, `<`, `<=` . . . 71
16.7 Maximum `$max` . . . 71
16.8 Minimum `$min` . . . 71
16.9 Absolute value `$abs` . . . 71
16.10 Euclidian division `/` . . . 72
16.11 Remainder `%` . . . 72
16.12 Floor division `/>` . . . 72
16.13 Ceiling division `/<` . . . 72
16.14 Bitwise logical operators: `$not`, `$and`, `$or`, `$xor` . . . 73
16.15 Power (`^`) . . . 73
16.16 Cast . . . 73
16.17 `bin2u` . . . 74
16.18 `bin2s` . . . 74
16.19 `u2bin` . . . 74
16.20 `s2bin` . . . 74
16.21 If-then-else . . . 75
16.22 Array projection . . . 75
16.23 Function application . . . 76
16.24 (... with ... := ...) . . . 76
16.25 Elementhood : `a:D` . . . 76

Appendix A: List of requirements **78**

Appendix B: List of reserved keywords **78**

1 PREAMBLE

1.1 Purpose

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

1.2 History

Index Date	Author	Chapters	Modifications
A 2021-09-10	N. Breton	ALL	Based on original LFD v2.7 by N. Breton and J.L. Colaço. Documents the new features of the HLL-forum version 4.0 language. – equality/disequality between finite domain functions of identical domains; – unified treatment of exceptional values (nil); – elementhood on scalar-types; – integer with empty ranges; – finite domain functions allowed in input/output sections; – empty types (instead of empty sort) forbidden in inputs and memories; – ordering of all types except sorts; – assignment of ordered domain functions with collections; – SELECT quantifier; – unfolding definitions (only combinatorial case); – quantification on any finite scalar type; – quantification on mapping streams; – combinatorial definition of variable with sized type.

1.3 Reference documents

None

1.4 Terms and abbreviations

EBNF	Extended Backus-Naur Form
HLL	High Level Language
LFD	Logical Foundations Document
MSB	Most Significant Bit of a binary word
LSB	Least Significant Bit of a binary word

2 INTRODUCTION

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

HLL allows to define streams (or sequences) of boolean or integer values in a declarative style; it offers a powerful mechanism to define arrays of streams. It aims at modelling sequential behaviours and expressing temporal properties on these behaviours.

2.1 How to read this document

The presentation of the HLL language proposed in this document is mainly dedicated to implementors, thus it targets more the absence of ambiguity than a pedagogical presentation.

For the end-user of HLL and particularly for the beginner, we propose to read the sections in the following order:

1. section 3
2. section 4
3. section 9
4. section 13
5. section 14
6. section 12
7. section 16
8. section 10
9. section 11
10. section 15

The rest of the document can be used as a reference manual only. Formalisation is here to reduce ambiguity for implementors and it is not needed for the end-user to invest a lot in a deep understanding to get a good representation of what HLL is.

2.2 Requirement identification

In this document, specific requirements are identified in order to provide a list of points that characterizes HLL. The identification is done with a tag on the form **HLL-xx** (where **xx** is an integer value) added in the right margin at the level of a section or subsection title. The requirement is defined by the whole content of the (sub)-section it is attached to.

The list of requirements present in the document is recalled in Appendix A.

2.3 Overview

- Section 3 gives the syntax of HLL, as an EBNF grammar.
- Section 4 presents the section oriented nature of HLL.
- Section 5 defines the namespaces and the scoping rules of the language.

[1] HLL stands for *High Level Language*

- Section 6 presents the types and type constructors available in HLL.
- Section 7 gives the formal rules defining the language's type system.
- Section 8 specifies additional (in the sense covered neither by the syntax nor the type system) semantic checks.
- Section 9 presents and discusses the semantics of sorts.
- Section 10 presents and discusses the semantics of array and function definitions.
- Section 11 details how collections can be used to define composite streams.
- Section 12 presents and discusses the different quantifiers.
- Section 13 defines the semantics of HLL arithmetics.
- Section 14 gives the semantics of the core stream language.
- Section 15 specifies the notion of causal HLL models.
- Section 16 defines all the combinatorial primitives offered by HLL.

3 SYNTAX

3.1 Notation

The syntax is given using the following subset of the EBNF notation:

- a non-terminal is written `<symbol>`;
- a symbol definition is introduced by `::=` with the defined symbol as the left-hand side;
- a terminal symbol is given by a string separated with quotes (`"terminal_string"`);
- the pipe, `|` represents the alternative;
- the square brackets are the optional items (`[<may-be-used>]`);
- the braces represent 0 or more times repetitions (`{<item>}`);
- the braces extended with `+` represent 1 or more times repetitions (`{<item>}+`).

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

3.2 HLL syntax specification

HLL-1

An HLL model is given as a text satisfying the following grammar:

```
<HLL>               ::= {<section>}

<section>           ::= <constants> ":"           {<constant> ";"}
                      | <types> ":"               {<type_def> ";"}
                      | <inputs> ":"              {<input> ";"}
                      | <declarations> ":"        {<declaration> ";"}
                      | <definitions> ":"         {<definition> ";"}
                      | <outputs> ":"             {<expr> ";"}
                      | <constraints> ":"         {<constraint> ";"}
                      | <proof> <obligations> ":" {<expr> ";"}
                      | <namespaces> ":"          {<id> "{" <HLL> "}"}

<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"

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

<type_def>          ::= <type> <name> {"," <name>}
                      | <enumerated> <id>
                      | "sort" [ <sort_contrib> "<" ] <id>

<name>              ::= <id> {<name_suffix>}

<name_suffix>       ::= "[" <expr_list> "]"
                      | "(" <type_list> ")"

<type>              ::= "bool"
                      | <integer>
                      | <tuple>
                      | <structure>
                      | <array>
                      | <path_id>
                      | <function>

<integer>           ::= "int"
                      | "int" <sign>
                      | "int" <range>

<sign>              ::= "signed" <id_or_int>
                      | "unsigned" <id_or_int>

<id_or_int>         ::= <id>
                      | <int_literal>

<range>             ::= "[" <expr> "," <expr> "]"

<enumerated>        ::= "enum" "{" <id_list> "}"

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

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

<sort_contrib>      ::= <path_id_list>
                      | "{" <id_list> "}"

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

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

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

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

<input>             ::= [<type>] <input_name> {"," <input_name>}

<input_name>        ::= <name>
                      | "I" "(" <name> ")"

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

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

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

<lhs>               ::= <id> {<formal_param>}

<unfold_lhs>        ::= <id_or_throw> { "," <id_or_throw> }

<id_or_throw>       ::= <id> | "_"

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

<rhs>               ::= <expr>
                      | <collection>

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

<expr>              ::= <closed_expr> { <accessor> }
                      | <expr> <binop> <expr>
                      | <expr> ":" <domain>
                      | <unop> <expr>
                      | "if"   <expr> "then" <expr>
                        {"elif" <expr> "then" <expr>}
                        "else" <expr>
                      | "lambda" {<name_suffix>}+ ":" {<formal_param>}+ ":=" <expr>

<closed_expr>       ::= <bool_literal>
                      | <int_literal>
                      | <path_id>
                      | "(" <expr> ")"
                      | "X" "(" <expr> ")"
                      | <fop> "(" <expr_list> ")"
                      | "cast" "<" <type> ">" "(" <expr> ")"
```

```
                    | "(" <expr> "with" {<accessor>}+ ":=" <rhs> ")"
                    | ("pre" | "PRE") ["<" <type> ">"] "(" <expr> ["," <expr>] ")"
                    | "(" <expr_list> {<case_item>}+ ")"
                    | <quantif_expr>

<quantif_expr>      ::= <quantifier> <quantif_var> {"," <quantif_var>}
                          ( "(" <expr> ")" | <quantif_expr> )
                    | "SELECT" <quantif_var> {"," <quantif_var>}
                        "(" <expr> ["," <expr>] ")"

<case_item>         ::= "|" <pattern_list> "=>" <expr>

<pattern>           ::= <expr>
                    | <path_id> ( <id> | "_" )
                    | "_"

<pattern_list>      ::= <pattern> { "," <pattern> }

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

<quantif_var>       ::= <id> ":" <domain>

<domain>            ::= <range>
                    | <path_id>
                    | "$items" "(" <path_id> ")"
                    | <integer>
                    | "bool"

<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"]+"

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

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

<dec_int_literal>    ::= regexp: [0-9][0-9_]*

<hex_int_literal>    ::= regexp: 0[Xx][0-9A-Fa-f][0-9A-Fa-f_]*

<bin_int_literal>    ::= regexp: 0[Bb][01][01_]*

<int_literal>        ::= <dec_int_literal>
                       | <hex_int_literal>
                       | <bin_int_literal>
```

3.3 Comments

HLL-2

An HLL text can contain comments in one of the following forms:

- lines containing a `"//"` (double slash) are ignored starting from the `"//"` sequence to the end of the line (including `"/*"` and `"*/"`);
- characters present between `"/*"` and `"*/"` are ignored (including `"//"`); comments of this kind can be nested.

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

Here are some examples that illustrate this specification:

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

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

3.4 Pragmas

HLL-3

All the characters after an `"@"` are interpreted as the text of a pragma until the end of the line.

Pragmas may be used by tools taking HLL as input language, the semantics of such pragmas is part of tool specifications.

3.5 Operator precedence and associativity

HLL-4

The relative priority is given in increasing order by the following table where all the operators of a given line share the same priority; the second column contains the associativity rule between these operators:

Operator	Associativity
`if . then . else .`, `lambda ...`	
`<->` `#!`	left
`->`	right
`#`	left
`&`	left
`>`, `>=`, `<`, `<=`, `=`, `==`, `!=`, `<>`, `:`	left
`<<`, `>>`	left
`+`, `-`	left
`*`, `/`, `/<`, `/>`, `%`	left
`^`	right
unary operators: `~`, `-`	

Remark: the associativity for the comparison operators (`<`, `>`, `<=`, `>=`) is given only to give an unambiguous mapping of an HLL text to a syntactic tree. In practice, any expression that involves this associativity will not type check because `a<b` is a boolean while comparisons apply on integers. All these verifications are specified by the type system.

3.6 Identifiers

HLL-5

The HLL syntax offers three syntactic forms for identifiers : *alpha-numeric*, *quoted* and *double quoted*. In the two last forms, the quotes are part of the identifier.

For instance, `A`, `'A'` and `"A"` represent three distinct identifiers that can be used to represent three different entities in the same namespace in a given model.

4 SECTIONS IN HLL

HLL-6

As described in the EBNF presented in Section 3, an HLL file is organised as a sequence of sections. These sections are of one of the following kinds:

- constant definitions (`constants`)
- type definitions (`types`)
- input declarations (`inputs`)
- stream declarations (`declarations`)
- stream definitions (`definitions`)
- output expressions (`outputs`)
- constraint expressions (`constraints`)
- proof obligations (`proof obligations`)
- namespace definitions (`namespaces`)

Each kind of section can appear several times in the file, for instance a model can contain two sections of type definitions; all the types defined by one of these two sections are visible at any point in the model. In the sections, the order of the items does not affect the meaning of the model. From a semantical point of view, declarations and definitions are treated as an unordered pool. In other words, the order present in the file is not relevant in the sequel of this document [2].

Constants defined in `constants` sections can only reference other constants; a constant cannot be defined with a stream even if it appears that this stream has a constant value. Constants are mainly used to parametrise a model with dimensions or boolean flags.

Inputs, declarations and type definitions contain type expressions that can need integer values for the array dimensions. These expressions must be built from constant and literal (in the sense defined by the syntactic entity `<int_literal>`) values only (no reference to a stream is allowed). This discipline is enforced by the type system described in Section 7.

[2] In the implementation of a tool based on HLL this order may be relevant to fulfill a functional requirement; in such a case the tool specification shall be explicit on this point.

5 NAMESPACES AND SCOPING RULES

This section defines the different namespaces and scopes that exist in HLL. This is an important notion that defines the way identifiers allow to bind a usage point in the model with a definition.

5.1 HLL namespaces

The HLL language has four namespaces :

1. one for *stream identifiers*, *enumeration values*, *sort values*, *iterator variables*, and *quantified variables*;
2. one for *type identifiers*;
3. one for *namespace identifiers*;
4. one for structure field labels.

The namespace for field labels is local to a structure type expression *i.e.* if a type `T` is a structure with a field named `m`, one can define anywhere else another structure type `U` with a field `m`.

The namespace for streams, iterator variables and quantifier variables offers nested scoping:

1. the top level one with all the stream and constant definitions;
2. the local one for the definition right-hand side;
3. and those introduced by the quantifiers and lambda expressions.

The scoping rule for the namespace of streams at the level of a definition is formalised by the two definitions below.

Definition 1 (local parameters variables)**.** *We define the function IV that computes the set of variables present in a left-hand side of a definition or in the formal parameters of a lambda. A* left-hand side *(lhs) is defined by:*

```
<lhs> ::= <id> <formal_params>
<formal_params>  ::= {("[" <id_list> "]") | ("(" <id_list> ")")}
```

Based on this syntactical form for lhs, the function IV is inductively defined by:

$$
\begin{array}{rcll}
IV(v) & = & \emptyset & \textit{where } v \textit{ is an identifier} \\
IV(v\ f) & = & IV(f) & \textit{where } v \textit{ is an identifier} \\
 & & & \textit{and } f \textit{ a list of formal parameters} \\
IV(f_1\ f_2) & = & IV(f_1) \cup IV(f_2) & \\
IV(\textit{lhs}[i_1, \ldots, i_k]) & = & IV(\textit{lhs}) \cup \{i_1, \ldots, i_k\} & \\
IV(\textit{lhs}(i_1, \ldots, i_k)) & = & IV(\textit{lhs}) \cup \{i_1, \ldots, i_k\} &
\end{array}
$$

For a definition, we call *free variables* the variables that appear in its *right-hand side* and are not bound. For instance, in the definition `a[i][j] := i - j * x`, `x` is a free variable while `i` and `j` are bound in the left-hand side of the definition.[3]

Definition 2 (Free variables)**.** *We define the function FV that computes the set of free variables, in the namespace of streams, present in an* expression*,* a type *or a* definition*. It is defined by mutual induction on streams, domains and types by:*

[3] Note that this notion of *free variable* is local to a definition and has nothing to do with the notion of model inputs.

On streams:

$$
\begin{array}{rcl}
FV(l) & = & \emptyset \textit{ where } l \textit{ is a literal} \\
FV(op(e_1,\ldots,e_n)) & = & \bigcup_{i\in[1..n]} FV(e_i) \\
 & & \textit{where } op \textit{ is any } n\textit{-ary operator } (n \geq 1) \\
 & & \textit{and } e_i \textit{ are expressions} \\
FV(\texttt{pre} < t > (e)) & = & FV(t) \cup FV(e) \\
FV(\texttt{cast} < t > (e)) & = & FV(t) \cup FV(e) \\
FV(f(e_1,\ldots,e_n)) & = & FV(f) \cup (\bigcup_{i\in[1..n]} FV(e_i)) \\
FV(a[e_1,\ldots,e_n]) & = & FV(a) \cup (\bigcup_{i\in[1..n]} FV(e_i)) \\
FV(e.m) & = & FV(e) \\
 & & \textit{where } e \textit{ is a stream expression} \\
 & & \textit{and } m \textit{ a structure label} \\
FV(v) & = & \{v\} \textit{ where } v \textit{ is a stream identifier} \\
FV(QTF\ i_1 : D_1,\ldots,i_n : D_n\ \ e) & = & (\bigcup_{k\in[1..n]} FV(D_k)) \cup (FV(e) \setminus \{i_1,\ldots,i_n\}) \\
 & & \textit{where } QTF \textit{ is an } \mathsf{HLL} \textit{ quantifier other than } \texttt{SELECT} \\
FV(\texttt{SELECT}\ i_1 : D_1,\ldots,i_n : D_n\ \ (e)) & = & (\bigcup_{k\in[1..n]} FV(D_k)) \cup (FV(e) \setminus \{i_1,\ldots,i_n\}) \\
FV(\texttt{SELECT}\ i_1 : D_1,\ldots,i_n : D_n\ \ (e,d)) & = & (\bigcup_{k\in[1..n]} FV(D_k)) \cup \\
 & & ((FV(e) \cup FV(d)) \setminus \{i_1,\ldots,i_n\}) \\
FV([e_1,\ldots,e_n]) & = & \bigcup_{i\in[1..n]} FV(e_i) \\
FV((t_1,\ldots,t_n)) & = & \bigcup_{i\in[1..n]} FV(t_i) \\
FV(\texttt{lambda}\ s_1 \ldots s_m : f_1 \ldots f_n := e) & = & (FV(e) \setminus \bigcup_{k\in[1..n]} \{f_k\}) \cup (\bigcup_{k\in[1..m]} FV(s_k)) \\
FV((e_1,\ldots,e_m | p_1 => ce_1 | \cdots | p_n => ce_n)) & = & (\bigcup_{i\in[1..n]} (FV(ce_k) \setminus V_{pat}(p_k))) \\
FV((e\ \texttt{with}\ a_1 \ldots a_n := e')) & = & FV(e) \cup FV(e') \cup (\bigcup_{i\in[1..n]} FV(a_i))
\end{array}
$$

On domains (note that a domain can also be a type):

$$FV(\texttt{int}[e_1,e_2]) \quad = \quad FV(e_1) \cup FV(e_2)$$

On types:

$$
\begin{array}{rcl}
FV(T) & = & \emptyset \textit{ where } T \textit{ is a type identifier} \\
FV([e_1,e_2]) & = & FV(e_1) \cup FV(e_2) \\
FV(\texttt{bool}) & = & \emptyset \\
FV(\texttt{int}) & = & \emptyset \\
FV(\texttt{int signed}\ n) & = & FV(n) \\
FV(\texttt{int unsigned}\ n) & = & FV(n) \\
FV(enum\ldots) & = & \emptyset \\
FV(sort\ldots) & = & \emptyset \\
FV(named\ldots) & = & \emptyset \\
FV(t\hat{}(e))) & = & FV(t) \cup FV(e) \\
FV(struct(l_0 : t_0,\ldots,l_n : t_n)) & = & \bigcup_{i\in[0..n]} FV(t_i) \\
FV(tuple(t_0,\ldots,t_n)) & = & \bigcup_{i\in[0..n]} FV(t_i) \\
FV(\{t_0,\ldots,t_n\}) & = & \bigcup_{i\in[0..n]} FV(t_i) \\
FV(t_1 \times \cdots \times t_n \to t) & = & FV(t) \cup (\bigcup_{i\in[1..n]} FV(t_i))
\end{array}
$$

And finally, on definitions:

$$
\begin{array}{rcl}
FV(\textit{lhs} := e) & = & FV(e) \setminus IV(lhs) \\
FV(v_1,\ldots,v_n := B(e_1,\ldots,e_n)) & = & \bigcup_{i\in[1..n]} FV(e_i)
\end{array}
$$

where $V_{pat}()$ is a function that takes a pattern and returns the set of variables it introduces:

$$\begin{array}{rcl} V_{pat}(v) & = & \emptyset \\ V_{pat}(S\ \_) & = & \emptyset \\ V_{pat}(S\ v) & = & \{v\} \\ V_{pat}(p_1, p_2) & = & V_{pat}(p_1) \cup V_{pat}(p_2) \end{array}$$

Based on these functions we can define the linking rule for a definition "$lhs := e$;" the free variables of this definition ($FV(lhs := e)$) are bound to the top level streams (inputs, outputs, local streams) while the other are bound locally in the iterator variables.

Another restriction for iterator variables: *for a given array definition, all the iterator variables must be different*. This point will be checked by the type checking rules. For instance `a[i,j][j]:=...` is incorrect since the iterator variable `j` appears twice on the left-hand side of the definition.

5.2 User namespaces

HLL-30

An HLL model can be organised as a hierarchy of named namespaces that allows to introduce new types or streams without any risk of name conflicts with another part of the model.

Such namespaces are introduced in a specific `namespaces` section; and may contain any kind of HLL sections including namespaces in this new scope (see syntax in section 3).

An identifier declared in a namespace can be referenced with the path to the namespace that declares it; this path can be either relative, *e.g.*:

```
localBox::drawer24::x,
```

or absolute, *e.g.*:

```
::topBox::drawer42::x.
```

The top level namespace is the one defined by the HLL file. In a given namespace, all the entities declared locally to this namespace must have different identifiers (except for the user namespaces that can be opened several times) and can hide any identifier introduced by an upper one.

A namespace can be defined in several parts, for example:

```
namespaces:
 a_namespace {
  ...
 }
 another_namespace {
  ...
 }

...

namespaces:
 a_namespace { // the namespace is re-opened
  ...
 }
```

Since HLL supports implicit declaration of scalar variables, the scoping rules for namespaces must consider this specificity. In the rules, we distinguish *simple identifier* and *path*, each involving different resolution mechanisms. These rules are:

1. in a given namespace, a simple identifier refers to the closest entity (stream or type), in the sense of the namespace hierarchy, declared or defined locally or in an upper level;
2. in a given namespace, a relative path identifier is first looked up locally by searching from the level it occurs in, if it is not found, the path is then looked up from the root of the model;
3. a local (to a namespace) definition of a type or contribution to a sort hides the definition of a type with the same identifier in any level above;
4. a local (to a namespace) declaration or definition of a stream, including the values of the locally defined enumerations or sorts, hides the declaration of a stream with the same identifier in any level above;
5. if a simple stream identifier is not declared, defined or used in any visible namespace but used in an expression, it is implicitly declared in the namespace where it is used, and visible in the namespaces below;
6. a declared stream can only be defined in the namespace that declares it, if it is not defined, it will be considered as an implicit input.

6 HLL TYPES SEMANTICS

This section defines the type language of HLL.

6.1 Basic types

HLL provides a boolean type `bool` defined by the set of values $\{\texttt{TRUE}, \texttt{FALSE}\}$. The values `true` and `True` (*resp.* `false` and `False`) are synonyms for `TRUE` (*resp.* `FALSE`). The set of boolean types contains a unique item: $\mathcal{T}_{\text{bool}} = \{\texttt{bool}\}$.

The type `int` contains all the positive and negative integer values ($\mathbb{Z}$); in practice, for a given model this set is bounded (see Section 13). In HLL the integer literals can be given either in base 10, such as `543`, in base 2, such as `0b1101`, or in base 16, such as `0x34AF5`. These literals are specified by the terminal grammar symbols `<dec_int_literal>`, `<bin_int_literal>`, and `<hex_int_literal>`. In these literals, underscores (_) are just used for readability and shall be discarded, `0x183_1AE5_23A5` is equivalent to `0x1831AE523A5`

An integer type in HLL may be constrained by a *range* or an *implementation*:

- *ranges* are specified by a pair of constant values $[a, b]$[4],
- *implementations* specify a size in bits and whether negative values are representable. In terms of the set of representable integers, we have:

$$\begin{aligned} \texttt{int signed } n &= \left(\left[-2^{n-1}, 2^{n-1}-1\right]\right) \text{ with } n > 0 \\ \texttt{int unsigned } n &= \left([0, 2^n - 1]\right) \text{ with } n \geq 0 \end{aligned}$$

The set $\mathcal{T}_{\text{i}}$ of the integer types is defined by:

$$\mathcal{T}_{\text{i}} = \{int\} \cup \bigcup_{a \leq b} \{int([a,b])\} \cup \bigcup_{n>0} \{int(\texttt{signed } n)\} \cup \bigcup_{n \geq 0} \{int(\texttt{unsigned } n)\}$$

In the sequel, we will denote by `int` an unconstrained integer type, $\texttt{int}(R)$ an integer type constrained by a range and $\texttt{int}(I)$ an integer constrained by an implementation.

Note: Implementation and range information is used to bound the arithmetics of the model. At the type checking level, the information on ranges is mostly irrelevant; two integer types are equivalent regardless of the specified range or representation (see Definitions 9, 13, and 14).

6.2 Enumerated types

The enumerated types are defined by:

$$\mathcal{T}_{\text{enum}} = \{enum(T; l_1, \ldots, l_n) \mid n > 0 \wedge (\forall i \in [1..n],\ l_i \in \mathcal{L}) \wedge (\forall i, j \in [1..n],\ i \neq j \ \Rightarrow\ l_i \neq l_j)\}$$

where $\mathcal{L}$ is the set of possible labels for enumerated values and T is the name of the enumerated type. Note that each label shall only be used in the definition of one enumerated type or sort.

[4] if $a > b$ the type is said to be empty (see Definition 8).

6.3 Sorts

A sort type represents a set of values as an enumerated type does. The difference is in the way the list of values is built; in the case of sorts this list is specified by giving both the values it introduces and the set of subsorts. For a given sort the corresponding set of values contains the values introduced for this sort and all the values of the subsorts. We first introduce $\mathcal{T}^0_{\text{sort}}$ that represents the set of sort types as they appear in the source:

$$\mathcal{T}^0_{\text{sort}} = \left\{ sort(S; L; Sub) \,|\, L \subseteq \mathcal{L}_S \wedge \ (\forall S' \in Sub,\ sort(S'; \ldots) \in \mathcal{T}^0_{\text{sort}}) \right\}$$

where $\mathcal{L}_S$ is the set of possible labels for sort values. Intuitively, $sort(S; L; Sub)$ means sort S contains all the labels in L and all the labels of the sorts present in Sub; it gathers all the contributions to S that are present in the considered HLL model. This definition is well founded because cyclic type definitions are forbidden (see restriction in 8.5). Note that each label shall only be used in the definition of one sort or enumerated type.

Let $\sqsubseteq^s$ be the smallest partial order on $\mathcal{T}^0_{\text{sort}}$ such that:

$$\forall S' \in Sub,\ sort(S'; L'; Sub') \sqsubseteq^s sort(S; L; Sub)$$

Let $(\mathcal{T}_{\text{sort}}, \sqcup^s, \sqsubseteq^s)$ be the smallest *upper semilattice* containing $\mathcal{T}^0_{\text{sort}}$. The following properties hold:

$$\mathcal{T}_{\text{sort}} = \mathcal{T}^0_{\text{sort}} \cup \{ \tau_1 \sqcup^s \tau_2 \,|\, \tau_1 \in \mathcal{T}_{\text{sort}} \ \wedge \ \tau_2 \in \mathcal{T}_{\text{sort}} \}$$

$$\forall \sigma, \sigma' \in \mathcal{T}_{\text{sort}}\ \sigma \sqsubseteq^s \sigma' \ \Leftrightarrow \ \sigma' = \sigma \sqcup^s \sigma'$$

6.4 Tuples and structures

The tuple types are defined by:

$$\mathcal{T}_{\text{tuple}} = \{ tuple(\tau_0, \ldots, \tau_n) \,|\, n \geq 0 \wedge \forall i \in [0..n],\ \ \tau_i \in \mathcal{T}_{\text{e}} \}$$

Where $\mathcal{T}_{\text{e}}$ is the set of all HLL types, as defined in 6.9.

The structure types are defined by:

$$\mathcal{T}_{\text{struct}} = \left\{ struct(l_0 : \tau_0, \ldots, l_n : \tau_n) \,|\, n \geq 0 \ \begin{array}{l} \wedge (\forall i \in [0..n],\ l_i \in \mathcal{L} \wedge \tau_i \in \mathcal{T}_{\text{e}}) \\ \wedge (\forall i, j \in [0..n], i \neq j \ \Rightarrow \ l_i \neq l_j) \end{array} \right\}$$

where $\mathcal{L}$ is the set of possible field names and $\mathcal{T}_{\text{e}}$ is the set of all HLL types, as defined in 6.9.

Note about tuples and structures: Tuples and structures are very similar (though incompatible); they only differ in the way one accesses the fields. In the case of a tuple, this access is positional (starting at index 0) and in the case of a structure, the access is through the name of the field. In both cases, the order of the fields matters.

6.5 Arrays

Arrays are mappings that associate a stream to each tuple of integer values in the definition domain. The array types are defined by:

$$\mathcal{T}_{\text{array}} = \{ \tau\hat{}(d_1, \ldots, d_n) \,|\, \tau \in \mathcal{T}_{\text{e}} \wedge n > 0 \wedge \forall i \in [1..n]\ \ d_i \geq 0 \}$$

Where $\mathcal{T}_{\text{e}}$ is the set of all HLL types, as defined in 6.9.

In the array type $\tau\hat{}(d_1, \ldots, d_n)$, τ is the type of the array elements, and the array has n dimensions such that dimension i has size d_i in the sense that the possible indices at that dimension is restricted by d_i.

The array indices start at 0. Thus if a is an array of type $\tau\hat{}(d_1, \ldots, d_n)$, it can be accessed by a tuple of indices $[i_1, \ldots, i_n]$ such that $\forall k \in [1..n],\ i_k \in [0..d_k - 1]$.

6.6 Function types

Functions are mappings that associate a stream to each tuple of scalar values in the definition domains.

The function types are defined by:

$$\mathcal{T}_{\text{fun}} = \{\tau_1 \times \cdots \times \tau_n \to \tau \,|\, n > 0 \wedge \tau \in \mathcal{T}_{\text{e}} \wedge \forall i \in [1..n]\, \tau_i \in \mathcal{T}_{\text{s}}\}$$

Where $\tau_1 \times \cdots \times \tau_n \to \tau$ is the type of a function that takes n arguments of type $\tau_1 \ldots \tau_n$ and produces a value of type τ. $\mathcal{T}_{\text{e}}$ is the set of all HLL types, as defined in 6.9.

6.7 Named types

A type can be named by associating a type name with a type expression in the `types` section.

The named types are defined by:

$$\mathcal{T}_{\text{named}} = \{named(l, \tau) \,|\, \tau \in \mathcal{T}_{\text{e}} \wedge l \in \mathcal{L}\}$$

where $\mathcal{L}$ is the set of all possible names and $\mathcal{T}_{\text{e}}$ is the set of all HLL types, as defined in 6.9.

A name acts as an alias except when it designates a sort; in that case, the name introduces a new type. This new type cannot be used as a sort type to receive new contributions.

6.8 Collection types

Collection types are used to represent the type information of a definition's right-hand side. A special type and a special *assignability* relation must be defined since the same right-hand side can be used to define either a *tuple*, a *structure* or an *array* depending on the declared type of the left-hand side.

To define this overloading, we introduce a special collection type:

$$\mathcal{T}_{\text{collection}} = \{collection(\tau_0, \ldots, \tau_n) \,|\, n \geq 0 \wedge \forall i \in [0, n]\ \ \tau_i \in \mathcal{T}_{\text{collection}} \cup \mathcal{T}_{\text{e}}\}$$

6.9 HLL types

The set of all HLL types is defined by:

$$\mathcal{T}_{\text{e}} = \mathcal{T}_{\text{s}} \cup \mathcal{T}_{\text{tuple}} \cup \mathcal{T}_{\text{struct}} \cup \mathcal{T}_{\text{array}} \cup \mathcal{T}_{\text{named}} \cup \mathcal{T}_{\text{fun}}$$

Where $\mathcal{T}_s$ is the set of scalar types defined by:

$$\mathcal{T}_s = \mathcal{T}_{bool} \cup \mathcal{T}_i \cup \mathcal{T}_{enum} \cup \mathcal{T}_{sort}$$

6.10 Definitions on types

We introduce here some definitions on types that we need to specify the type checking rules.

Definition 3 (Sized types)**.** *A type is* sized *if all the integer components appearing in its final base type (see Definition 10) are specified with a range or size. The predicate* $Sized$ *defined below formalises this definition:*

$$\begin{aligned}
Sized(\texttt{bool}) &= \top \\
Sized(\texttt{int}) &= \bot \textit{ see note below} \\
Sized(\texttt{int}(R)) &= \top \\
Sized(\texttt{int}(I)) &= \top \\
Sized(enum\ _) &= \top \\
Sized(sort\ _) &= \top \\
Sized(named(_, \tau)) &= Sized(\tau) \\
Sized(\tau\hat{}(_)) &= Sized(\tau) \\
Sized(struct(l_0 : \tau_0, \ldots, l_n : \tau_n)) &= \forall i \in [0..n], Sized(\tau_i) \\
Sized(tuple(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], Sized(\tau_i) \\
Sized(collection(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], Sized(\tau_i) \\
Sized(\tau_1 \times \cdots \times \tau_n \to \tau) &= Sized(\tau)
\end{aligned}$$

where $\top$ *(resp.* $\bot$*) represents the logical value* true *(resp.* false*) that the predicate takes.*

Note about tool behaviour regarding sized types: The previous definition basically says that this predicate is *true* when applied to a type that has a specified size for all the integers it contains (given as a bit representation or as a range of values). It is possible for a tool that takes HLL as an input language to provide a way (option or pragma) to specify a *default integer size* to be used each time such a size is needed and not present in the model. When such a feature is used the predicate $Sized$ is true for every type. However, the setting of this default integer size shall **not** apply to plain integer function domains which shall remain unsized.

Definition 4 (Finite domain types)**.** *A type is of finite domain if all the integer components appearing in its domains are specified with a range or size. The predicate* $FiniteDom$ *defined below formalises this definition:*

$$\begin{aligned}
FiniteDom(\texttt{bool}) &= \top \\
FiniteDom(\texttt{int}) &= \top \\
FiniteDom(\texttt{int}(R)) &= \top \\
FiniteDom(\texttt{int}(I)) &= \top \\
FiniteDom(enum\ _) &= \top \\
FiniteDom(sort\ _) &= \top \\
FiniteDom(named(_, \tau)) &= FiniteDom(\tau) \\
FiniteDom(\tau\hat{}(_)) &= FiniteDom(\tau) \\
FiniteDom(struct(l_0 : \tau_0, \ldots, l_n : \tau_n)) &= \top \\
FiniteDom(tuple(\tau_0, \ldots, \tau_n)) &= \top \\
FiniteDom(collection(\tau_0, \ldots, \tau_n)) &= \top \\
FiniteDom(\tau_1 \times \cdots \times \tau_n \to \tau) &= FiniteDom(\tau) \wedge \forall i \in [0..n], Sized(\tau_i)
\end{aligned}$$

where $\top$ *(resp.* $\bot$*) represents the logical value* true *(resp.* false*) that the predicate takes.*

In the last line, the definition for functions, the parameters of the function is asked to be $Sized$. As previously noted above the definition, a tool shall **not** attempt to constrain the size of these parameters using some kind of default integer size.

Definition 5 (Types with an ordered domain)**.** *We define a predicate over types that is true if an order can be defined on the domain of these types.*

It is defined recursively by:

$$\begin{aligned}
HasOrderedDomain(\texttt{bool}) &= \top \\
HasOrderedDomain(\texttt{int}) &= \top \\
HasOrderedDomain(\texttt{int}(R)) &= \top \\
HasOrderedDomain(\texttt{int}(I)) &= \top \\
HasOrderedDomain(enum\ _) &= \top \\
HasOrderedDomain(sort\ _) &= \bot \\
HasOrderedDomain(named(_, \tau)) &= HasOrderedDomain(\tau) \\
HasOrderedDomain(\tau\hat{}(_)) &= \top \\
HasOrderedDomain(struct(l_0 : \tau_0, \ldots, l_n : \tau_n)) &= \forall i \in [0..n], HasOrderedDomain(\tau_i) \\
HasOrderedDomain(tuple(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], HasOrderedDomain(\tau_i) \\
HasOrderedDomain(collection(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], HasOrderedDomain(\tau_i) \\
HasOrderedDomain(\tau_0, \ldots, \tau_n \to \tau) &= \forall i \in [0, n], HasOrderedDomain(\tau_i)
\end{aligned}$$

where $\top$ (resp. $\bot$) represents the logical value true *(resp.* false*) that the predicate takes.*

In the current version of HLL, all types satisfy the $HasOrderedDomain$ predicate except sorts. The actual order for each type is described in section 6.11.

Definition 6 (Scalar types)**.** *We define a predicate over types that is true ($\top$) when the type is scalar. It is defined recursively by:*

$$\begin{aligned}
Scalar(\texttt{bool}) &= \top \\
Scalar(\texttt{int}) &= \top \\
Scalar(enum\ _) &= \top \\
Scalar(sort\ _) &= \top \\
Scalar(named(_, \tau)) &= Scalar(\tau) \\
Scalar(_\hat{}(_)) &= \bot \\
Scalar(struct(l_0 : \tau_0, \ldots, l_n : \tau_n)) &= \bot \\
Scalar(tuple(\tau_0, \ldots, \tau_n)) &= \bot \\
Scalar(collection(\tau_0, \ldots, \tau_n)) &= \bot \\
Scalar(\tau_1 \times \cdots \times \tau_n \to \tau) &= \bot
\end{aligned}$$

Definition 7 (Type cardinal)**.** *For a sized scalar type τ, we define the cardinal of the type, that is, the number of possible values of that type. The predicate $Card$ defined below formalises this definition:*

$$\begin{aligned}
Card(\texttt{bool}) &= 2 \\
Card(int([a, b])) &= (1 + b - a) \\
Card(int(\texttt{signed}\ n)) &= 2^n \\
Card(int(\texttt{unsigned}\ n)) &= 2^n \\
Card(enum(_; l_1, \ldots, l_n)) &= n \\
Card(sort(_; l_1, \ldots, l_n; Sub_1, \ldots, Sub_m)) &= n + \textstyle\sum_{i=1}^{m} Card(Sub_i) \\
Card(named(_, \tau)) &= Card(\tau)
\end{aligned}$$

Definition 8 (Non empty type)**.** *We define a predicate over types that is true ($\top$) when the type does not contain an empty type. It is defined recursively by:*

$$
\begin{aligned}
noEmptyType(\texttt{bool}) &= \top \\
noEmptyType(\texttt{int}(I)) &= \top \\
noEmptyType(\texttt{int}(R)) &= R \neq \emptyset \text{ (which happens in } [a,b] \text{ when } a > b) \\
noEmptyType(\texttt{int}) &= \top \\
noEmptyType(enum\ \_) &= \top \\
noEmptyType(sort(S; L; Sub)) &= L \neq \emptyset \vee \exists S' \in Sub, noEmptyType(S') \\
noEmptyType(named(\_, \tau)) &= noEmptyType(\tau) \\
noEmptyType(\tau\hat{}(d_1, \ldots, d_n)) &= noEmptyType(\tau) \vee \exists i \in [1..n], d_i = 0 \\
noEmptyType(\tau_1 \times \cdots \times \tau_n \to \tau) &= noEmptyType(\tau) \vee \exists i \in [1..n], \neg noEmptyType(\tau_i) \\
noEmptyType(struct(l_0 : \tau_0, \ldots, l_n : \tau_n)) &= \forall i \in [0..n], noEmptyType(\tau_i) \\
noEmptyType(tuple(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], noEmptyType(\tau_i) \\
noEmptyType(collection(\tau_0, \ldots, \tau_n)) &= \forall i \in [0..n], noEmptyType(\tau_i)
\end{aligned}
$$

This predicate characterises the presence of an empty type in a given HLL type, it is used to restrict the inputs/memories of a model that must all be of an inhabited type.

In the case of functions and arrays, this predicate characterizes the fact that the return value is non empty, since calling a function or dereferencing an array returning a value of an empty type would have no meaning. However if one of the dimension of the array is 0, or if one of the parameter of the function is an empty type (*i.e.* the domain is empty), it is then only impossible to use the function or array. When both the domain and co-domain are empty, we do not care about the emptyness of the co-domain because the function or array cannot be used. In this case we rather artificially consider the type to be non empty.

Definition 9 (Compatibility)**.** *The compatibility between two types τ and τ', denoted $\tau \equiv \tau'$, is the equivalence relation* (reflexive, transitive and symmetric) *inductively defined by:*

$$
\begin{aligned}
\texttt{int} &\equiv \texttt{int}(R) && \forall R \\
\texttt{int} &\equiv \texttt{int}(I) && \forall I \\
tuple(\tau_0, \ldots, \tau_n) &\equiv tuple(\tau'_0, \ldots, \tau'_n) && \textit{iff } \forall i \in [0..n]\ \ \tau_i \equiv \tau'_i \\
struct(l_0 : \tau_0, \ldots, l_n : \tau_n) &\equiv struct(l_0 : \tau'_0, \ldots, l_n : \tau'_n) && \textit{iff } \forall i \in [0..n]\ \ \tau_i \equiv \tau'_i \\
\tau\hat{}(d_1, \ldots, d_n) &\equiv \tau'\hat{}(d_1, \ldots, d_n) && \textit{iff } \tau \equiv \tau' \\
\tau_1 \times \cdots \times \tau_n \to \tau &\equiv \tau'_1 \times \cdots \times \tau'_n \to \tau' && \textit{iff } \forall i \in [1..n]\ \ \tau_i \equiv \tau'_i \wedge \tau \equiv \tau' \\
collection(\tau_0, \ldots, \tau_n) &\equiv collection(\tau'_0, \ldots, \tau'_n) && \textit{iff } \forall i \in [0..n]\ \ \tau_i \equiv \tau'_i \\
enum(T; \_) &\equiv enum(T'; \_) && \textit{iff } \ T = T' \\
sort(S; \_; \_) &\equiv sort(S'; \_; \_) \\
named(l, \tau) &\equiv \tau
\end{aligned}
$$

Note that the last case distinguishes enumerations and sorts from other named types; two enumerated type expressions are compatible if they refer to the same enumeration name and two sorts are always compatible (the type compatibility relation does need to look inside enumeration or sort definitions).

Definition 10 (Final base type of a type)**.** *The* base type *of a type, in its usual mathematical definition, is the type itself, or the return type of a mapping type. The* final base type *is the result of applying the* base type *definition until reaching a fixed point.*

$$
\begin{aligned}
FinalBaseType(\texttt{bool}) &= \texttt{bool}\\
FinalBaseType(\texttt{int}) &= \texttt{int}\\
FinalBaseType(\texttt{int}(R)) &= \texttt{int}(R)\\
FinalBaseType(\texttt{int}(I)) &= \texttt{int}(I)\\
FinalBaseType(enum(E)) &= enum(E)\\
FinalBaseType(sort(S)) &= sort(S)\\
FinalBaseType(named(\_,\tau)) &= FinalBaseType(\tau)\\
FinalBaseType(\tau\hat{}(\_)) &= FinalBaseType(\tau)\\
FinalBaseType(struct(l_0:\tau_0,\ldots,l_n:\tau_n)) &= struct(l_0:\tau_0,\ldots,l_n:\tau_n)\\
FinalBaseType(tuple(\tau_0,\ldots,\tau_n)) &= tuple(\tau_0,\ldots,\tau_n)\\
FinalBaseType(collection(\tau_0,\ldots,\tau_n)) &= collection(\tau_0,\ldots,\tau_n)\\
FinalBaseType(\tau_1\times\cdots\times\tau_n\to\tau) &= FinalBaseType(\tau)
\end{aligned}
$$

Definition 11 (Union)**.** *The union on sorts extends to types as specified by the following definition:*

$$
\begin{aligned}
\tau \sqcup^s \tau &= \tau\\
\sigma \sqcup^s \sigma' &= \textit{defined by definition of } \mathcal{T}_{sort}\\
tuple(\tau_0,\ldots,\tau_n) \sqcup^s tuple(\tau'_0,\ldots,\tau'_n) &= tuple(\tau_0\sqcup^s\tau'_0,\ldots,\tau_n\sqcup^s\tau'_n)\\
struct(l_0:\tau_0,\ldots,l_n:\tau_n) \sqcup^s struct(l_0:\tau'_0,\ldots,l_n:\tau'_n) &= struct(l_0:\tau_0\sqcup^s\tau'_0,\ldots,l_n:\tau\sqcup^s\tau'_n)\\
collection(\tau_0,\ldots,\tau_n) \sqcup^s collection(\tau'_0,\ldots,\tau'_n) &= collection(\tau_0\sqcup^s\tau'_0,\ldots,\tau_n\sqcup^s\tau'_n)\\
\tau\hat{}(d_1,\ldots,d_n) \sqcup^s \tau'\hat{}(d_1,\ldots,d_n) &= (\tau\sqcup^s\tau')\hat{}(d_1,\ldots,d_n)\\
(\tau_1\times\cdots\times\tau_n\to\tau) \sqcup^s (\tau_1\times\cdots\times\tau_n\to\tau') &= \tau_1\times\cdots\times\tau_n\to(\tau\sqcup^s\tau')
\end{aligned}
$$

Where σ and σ' are sorts.

All the cases that do not match one of the cases given in this list are undefined.

Definition 12 (Subsorting)**.** *The subsorting relation $\sqsubseteq^s$ is the partial order (*reflexive, transitive, anti-symmetric*) of the upper semilattice $(\mathcal{T}_{sort}, \sqcup^s, \sqsubseteq^s)$ defined in 6.3.*

Definition 13 (Subtyping)**.** *The subtyping relation between two types τ and τ', denoted $\tau \preceq \tau'$, is the partial order* (reflexive, transitive and anti-symmetric) *inductively defined by:*

$$
\begin{aligned}
\sigma &\preceq \sigma' && \textit{iff } \sigma\sqsubseteq^s\sigma'\\
named(l,\tau) &\preceq \tau' && \textit{iff } \tau\preceq\tau'\\
\tau &\preceq named(l,\tau') && \textit{iff } \tau\preceq\tau'\\
\texttt{int}(R) &\preceq \texttt{int}\\
\texttt{int}(I) &\preceq \texttt{int}\\
\texttt{int}([a,b]) &\preceq \texttt{int}([a',b']) && \textit{iff } [a,b]\subseteq[a',b']\\
\texttt{int}([a,b]) &\preceq \texttt{int unsigned n} && \textit{iff } [a,b]\subseteq[0,2^n-1]\\
\texttt{int}([a,b]) &\preceq \texttt{int signed n} && \textit{iff } [a,b]\subseteq[-2^{n-1},2^{n-1}-1]\\
\texttt{int unsigned n} &\preceq \texttt{int}([a,b]) && \textit{iff } [0,2^n-1]\subseteq[a,b]\\
\texttt{int signed n} &\preceq \texttt{int}([a,b]) && \textit{iff } [-2^{n-1},2^{n-1}-1]\subseteq[a,b]\\
\texttt{int signed n} &\preceq \texttt{int signed n}' && \textit{iff } n\le n'\\
\texttt{int unsigned n} &\preceq \texttt{int unsigned n}' && \textit{iff } n\le n'\\
tuple(\tau_0,\ldots,\tau_n) &\preceq tuple(\tau'_0,\ldots,\tau'_n) && \textit{iff } \forall i\in[0..n]\ \ \tau_i\preceq\tau'_i\\
collection(\tau_0,\ldots,\tau_n) &\preceq collection(\tau'_0,\ldots,\tau'_n) && \textit{iff } \forall i\in[0..n]\ \ \tau_i\preceq\tau'_i\\
struct(l_0:\tau_0,\ldots,l_n:\tau_n) &\preceq struct(l_0:\tau'_0,\ldots,l_n:\tau'_n) && \textit{iff } \forall i\in[0..n]\ \ \tau_i\preceq\tau'_i\\
\tau\hat{}(d_1,\ldots,d_n) &\preceq \tau'\hat{}(d_1,\ldots,d_n) && \textit{iff } \tau\preceq\tau'\\
\tau_1\times\cdots\times\tau_n\to\tau &\preceq \tau'_1\times\cdots\times\tau'_n\to\tau' && \textit{iff } \begin{cases}\forall i\in[1..n]\,\tau_i\preceq\tau'_i\ \wedge\ \tau'_i\preceq\tau_i\\ \wedge\,\tau\preceq\tau'\end{cases}\\
\tau_1\sqcup^s\tau_2 &\preceq \tau && \textit{iff } \tau_1\preceq\tau\ \wedge\ \tau_2\preceq\tau
\end{aligned}
$$

where σ and σ' represent sorts.

*The informal understanding of this relation is that, if $\tau \preceq \tau'$, then any value of type τ can be used where a value of type τ' is required (*substitutability*).*

The following *assignability* relation defines the type correctness for an HLL definition where the first type represents the declared left-hand side type and the second the right-hand side one.

Definition 14 (Assignability)**.** *The assignability relation between two types τ and τ', denoted $\tau \lhd \tau'$ (pronounce τ'* is assignable to *τ) is the pre-order* (reflexive and transitive) *inductively defined by:*

$$
\begin{array}{rl}
\tau \lhd \tau' & \textit{if } \tau' \preceq \tau \vee (\tau \in \mathcal{T}_i \wedge \tau' \in \mathcal{T}_i) \\
\tau_1 \times \cdots \times \tau_n \to \tau \lhd \tau'_1 \times \cdots \times \tau'_n \to \tau' & \textit{if } \left\{ \begin{array}{l} \tau \lhd \tau' \wedge \\ \forall i \in [1,n] \quad (\tau_i \preceq \tau'_i \wedge \tau'_i \preceq \tau_i) \end{array} \right. \\
\tau\hat{}(d) \lhd \tau'\hat{}(d) & \textit{iff } \tau \lhd \tau' \\
tuple(\tau_0, \ldots, \tau_n) \lhd collection(\tau'_0, \ldots, \tau'_n) & \textit{iff } \forall i \;\; \tau_i \lhd \tau'_i \\
struct(l_0 : \tau_0, \ldots, l_n : \tau_n) \lhd collection(\tau'_0, \ldots, \tau'_n) & \textit{iff } \forall i \;\; \tau_i \lhd \tau'_i \\
\tau \to \tau' \lhd collection(\tau_0, \ldots, \tau_{Card(\tau)-1}) & \textit{iff } \left\{ \begin{array}{l} Sized(\tau) \wedge \\ \forall i \in [0..Card(\tau)-1] \\ \quad \tau' \lhd \tau_i \end{array} \right. \\
(\tau_1 \times \cdots \times \tau_n \to \tau)_{(n>1)} \lhd collection(\tau'_0, \ldots, \tau'_{Card(\tau_1)-1}) & \textit{iff } \left\{ \begin{array}{l} Sized(\tau_1) \wedge \\ \forall i \in [0..Card(\tau_1)-1] \\ \quad \tau_2 \times \cdots \times \tau_n \to \tau \lhd \tau'_i \end{array} \right. \\
\tau\hat{}(d) \lhd collection(\tau_0, \ldots, \tau_{d-1}) & \textit{iff } \forall i \in [0..d-1] \;\; \tau \lhd \tau_i \\
\tau\hat{}(d_1, \ldots, d_n)_{(n>1)} \lhd collection(\tau_0, \ldots, \tau_{d_1-1}) & \textit{iff } \forall i \in [0..d_1-1] \;\; \tau\hat{}(d_2, \ldots, d_n) \lhd \tau_i
\end{array}
$$

There is in fact one rule in the typechecking in which integer domain (implementation or range) is relevant, it is in the assignability of functions containing integer domains, but this is treated as a specific additional check in section 8.4.

6.11 Order on types

All scalar types in HLL (as defined by the $Scalar()$ predicate) are ordered except sorts.

6.11.1 Booleans

The order on booleans is defined by the relation `FALSE` $<$ `TRUE`.

6.11.2 Integers

The order on all integer types, constrained or not, is the natural order on relative numbers.

6.11.3 Enumerated types

The order on enumerated types is defined by the order of their declaration. For instance, considering the following model:

```
Types:
  enum { FIRST, SECOND, THIRD } TEnum;
```

the order on the enumerated type `TEnum` is defined by `FIRST` $<$ `SECOND` $<$ `THIRD`. There is no order defined between different enumerated types.

6.11.4 Sorts

Sorts are not ordered.

6.12 Order on composite domains

Composite domains are also ordered in HLL, as specified in definition 5.

6.12.1 Tuples

The domain of a tuple type $tuple(\tau_0, \ldots, \tau_n)$ is the set of natural numbers in the range $[0, n]$. The order over that domain is the natural order over integers.

6.12.2 Structures

The domain of a structure type $struct(l_0 : \tau_0, \ldots, l_n : \tau_n)$ is the list of its fields $l_0, \ldots, l_n$. The order over that domain is defined by the relation $\forall (i, j) \in [0, n], l_i < l_j \Leftrightarrow i < j$.

6.12.3 Arrays and Functions

We define for the scope of this paragraph a lexicographical order relation $<_{lex}$ on tuples of scalar types inductively by

$$(i_1, \ldots, i_n) <_{lex} (j_1, \ldots, j_n) \Leftrightarrow \begin{cases} i_1 <_1 j_1 & \text{if } n = 1 \\ (i_1 <_1 j_1) \vee \\ ((i_1 = j_1) \wedge ((i_2, \ldots, i_n) <_{lex} (j_2, \ldots, j_n))) & \text{if } n > 1 \end{cases}$$

where $<_1, \ldots, <_n$ are the orders over the types of the members of the tuple as defined in section 6.11.

The domain of an array type $\tau\hat{}(d_1, \ldots, d_n)$ is a tuple of types $(\texttt{int}([0, d_1 - 1]), \ldots, \texttt{int}([0, d_n - 1]))$, and the domain of a function type $\tau_1 \times \cdots \times \tau_n \to \tau$ is a tuple of scalar types $(\tau_1, \ldots, \tau_n)$. Both are ordered by the relation $<_{lex}$ defined above.

6.12.4 Nested composites

HLL allows nesting composites, it is for example possible to declare a type that is a tuple of structures. The lexicographical ordering on nested domains is defined by iterating on each domain in a depth first manner.

For instance, the two HLL models below are semantically equivalent:

```
Constants:
  bool T := true; bool F := false;

Types:
  tuple {bool, bool} TupleType;
  struct {
    x: int,
    y: TupleType
  } StructType;
  StructType^(2, 2) ArrayType;
  (bool*bool->TupleType) FuncType1;
  (bool->(bool->TupleType)) FuncType2;

Inputs:
  ArrayType a;
  FuncType1 f1;
  FuncType2 f2;

Outputs:
  a;




  f1;




  f2;
```

```
Constants:
  bool T := true; bool F := false;

Types:
  tuple {bool, bool} TupleType;
  struct {
    x: int,
    y: TupleType
  } StructType;
  StructType^(2, 2) ArrayType;
  (bool*bool->TupleType) FuncType1;
  (bool->(bool->TupleType)) FuncType2;

Inputs:
  ArrayType a;
  FuncType1 f1;
  FuncType2 f2;

Outputs:
  a[0,0].x; a[0,0].y.0; a[0,0].y.1;
  a[0,1].x; a[0,1].y.0; a[0,1].y.1;
  a[1,0].x; a[1,0].y.0; a[1,0].y.1;
  a[1,1].x; a[1,1].y.0; a[1,1].y.1;

  f1(F, F).0; f1(F, F).1;
  f1(F, T).0; f1(F, T).1;
  f1(T, F).0; f1(T, F).1;
  f1(T, T).0; f1(T, T).1;

  f2(F)(F).0; f2(F)(F).1;
  f2(F)(T).0; f2(F)(T).1;
  f2(T)(F).0; f2(T)(F).1;
  f2(T)(T).0; f2(T)(T).1;
```

7 TYPE CHECKING RULES

This section defines the type checking rules of HLL. An HLL model is considered as correct only if it follows the typing discipline described in this section.

Before presenting the rules, we introduce some preliminary notions used in the type system specification.

7.1 Preliminary definitions

A *static flag* qualifies a stream expression, it distinguishes expressions that can be computed statically by considering constant definitions only, from those that can be computed statically by considering both constants and stream definitions and those that are not static at all. In HLL, where any expression represents a stream, some are used in a context that requires the ability to evaluate them once and for all. These additional constraints are then specified using *static flags*[5] in the corresponding typing rule. The definition below introduces a notation for these flags.

Definition 15 (static flag)**.** *A* static flag b *can take three possible values (*$b \in \{0, 1, 2\}$*) that indicates if an expression is static (*1*), if it is a pure constant and literal*[6] *values combination (*2*) and if it is not static (*0*).* $b_1 \sqcap b_2$ *combines two static flags in the following way:*

b_1	b_2	$b_1 \sqcap b_2$
_	0	0
0	_	0
1	1	1
1	2	1
2	1	1
2	2	2

Static flags are ordered by the relation $\sqsubseteq$ *such that:* $0 \sqsubseteq 1 \sqsubseteq 2$*.*

Note about the separation of static and constants: Intuitively a constant flag (2) represents a notion that is stronger than static since it means "*composed of streams defined in constants sections only*". We distinguish between static and constant expressions in order to allow the type system to detect incorrectly sized arrays. In order to keep the definition of the type system simple, sizes of arrays must be specified using constants. Static expressions are however allowed in other constructs, such as population counts.

Definition 16 (Typing environments)**.** *A typing environment* H *is a partial mapping that associates pairs* (type, static flag) *to identifiers;* $Dom(H)$ *represents the domain of* H*, i.e. the set of identifiers mapped by* H*; when* $x \in Dom(H)$*,* $H(x)$ *represents the type and flags associated to* x *in* H *if any and* `bool` *otherwise:*

$$H(x) = \begin{cases} \tau, b \text{ if } (x : \tau, b) \in H \\ \texttt{bool}, 0 \text{ otherwise.} \end{cases}$$

An environment can be given by extension as the set of pairs that defines the mapping (e.g. $\{x : \texttt{bool}, 0,\ y : \texttt{int}, 2\}$*); the empty environment is denoted* $\{\}$*.*

[5] in a typing rule without explicit constraints on static flags, the involved expressions represent any stream of the specified type.

[6] a *literal* is a syntactical entity belonging either to `<int_literal>` or `<bool_literal>`.

Definition 17 (Environment merging)**.** *Given two environments H_1 and H_2 the merging $H_1 \oplus H_2$, defined if $Dom(H_1) \cap Dom(H_2) = \emptyset$, represents an environment such that:*

$$Dom(H_1 \oplus H_2) = Dom(H_1) \cup Dom(H_2)$$
$$\forall x \in Dom(H_1 \oplus H_2),\ (H_1 \oplus H_2)(x) = \begin{cases} H_1(x) & \text{if } x \in Dom(H_1) \\ H_2(x) & \text{if } x \in Dom(H_2) \end{cases}$$

Definition 18 (Environment hiding)**.** *Given two environments H_1 and H_2 we can build an environment $H_1; H_2$, defined by:*

$$Dom(H_1; H_2) = Dom(H_1) \cup Dom(H_2)$$
$$\forall x \in Dom(H_1; H_2),\ (H_1; H_2)(x) = \begin{cases} H_1(x) & \text{if } x \in Dom(H_1) \\ H_2(x) & \text{otherwise} \end{cases}$$

Definition 19 (Judgements)**.** *To express the type checking rules of an* HLL *system, the following judgements are introduced.*

1. $H \overset{dcl}{\vdash}$ *decl* $: H'$ *states that the declaration list decl is well typed in the typing environment H and defines the typing environment H';*
2. $H \overset{exp}{\vdash}$ *expr* $: (\tau, b)$ *states that the expression expr has type τ and static flag b when typed in the environment H;*
3. $H \overset{lhs}{\vdash}$ *lhs* $: (\tau, b, H')$ *states that the left-hand side lhs is well typed in the environment H, has type τ, static flag b and defines the environment H' that contains the iterator variables type declarations (IV(lhs));*
4. $H \overset{cst}{\vdash}$ *cst* $: H'$ *states that the constant declaration list cst is well typed in the environment H and defines the environment H';*
5. $D, H \overset{def}{\vdash}$ *def* $: H'$ *states that the definition def is well typed and defines the environment H' in the typing environment H and the set of declared variables D;*
6. $H \overset{typ}{\vdash}$ *t states that the type t is legal in the typing environment H (e.g. array bounds are static, structures do not have name conflicts in the names of their fields, etc...);*
7. $H \overset{tdef}{\vdash}$ *typedef* $: H', D$ *states that the type definition typedef is well defined in the environment H and defines the typing environment H' (i.e. introduction in the typing environment of the enumeration and sort values); D maps type identifiers with the corresponding type expressions; type equivalence must be understood modulo this mapping;*
8. $H \overset{pat}{\vdash}$ *pattern* $: (\tau, H')$ *states that the switch case pattern is well typed in the typing environment H, that it matches values of type τ and defines the typing environment H';*
9. $H \overset{\mathcal{D}}{\vdash} D : (\tau, b)$ *states that the domain D is well typed in the typing environment H, that the values it covers has type τ and the set of values it contains has static flag b;*
10. $H \overset{in}{\vdash} inpt : H'$ *states that $inpt$ is a correct input and defines the typing environment H';*
11. $H \overset{out}{\vdash} out$ *states that out is a correct output;*
12. $H \overset{po}{\vdash} po$ *states that po is a correct proof obligation.*

7.2 Typing rules

HLL-7

An HLL model is considered as correct with respect to the typing discipline if there exists a proof tree whose root is the model itself, using the type rules defined in this section.

7.2.1 Typing expressions

$$\frac{H(x) = (\tau, b)}{H \overset{exp}{\vdash} x : (\tau, b)} \textbf{ (context)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, b) \qquad \tau \preceq \tau'}{H \overset{exp}{\vdash} e : (\tau', b)} \textbf{ (type subsumption)}$$

This subsumption rule specifies that an expression of type τ can always safely be considered as an expression of type τ' provided that τ is a subtype of τ' ($\tau \preceq \tau'$).

$$\frac{H \overset{exp}{\vdash} e : (\tau, b) \qquad b' \sqsubseteq b}{H \overset{exp}{\vdash} e : (\tau, b')} \textbf{ (static subsumption)}$$

This subsumption rule specifies that a *static expression* can, if needed, be considered as a *non-static expression* or that a *constant expression* can be considered either as *static* or *non-static*. Same goes for domains:

$$\frac{H \overset{\mathcal{D}}{\vdash} D : (\tau, b) \qquad b' \sqsubseteq b}{H \overset{\mathcal{D}}{\vdash} D : (\tau, b')} \textbf{ (static domain subsumption)}$$

$$\frac{}{H \overset{exp}{\vdash} \texttt{TRUE} : (\texttt{bool}, 2)} \textbf{ (bool literal 1)}$$

$$\frac{}{H \overset{exp}{\vdash} \texttt{FALSE} : (\texttt{bool}, 2)} \textbf{ (bool literal 2)}$$

$$\frac{}{H \overset{exp}{\vdash} l_{\texttt{int}} : (\texttt{int}, 2)} \textbf{ (int literal)}$$

where $l_{\texttt{int}}$ represents an integer literal (token `<int_literal>` in the EBNF).

$$\frac{H \overset{exp}{\vdash} e : (\texttt{bool}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{bool}, b')}{H \overset{exp}{\vdash} e \circ e' : (\texttt{bool}, b \sqcap b')} \textbf{ (bool binop)}$$

where $\circ \in \{$ `#`, `&`, `#!`, `->`, `<->` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{int}, b')}{H \overset{exp}{\vdash} e \circ e' : (\texttt{int}, b \sqcap b')} \textbf{ (int binop)}$$

where $\circ \in \{$ `+`, `-`, `*`, `%`, `^`, `/`, `/>`, `/<` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{int}, b') \qquad 1 \sqsubseteq b'}{H \overset{exp}{\vdash} e \circ e' : (\texttt{int}, b \sqcap b')} \textbf{ (int shift)}$$

where $\circ \in \{$ `>>`, `<<` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{int}, b')}{H \overset{exp}{\vdash} op(e, e') : (\texttt{int}, b \sqcap b')} \textbf{ (int bitwise)}$$

where $op \in \{$ `$and`, `$or`, `$xor` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\tau, b) \qquad H \overset{exp}{\vdash} e' : (\tau, b') \qquad FiniteDom(\tau)}{H \overset{exp}{\vdash} e \circ e' : (\texttt{bool}, b \sqcap b')} \textbf{ (equality relational binop)}$$

where $\circ \in \{$ `=`, `==`, `!=`, `<>` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{int}, b')}{H \overset{exp}{\vdash} e \circ e' : (\texttt{bool}, b \sqcap b')} \textbf{ (order relational binop)}$$

where $\circ \in \{$ `>`, `>=`, `<`, `<=` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{bool}, b)}{H \overset{exp}{\vdash} \sim e : (\texttt{bool}, b)} \textbf{ (bool negation)}$$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b)}{H \overset{exp}{\vdash} \texttt{\$not}(e) : (\texttt{int}, b)} \textbf{ (int bitwise negation)}$$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b)}{H \overset{exp}{\vdash} -e : (\texttt{int}, b)} \textbf{ (int negation)}$$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b) \qquad H \overset{exp}{\vdash} e' : (\texttt{int}, b')}{H \overset{exp}{\vdash} op(e, e') : (\texttt{int}, b \sqcap b')} \textbf{ (min-max)}$$

where $op \in \{$ `$min`, `$max` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, b)}{H \overset{exp}{\vdash} \texttt{\$abs}(e) : (\texttt{int}, b)} \textbf{ (abs)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_)}{H \overset{exp}{\vdash} X(e) : (\tau, 0)} \textbf{ (next)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_)}{H \overset{exp}{\vdash} pre(e) : (\tau, 0)} \textbf{ (pre 1)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau_e, \_) \qquad \tau_e \preceq \tau \qquad H \overset{typ}{\vdash} \tau \qquad Sized(\tau) \qquad noEmptyType(\tau)}{H \overset{exp}{\vdash} pre < \tau > (e) : (\tau, 0)} \textbf{ (pre 2)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_) \qquad H \overset{exp}{\vdash} i : (\tau, \_)}{H \overset{exp}{\vdash} pre(e, i) : (\tau, 0)} \textbf{ (pre 3)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau_e, \_) \qquad H \overset{exp}{\vdash} i : (\tau_i, \_) \qquad H \overset{typ}{\vdash} \tau \qquad \tau_e \preceq \tau \qquad \tau_i \preceq \tau \qquad Sized(\tau) \qquad noEmptyType(\tau)}{H \overset{exp}{\vdash} pre < \tau > (e, i) : (\tau, 0)} \textbf{ (pre 4)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_)}{H \overset{exp}{\vdash} I(e) : (\tau, 0)} \textbf{ (initial)}$$

Note that this rule is only used to type check constraints for which the initial modifier may be added.

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, \_) \qquad H \overset{typ}{\vdash} \tau \qquad \tau \equiv \texttt{int}}{H \overset{exp}{\vdash} cast < \tau > (e) : (\texttt{int}, 0)} \textbf{ (cast)}$$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{bool}\hat{}(n), \_) \qquad H \overset{exp}{\vdash} p : (\texttt{int}, 2) \qquad p \leq n}{H \overset{exp}{\vdash} op(e, p) : (\texttt{int}, 0)} \textbf{ (bin2)}$$

where $op \in \{$ `bin2s`, `bin2u` $\}$

$$\frac{H \overset{exp}{\vdash} e : (\texttt{int}, \_) \qquad H \overset{exp}{\vdash} n : (\texttt{int}, 2)}{H \overset{exp}{\vdash} op(e, n) : (\texttt{bool}\hat{}(n), 0)} \textbf{ (2bin)}$$

where $op \in \{$ `s2bin`, `u2bin` $\}$

$$\frac{H \overset{exp}{\vdash} e : (tuple(\tau_0, \ldots, \tau_n), \_) \qquad i \in [0..n]}{H \overset{exp}{\vdash} e.i : (\tau_i, 0)} \textbf{ (tuple access)}$$

$$\frac{H \overset{exp}{\vdash} e : (struct(l_0 : \tau_0, \ldots, l_n : \tau_n), \_) \qquad i \in [0..n]}{H \overset{exp}{\vdash} e.l_i : (\tau_i, 0)} \textbf{ (struct access)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau\hat{}(d_1, \ldots, d_n), \_) \qquad \forall i \in [1..n], H \overset{exp}{\vdash} e_i : (\texttt{int}, \_)}{H \overset{exp}{\vdash} e[e_1 \ldots e_n] : (\tau, 0)} \textbf{ (array access)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau_1 \times \cdots \times \tau_n \to \tau, \_) \qquad \forall i \in [1..n], H \overset{exp}{\vdash} e_i : (\tau_i, \_)}{H \overset{exp}{\vdash} e(e_1 \ldots e_n) : (\tau, 0)} \textbf{ (function application)}$$

$$\frac{H \overset{exp}{\vdash} c : (\texttt{bool}, b_c) \qquad H \overset{exp}{\vdash} e : (\tau, b) \qquad H \overset{exp}{\vdash} e' : (\tau, b')}{H \overset{exp}{\vdash} \texttt{if } c \texttt{ then } e \texttt{ else } e' : (\tau, b_c \sqcap b \sqcap b')} \textbf{ (if-then-else)}$$

$$\frac{\begin{array}{c}\forall j \in [1..m], e'_j : \tau' \\ \forall i \in [1..n], (H_i^1 \oplus \ldots \oplus H_i^m); H \overset{exp}{\vdash} e_i : (\tau_i, \_) \\ \forall i \in [1..n], j \in [1..m], H \overset{pat}{\vdash} p_i^j : (\tau_i^j, H_i^j) \\ \forall i \in [1..n], j \in [1..m], \tau_i^j \preceq \tau_i \end{array}}{\begin{array}{l} (e_1, \ldots, e_n \\ |\, p_1^1, \ldots, p_n^1 \texttt{ => } e'_1 \\ |\, p_1^2, \ldots, p_n^2 \texttt{ => } e'_2 \qquad : (\tau', 0) \\ |\, \cdots \\ |\, p_1^m, \ldots, p_n^m \texttt{ => } e'_m) \end{array}} \textbf{ (case)}$$

Note that this rule requires to have $\tau_i^j \preceq \tau_i$ while the compatibility (see Definition 9) is enough and in presence of the subsumption rule, it is sometimes possible to satisfy this relation using *type subsumption* rule in order to weaken the types of the e_i. However the subtyping relation implies the compatibility and each time it is violated, it corresponds to trivially dead cases that can be captured during type checking.

$$\frac{H \overset{exp}{\vdash} v : (\tau, 2) \qquad Scalar(\tau)}{H \overset{pat}{\vdash} v : (\tau, \{\})} \textbf{ (pattern value)}$$

$$\frac{H \overset{typ}{\vdash} \tau \qquad Scalar(\tau)}{H \overset{pat}{\vdash} \_ : (\tau, \{\})} \textbf{ (pattern any)}$$

$$\frac{H \overset{typ}{\vdash} T \qquad T \equiv sort\ldots}{H \overset{pat}{\vdash} T\ x : (T, \{x : T, 0\})} \textbf{ (pattern sort)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_) \qquad H \overset{exp}{\vdash} e\, a_1 \ldots a_n : (\tau'', \_) \qquad H \overset{exp}{\vdash} e' : (\tau', \_) \qquad \tau'' \lhd \tau'}{H \overset{exp}{\vdash} (e \texttt{ with } a_1 \ldots a_n := e') : (\tau, 0)} \textbf{ (with)}$$

$$\frac{\forall i \in [1..n], H \overset{exp}{\vdash} e_i : (\texttt{bool}, \_) \qquad H \overset{exp}{\vdash} N : (\texttt{int}, 1)}{H \overset{exp}{\vdash} \texttt{population\_count\_\{eq,lt,gt\}}(e_1, \ldots, e_n, N) : (\texttt{bool}, 0)} \textbf{ (population count)}$$

$$\frac{H \overset{exp}{\vdash} e : (\tau, \_) \qquad H \overset{\mathcal{D}}{\vdash} D : (\tau, \_) \qquad \tau \equiv sort\cdots \vee \tau \equiv \texttt{int}}{H \overset{exp}{\vdash} e \texttt{:} D : (bool, 0)} \textbf{ (elementhood)}$$

Note elementhood rule rejects the case where the type is an enumeration because in this case, the type system does the check and this predicate is statically true.

$$\frac{\forall i \in [1..n], H \overset{\mathcal{D}}{\vdash} D_i : (\tau_i, 1) \qquad \{v_i : \tau_i, 1 \mid i \in [1..n]\}; H \overset{exp}{\vdash} e : (\texttt{bool}, \_) \qquad \forall i, j \in [1..n], i \neq j \Rightarrow v_i \neq v_j}{H \overset{exp}{\vdash} QTF\, v_1 : D_1, \ldots, v_n : D_n\ e : (\texttt{bool}, 0)}$$ **(bool quantifier)**

where $QTF \in \{\texttt{SOME}, \texttt{ALL}, \texttt{CONJ}, \texttt{DISJ}\}$

$$\frac{\forall i \in [1..n], H \overset{\mathcal{D}}{\vdash} D_i : (\tau_i, 1) \qquad \{v_i : \tau_i, 1 \mid i \in [1..n]\}; H \overset{exp}{\vdash} e : (\texttt{int}, \_) \qquad \forall i, j \in [1..n], i \neq j \Rightarrow v_i \neq v_j}{H \overset{exp}{\vdash} QTF\, v_1 : D_1, \ldots, v_n : D_n\ e : (\texttt{int}, 0)}$$ **(int quantifier)**

where $QTF \in \{\texttt{SUM}, \texttt{PROD}, \texttt{\$min}, \texttt{\$max}\}$

$$\frac{H \overset{\mathcal{D}}{\vdash} D : (\tau, 1) \qquad \{v : \tau, 1\}; H \overset{exp}{\vdash} e : (\texttt{bool}, \_)}{H \overset{exp}{\vdash} \texttt{SELECT}\, v : D\ e : (\tau, 0)}$$ **(select-quantifier-1)**

$$\frac{\forall i \in [1..n], H \overset{\mathcal{D}}{\vdash} D_i : (\tau_i, 1) \qquad \{v_i : \tau_i, 1 \mid i \in [1..n]\}; H \overset{exp}{\vdash} e : (\texttt{bool}, \_) \qquad \forall i, j \in [1..n], i \neq j \Rightarrow v_i \neq v_j \qquad n > 1}{H \overset{exp}{\vdash} \texttt{SELECT}\, v_1 : D_1, \ldots, v_n : D_n\ e : (tuple(\tau_1, \ldots, \tau_n), 0)}$$ **(select-quantifier-2)**

$$\frac{H \overset{\mathcal{D}}{\vdash} D : (\tau', 1) \qquad \{v : \tau', 1\}; H \overset{exp}{\vdash} e : (\texttt{bool}, \_) \qquad H \overset{exp}{\vdash} d : (\tau'', \_) \qquad H \overset{typ}{\vdash} \tau \qquad \tau' \preceq \tau \qquad \tau'' \preceq \tau}{H \overset{exp}{\vdash} \texttt{SELECT}\, v : D\ e, d : (\tau, 0)}$$ **(select-quantifier-default-1)**

$$\frac{\begin{array}{c}\forall i \in [1..n], H \overset{\mathcal{D}}{\vdash} D_i : (\tau'_i, 1) \\ \{v_i : \tau'_i, 1 \mid i \in [1..n]\}; H \overset{exp}{\vdash} e : (\texttt{bool}, \_) \\ \forall i, j \in [1..n], i \neq j \Rightarrow v_i \neq v_j \qquad \forall i \in [1..n], H \overset{exp}{\vdash} d_i : (\tau''_i, \_) \\ \forall i \in [1..n], H \overset{typ}{\vdash} \tau_i \qquad \forall i \in [1..n], \tau'_i \preceq \tau_i \wedge \tau''_i \preceq \tau_i \\ n > 1\end{array}}{H \overset{exp}{\vdash} \texttt{SELECT}\, v_1 : D_1, \ldots, v_n : D_n\ e, \{d_1, d_2, \ldots, d_n\} : (tuple(\tau_1, \ldots, \tau_n), 0)}$$ **(select-quantifier-default-2)**

$$\frac{m \geq n \qquad \forall j \in [1..m], H \overset{typ}{\vdash} s_j \qquad \forall i \in [1..n], H \overset{\lambda_{par}}{\vdash} f_i : s_i : H_i \qquad H_1 \oplus \ldots \oplus H_n; H \overset{exp}{\vdash} e : (\tau, \_) \qquad \exists \tau', (\tau')^{(s_1 \ldots s_m)} \equiv (\tau)^{(s_1 \ldots s_n)}}{H_1 \oplus \ldots \oplus H_n; H \overset{exp}{\vdash} \texttt{lambda}\, s_1 \ldots s_m\ :\ f_1 \ldots f_n := e : ((\tau)^{(s_1 \ldots s_n)}, 0)}$$ **(lambda)**

Where the operation $(\tau)^{(s_1 \ldots s_n)}$ is inductively defined by:

$$(\tau)^{([e_1,\ldots,e_n]s_2\ldots s_n)} = (\tau)^{(s_2\ldots s_n)\hat{}}(e_1,\ldots,e_n)$$
$$(\tau)^{((t_1,\ldots,t_n)s_2\ldots s_n)} = t_1 \times \ldots \times t_n \to (\tau)^{(s_2\ldots s_n)}$$
$$(\tau)^{()} = \tau$$

$$\frac{H \overset{typ}{\vdash} (t_1,\ldots,t_n) \qquad \forall i,j \in [1..n], i \neq j \Rightarrow v_i \neq v_j}{H \overset{\lambda_{par}}{\vdash} (v_1,\ldots,v_n) : (t_1,\ldots,t_n) : \{i \in [1..n] \mid v_i : t_i, 1\}} \textbf{ (lambda par function)}$$

$$\frac{H \overset{typ}{\vdash} [e_1,\ldots,e_n] \qquad \forall i,j \in [1..n], i \neq j \Rightarrow v_i \neq v_j}{H \overset{\lambda_{par}}{\vdash} [v_1,\ldots,v_n] : [e_1,\ldots,e_n] : \{i \in [1..n] \mid v_i : int, 1\}} \textbf{ (lambda par array)}$$

$$\frac{H \overset{exp}{\vdash} e_1 : (\texttt{int}, b_1) \qquad H \overset{exp}{\vdash} e_2 : (\texttt{int}, b_2)}{H \overset{\mathcal{D}}{\vdash} [e_1, e_2] : (\texttt{int}, b_1 \sqcap b_2)} \textbf{ (domain range)}$$

$$\frac{H \overset{typ}{\vdash} T \qquad Scalar(T) \qquad Sized(T)}{H \overset{\mathcal{D}}{\vdash} T : (T, 2)} \textbf{ (domain scalar)}$$

$$\frac{H \overset{exp}{\vdash} F : (T, \_) \qquad T = \_ \to T' \vee T = T'\hat{}(\_) \qquad FiniteDom(T)}{H \overset{\mathcal{D}}{\vdash} \texttt{\$items}(F) : (T', 1)} \textbf{ (domain map)}$$

$$\frac{\forall i \in [1..n], H \overset{exp}{\vdash} e_i : (\tau_i, \_)}{H \overset{exp}{\vdash} \{e_1,\ldots,e_n\} : (collection(\tau_1,\ldots,\tau_n), 0)} \textbf{ (collection)}$$

7.2.2 Typing definitions

$$\frac{H(x) = \tau, b}{H \overset{lhs}{\vdash} x : (\tau, b, \{\})} \textbf{ (lhs-var)}$$

$$\frac{H \overset{lhs}{\vdash} a : (\tau\hat{}(d_1,\ldots,d_n), \_, H_a) \qquad \forall i,j \in [1..n], i \neq j \Rightarrow v_i \neq v_j \qquad \forall i \in [1..n], v_i \notin Dom(H_a)}{H \overset{lhs}{\vdash} a[v_1,\ldots,v_n] : (\tau, \_, \{v_i : \texttt{int}, 1 \mid i \in [1..n]\} \oplus H_a)} \textbf{ (lhs-iterator)}$$

$$\frac{H \overset{lhs}{\vdash} f : (\tau_1 \times \cdots \times \tau_n \to \tau, \_, H_f) \qquad \forall i,j \in [1..n], i \neq j \Rightarrow v_i \neq v_j \qquad \forall i \in [1..n], v_i \notin Dom(H_f)}{H \overset{lhs}{\vdash} f(v_1,\ldots,v_n) : (\tau, \_, \{v_i : \tau_i, 1 \mid i \in [1..n]\} \oplus H_f)} \textbf{ (lhs-parameters)}$$

$$\frac{H \overset{lhs}{\vdash} v : (\tau, b, H_v) \qquad H_v; H \overset{exp}{\vdash} e : (\tau', b) \qquad \tau \lhd \tau'}{D, H \overset{def}{\vdash} v := e : \{\}}$$ **(c-definition)**

$$\frac{v \notin D \qquad H \overset{exp}{\vdash} e : (\tau, b) \qquad Scalar(\tau)}{D, H \overset{def}{\vdash} v := e : \{v : \tau, 1 \sqcap b\}}$$ **(c-definition-decl)**

$$\frac{H \overset{exp}{\vdash} e : (\_, \_)}{D, H \overset{def}{\vdash} \_ := e : \{\}}$$ **(throw-away-definition)**

The type-checking of an unfolding definition is performed after its flattening as described in section 8.3.

The three rules below are about memory definitions and all require the declared type of a memory to be sized and to not contain an empty type.

$$\frac{H \overset{lhs}{\vdash} v : (\tau, \_, H_v) \qquad H_v; H \overset{exp}{\vdash} e : (\tau', \_) \qquad \tau \lhd \tau' \qquad Sized(\tau) \qquad noEmptyType(\tau)}{D, H \overset{def}{\vdash} I(v) := e : \{\}}$$ **(i-definition)**

$$\frac{H \overset{lhs}{\vdash} v : (\tau, \_, H_v) \qquad H_v; H \overset{exp}{\vdash} e : (\tau', \_) \qquad \tau \lhd \tau' \qquad Sized(\tau) \qquad noEmptyType(\tau)}{D, H \overset{def}{\vdash} X(v) := e : \{\}}$$ **(x-definition)**

$$\frac{H \overset{lhs}{\vdash} v : (\tau, \_, H_v) \qquad H \overset{exp}{\vdash} e_1 : (\tau_1, \_) \qquad H \overset{exp}{\vdash} e_2 : (\tau_2, \_) \qquad \tau \lhd \tau_1 \qquad \tau \lhd \tau_2 \qquad Sized(\tau) \qquad noEmptyType(\tau)}{D, H \overset{def}{\vdash} v := e_1, e_2 : \{\}}$$ **(l-definition)**

$$\frac{H \overset{exp}{\vdash} e : (\tau, 2)}{H \overset{cst}{\vdash} \tau\ c := e\ : \{c : \tau, 2\}}$$ **(constant)**

$$\frac{D, H \overset{def}{\vdash} def_1 : H_1 \qquad D, H \overset{def}{\vdash} def_2 : H_2}{D, H \overset{def}{\vdash} def_1\ def_2 : H_1 \oplus H_2}$$ **(definitions)**

Note that in this rule, def_1 and def_2 represent several definitions. The rule states that a group of definitions type check correctly if it can be cut in two sub-groups (def_1 and def_2) that type check correctly in the same environment.

7.2.3 Typing declarations

$$\frac{H \overset{typ}{\vdash} \tau \qquad b \sqsubseteq 1}{H \overset{dcl}{\vdash} \tau\ v : \{v : \tau, b\}} \textbf{ (simple declaration)}$$

$$\frac{b \sqsubseteq 1}{H \overset{dcl}{\vdash} v : \{v : \texttt{bool}, b\}} \textbf{ (implicit simple declaration)}$$

$$\frac{H \overset{typ}{\vdash} \tau \qquad \forall i \in [1..n], H \overset{dcl}{\vdash} \tau\ v_i : H_i}{H \overset{dcl}{\vdash} \tau\ v_1, \ldots, v_n : H_1 \oplus \ldots \oplus H_n} \textbf{ (declaration)}$$

$$\frac{\forall i \in [1..n], H \overset{dcl}{\vdash} v_i : H_i}{H \overset{dcl}{\vdash} v_1, \ldots, v_n : H_1 \oplus \ldots \oplus H_n} \textbf{ (implicit declaration)}$$

This rule specifies the declaration of a stream. The constraint on the static flag implies that a stream cannot participate (directly or not) to an expression that would be considered as a constant combination. This is why 2 is not a possible value.

In the following rules, `<param_dim>` represents a list of formal parameter types or array dimensions as allowed by the non-terminal symbol `<name>` in the grammar.

$$\frac{H \overset{dcl}{\vdash} \tau\hat{}(e_1, \ldots, e_n)\ \ a\ \texttt{<param\_dim>} : \{a : \tau', 0\}}{H \overset{dcl}{\vdash} \tau\ a\ \texttt{<param\_dim>}[e_1, \ldots, e_n] : \{a : \tau', 0\}} \textbf{ (array declaration)}$$

$$\frac{H \overset{dcl}{\vdash} (\tau_1 \times \cdots \times \tau_n \to \tau)\ \ f\ \texttt{<param\_dim>} : \{f : \tau', 0\}}{H \overset{dcl}{\vdash} \tau\ f\texttt{<param\_dim>}(e_1, \ldots, e_n) : \{f : \tau', 0\}} \textbf{ (function declaration)}$$

$$\frac{H \overset{dcl}{\vdash} decl_1 : H_1 \qquad H \overset{dcl}{\vdash} decl_2 : H_2}{H \overset{dcl}{\vdash} decl_1\ decl_2 : H_1 \oplus H_2} \textbf{ (declarations)}$$

7.2.4 Typing types

$$\frac{\forall i \in [0..n], H \overset{typ}{\vdash} \tau_i}{H \overset{typ}{\vdash} tuple\{\tau_0, \ldots, \tau_n\}} \textbf{ (tuple)}$$

$$\frac{\forall i \in [1..n], H \overset{typ}{\vdash} \tau_i \qquad \forall i, j \in [1..n], i \neq j \Rightarrow l_i \neq l_j}{H \overset{typ}{\vdash} struct\{l_0 : \tau_0, \ldots, l_n : \tau_n\}} \textbf{ (structure)}$$

$$\frac{H \overset{typ}{\vdash} \tau \qquad H \overset{typ}{\vdash} [e_1, \ldots, e_n]}{H \overset{typ}{\vdash} \tau\hat{}(e_1, \ldots, e_n)} \textbf{ (array)}$$

$$\frac{\forall i \in [1..n], H \overset{exp}{\vdash} e_i : (\texttt{int}, 2)}{H \overset{typ}{\vdash} [e_1, \ldots, e_n]} \textbf{ (dimensions)}$$

$$\frac{H \overset{typ}{\vdash} \tau \qquad H \overset{typ}{\vdash} (\tau_1, \ldots, \tau_n)}{H \overset{typ}{\vdash} \tau_1 \times \cdots \times \tau_n \to \tau} \textbf{ (function)}$$

$$\frac{\forall i \in [1..n], (H \overset{typ}{\vdash} \tau_i \wedge Scalar(\tau_i))}{H \overset{typ}{\vdash} (\tau_1, \ldots, \tau_n)} \textbf{ (parameters)}$$

$$\frac{}{H \overset{typ}{\vdash} \texttt{bool}} \textbf{ (bool)}$$

$$\frac{}{H \overset{typ}{\vdash} \texttt{int}} \textbf{ (int)}$$

$$\frac{H \overset{exp}{\vdash} e_1 : (int, 2) \qquad H \overset{exp}{\vdash} e_2 : (int, 2)}{H \overset{typ}{\vdash} \texttt{int}[e_1, e_2]} \textbf{ (int range)}$$

$$\frac{H \overset{exp}{\vdash} e : (int, 2) \qquad e > 0}{H \overset{typ}{\vdash} \texttt{int signed}\, e} \textbf{ (int signed)}$$

$$\frac{H \overset{exp}{\vdash} e : (int, 2) \qquad e \geq 0}{H \overset{typ}{\vdash} \texttt{int unsigned}\, e} \textbf{ (int unsigned)}$$

7.2.5 Typing type definitions

$$\frac{}{H \overset{tdef}{\vdash} enum\{l_1, \ldots, l_n\}\ t : \{l_1 : t, 2, \ldots, l_n : t, 2\}, \{t \mapsto enum(t; l_1, \ldots, l_n)\}} \textbf{ (enum definition)}$$

$$\frac{\forall i \in [1..n], \left(l_i \in L\ \wedge\ \forall j \in [1..n], i \neq j \Rightarrow l_i \neq l_j\right)}{H \overset{tdef}{\vdash} sort\{l_1, \ldots, l_n\}\ <\ t : \{l_1 : t, 2, \ldots, l_n : t, 2\}, \{t \mapsto sort(t; L; \ldots)\}} \textbf{ (sort contribution 1)}$$

$$\frac{\forall i \in [1..n], S_i \equiv sort \ldots \wedge\ S_i \in Sub}{H \overset{tdef}{\vdash} sort\ S_1, \ldots, S_n\ <\ t : \{\}, \{t \mapsto sort(t; \ldots; Sub)\}} \textbf{ (sort contribution 2)}$$

$$\frac{H \overset{typ}{\vdash} \tau}{H \overset{tdef}{\vdash} \tau\ t_1, \ldots, t_p : \{\}, \{t_1 \mapsto \tau, \ldots, t_p \mapsto \tau\}} \textbf{(non-enum definition)}$$

$$\frac{H \overset{dcl}{\vdash} \tau\hat{}(e_1, \ldots, e_n)\ t\texttt{<param\_dim>} : \{\}\{t \mapsto \tau'\}}{H \overset{tdef}{\vdash} \tau\ t\texttt{<param\_dim>}[e_1, \ldots, e_n] : \{\}, \{t \mapsto \tau'\}} \textbf{(array-type definition)}$$

$$\frac{H \overset{dcl}{\vdash} t_1 \times \cdots \times t_n \to \tau\ t\texttt{<param\_dim>} : \{\}\{t \mapsto \tau'\}}{H \overset{tdef}{\vdash} \tau\ t\texttt{<param\_dim>}(t_1, \ldots, t_n) : \{\}, \{t \mapsto \tau'\}} \textbf{(function-type definition)}$$

$$\frac{H \overset{tdef}{\vdash} tdef_1 : H_1, D_1 \qquad H \overset{tdef}{\vdash} tdef_2 : H_2, D_2}{H \overset{tdef}{\vdash} tdef_1\ tdef_2 : H_1 \oplus H_2, D_1 \oplus D_2} \textbf{(type definitions)}$$

7.2.6 Typing the entire model

$$\frac{\begin{array}{c} H \overset{dcl}{\vdash} inpt : \{\tau, \_\} \\ Sized(\tau) \qquad FiniteDom(\tau) \\ HasOrderedDomain(\tau) \qquad noEmptyType(\tau)) \end{array}}{H \overset{in}{\vdash} inpt : \{\tau, \_\}} \textbf{(input)}$$

$$\frac{\begin{array}{c} H \overset{exp}{\vdash} out : (\tau, \_) \\ FiniteDom(\tau) \qquad HasOrderedDomain(\tau) \end{array}}{H \overset{out}{\vdash} out} \textbf{(output)}$$

$$\frac{\begin{array}{c} H \overset{exp}{\vdash} po : (\tau, \_) \\ FinalBaseType(\tau) = \texttt{bool} \\ FiniteDom(\tau) \qquad HasOrderedDomain(\tau)\big) \end{array}}{H \overset{po}{\vdash} po} \textbf{(proof obligation)}$$

$$\frac{\begin{array}{l}\forall cst \in Constants(M),\ H_{Constants} \overset{cst}{\vdash} cst : H_{cst} \\ \forall tdef \in Types(M),\ H_{Constants} \overset{tdef}{\vdash} tdef : H_{tdef}, D \\ \forall inpt \in Inputs(M),\ H_{Constants} \overset{in}{\vdash} inpt : H_{inpt} \\ \forall dcl \in Declarations(M),\ H_{Constants} \overset{dcl}{\vdash} dcl : H_{dcl} \\ \forall def \in Definitions(M),\ D, H \overset{def}{\vdash} def : H_{def} \\ \forall cstr \in Constraints(M),\ H \overset{exp}{\vdash} cstr : (\texttt{bool}, \_) \\ \forall po \in Proof\_obligations(M),\ H \overset{po}{\vdash} po \\ \forall out \in Outputs(M),\ H \overset{out}{\vdash} out \\ \text{where } H_{Constants} = \bigoplus_{cst \in Constants(M)} H_{cst} \\ \text{and } H = \begin{array}{l} H_{Constants} \oplus (\bigoplus_{tdef \in Types(M)} H_{tdef}) \oplus (\bigoplus_{inpt \in Inputs(M)} H_{inpt}) \\ \oplus (\bigoplus_{dcl \in Declarations(M)} H_{dcl}) \\ \oplus (\bigoplus_{def \in Definitions(M)} H_{def}) \end{array} \\ \text{and } D = \begin{array}{l} Dom((\bigoplus_{cst \in Constants(M)} H_{cst}) \oplus (\bigoplus_{inpt \in Inputs(M)} H_{inpt}) \\ \oplus (\bigoplus_{dcl \in Declarations(M)} H_{dcl})) \end{array}\end{array}}{\text{model } M \text{ is well typed}}\ \textbf{(system)}$$

8 ADDITIONAL STATIC CHECKS

This section specifies checks that are neither covered by the grammar nor by the type checking.

Definition 20 (Definition forms)**.** *An item of the* `definitions` *section can have one of the following forms:*

name	*syntactic form*	*contribution to the stream definition*
combinatorial	`a := <rhs>`	*initial and next*
unfolding	`a1,...,an := <rhs>`	*initial and next (see section 8.3)*
initial	`I (a) := <rhs>`	*initial*
next	`X (a) := <rhs>`	*next*
memory	`a := <rhs>, <rhs>`	*initial and next*

A stream that is used in an expression but never appears on the left-hand side of a definition (neither initial nor next) is considered as an input.

8.1 Partial stream definition

HLL-8

Restriction 1 (partial definition)**.** *A stream that has an initial definition or that is declared as initial in the inputs section must have a* next definition*.*

Note that a stream that has a next definition may lack an initial one.

8.2 Unicity of stream definitions

HLL-10

Restriction 2 (unicity of the definition)**.** *A stream can have, at most, one* initial definition *and one* next definition*. i.e. a stream can neither have two definitions contributing to its initial value specification nor two definitions contributing to its next value specification.*

Restriction 3 (declared inputs)**.** *A stream declared in an* `inputs` *section cannot appear on the left-hand side of a definition, except when it is declared as initial in an* `inputs` *section, in which case it must have a next definition but no initial definition.*

8.3 Unfolding stream definitions

Unfolding definitions, in which multiple identifiers (potentially including “throw-aways”) are on the left-hand-side of the assignment are equivalent to the flattened version with multiple individual *combinatorial* definitions. For example:

```
a,b,_,c := <expr>;
```

with `<expr>` a 4 element array, is equivalent to the flattened version:

```
__tmp := <expr>;
a := __tmp[0];
```

```
b := __tmp[1];
_ := __tmp[2]; // this is a throw-away, doing nothing
c := __tmp[3];
```

Such definitions can be applied to any composite right-hand-side expression provided that this expression has exactly the same number of top-level elements as the number of items on the left-hand-side (including throw-aways), and that its type has an ordered domain (*i.e.* not functions on sorts). The order in which the composite is unfolded relies on the definition given in section 6.12. Standard type-checking applies on the individual combinatorial definitions.

8.4 Function Assignability

HLL-32

Restriction 4. *When assigning a function to another function, corresponding parameter types shall resolve to the same basic scalar types. For boolean, enumerateds, and sorts, this is already captured by the subtyping relation (see Definition 13). However, the same is also required for integer parameters. Their types shall resolve to identical integer ranges.*

8.5 Named type references and definitions

HLL-11

Restriction 5. *All the referenced named types must be defined in the model. A named type definition cannot reference itself, directly or indirectly, in the type expression (non-terminal* `<type>` *in the EBNF) that defines it.*

8.6 Scoping rules (or namespaces)

HLL-12

Restriction 6 (conflicts in the stream namespace)**.** *In the top level namespace of streams (see Section 5) a given identifier represents a unique stream. As a consequence:*

- *an identifier* `v` *can be declared only once;*
- *an enumeration or sort value* `v` *can only appear in one enumerated type or one sort contribution;*
- *an identifier* `v` *used as an enumeration or sort value cannot be declared in any stream declaration section nor defined in any definition section;*
- *an identifier used as an iterator variable in an array definition or as a formal parameter in a function definition must be unique within the definition, for instance it is not allowed to write* `a[i, i] := ...` *or* `f(x, x) :=...;`
- *an identifier used as a quantification variable must be unique among the variables introduced by the quantifier, for instance* `ALL i:[1,2] i:[4,2] ...` *is not allowed.*

Restriction 7 (unicity of named types)**.** *A named type can be defined only once.*

8.7 Well-definedness of constants

HLL-32

Restriction 8. *All values for which typing rules require a static flag 2 (constants) must be well defined according to the rules described in section 16. It is for example not allowed to use 3/0 as the number of bits of an integer type or as an array dimension, as euclidian division is not defined if the divisor is 0.*

This restriction does not apply to constants that are used exclusively in other contexts, for example as stream values.

9 SORTS: A HIERARCHY OF ENUMERATIONS

Sorts are a particular kind of user defined types that can be seen as hierarchized (in the sense of set inclusion) finite sets of enumerated values. These sets can be understood as sets of object instances and the subset relation as the inheritance relation (*class A inherits class B* also means that *the set of all the object instances of class B contains all the object instances of class A*).

9.1 Specifying a sort hierarchy

HLL-28

Sorts are defined in the `types` sections. A sort definition is composed of several partial definitions called *contributions*. All the contributions of a sort appear in the same namespace and these contributions can be spread all over the model[7].

Contributions are of one of the two following forms:

1. those that give the inclusion relation with other defined sorts [8] :

   ```
   sort hotColor, coldColor < color;
   ```

   An equivalent formulation is:

   ```
   sort coldColor < color;
   sort hotColor  < color;
   ```

2. those that specify values introduced by the sort:

   ```
   sort {red, yellow} < hotColor;
   ```

   Several disjoint sets of values can be specified for a unique sort, for instance a different still equivalent form to specify the `hotColor` values is:

   ```
   sort {yellow} < hotColor;
   sort {red}    < hotColor;
   ```

3. and those that only introduce a sort:

   ```
   sort color;
   ```

As an example, here is a sort based description of a *playing card deck* where the values are all the cards of the deck:

```
types:
  sort Reds,   Blacks   < Deck;
  sort Spades, Clubs    < Blacks;
  sort Hearts, Diamonds < Reds;
  sort {S_A, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_10, S_J, S_Q, S_K} < Spades;
  sort {C_A, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_10, C_J, C_Q, C_K} < Clubs;
  sort {H_A, H_2, H_3, H_4, H_5, H_6, H_7, H_8, H_9, H_10, H_J, H_Q, H_K} < Hearts;
  sort {D_A, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9, D_10, D_J, D_Q, D_K} < Diamonds;
```

[7] Note this is a consequence of the syntax that does not allow to specify a path in a type definition and of the scoping rule 3 of section 5.2.

[8] To be understood as: *the set of* color *contains both the* hotColor *and the* coldColor.

Using the type notation introduced in 6, the defined sorts are:

$$sort(\texttt{Blacks};;\texttt{Clubs},\texttt{Spades})$$
$$sort(\texttt{Reds};;\texttt{Hearts},\texttt{Diamonds})$$
$$sort(\texttt{Spades};\texttt{S\_A},\ldots,\texttt{S\_K};)$$
$$sort(\texttt{Clubs};\texttt{C\_A},\ldots,\texttt{C\_K};)$$
$$sort(\texttt{Hearts};\texttt{H\_A},\ldots,\texttt{H\_K};)$$
$$sort(\texttt{Diamonds};\texttt{D\_A},\ldots,\texttt{D\_K};)$$

9.2 Sorts and the switch-case expression

HLL-29

HLL provides a generalised switch case construct that allows to:

- specify a case based on a tuple of inspected expressions and
- capture several cases in a single line pattern using one or more wildcards in the pattern tuple and the sort hierarchy.

The specified cases may overlap. They are inspected *sequentially* in the order they appear. The selected branch is the first pattern that matches the inspected value.

To illustrate the usage of wildcards and tuples based selection the example below gives the truth table of the implication:

```
a_implies_b := ( a,     b
               |  false, _      => true
               |  true,  true   => true
               |  true,  false  => false);
```

This formulation of the truth table is not sensitive to branch order because they are all disjoint, thus it makes no use of the sequential evaluation of the switch-case.

Here is a second form where branches overlap and their relative position matters:

```
a_implies_b := ( a,     b
               |  false, _      => true
               |  _,     true   => true
               |  _,     _      => false);
```

The last line matches any couple of values, but its position makes it a global default. The second line must be understood as follows: in the case the first line does not match the value the second component's value is sufficient to define the result.

The hierarchy of the example below illustrates patterns specification with sorts. The first pattern does not capture the matched value (wildcard _) while the second one captures it in variable `c`:

```
types:
  sort hotColor, coldColor         < color;
  sort {red, yellow, brown}        < hotColor;
  sort {blue, green, white, black} < coldColor;

inputs:
```

```
  color aColor;

definitions:
  is_dark :=
  (aColor
  | black       => true
  | coldColor _ => false /* for all the hotColor but black */
  | hotColor c  => /* c is a local identifier that captures the value of
                      aColor but with the more precise type: hotColor. */
      ( c
      | red    => true
      | yellow => false
      | brown  => false) /* this case is known to be exhaustive
                            because c has type hotColor. */
  );

outputs:
  is_dark;
```

HLL-31

Restriction 9 (case exhaustivity)**.** *In a correct* HLL *model, all the cases are exhaustive.*

Note: Tools implementing HLL have to provide a way to check the exhaustivity of the list of patterns *i.e.* check that all the possible values taken by the inspected expressions are covered.

10 MAPPING SEMANTICS

HLL-13

Mapping is the general concept behind arrays and functions. Both arrays and functions share the same semantics in HLL *i.e.* they are both mappings from a finite domain to HLL streams. They mainly differ in the way they are declared:

- an array type is defined by the *type* of its elements and the *sizes* of its dimensions (`bool A[5,1]`);
- a function type is defined by the *types* of its parameters and the *type* of the streams it defines.

They also differ in the syntax of the mapping application:

- an access to a stream defined by an array is called a *projection* and made using square brackets as follows: `A[21,42]`;
- an access to a stream defined by a function is called an *application* and made using parentheses as follows: `f(25,a)`.

There is also a difference in the typing of the accessor that must be an integer for an array while it is only required to be scalar for a function.

The reason for the presence of these two close concepts in HLL is that HLL aims at providing constructs that are not only high level but also close to the end user's intention.

10.1 Arrays

HLL provides a way of defining a multidimensional array of items as a (possibly recursive) function over integer indices. For instance the definition `a[i] := i` defines the content of the array `a` by the assignment of i to the i^{th} array cell:

```
  a := { 'a[0]', 'a[1]', 'a[2]', ... };
```

with

```
  'a[0]' := 0;
  'a[1]' := 1;
  'a[2]' := 2;
...
```

The dimension (let say `n`) of `a` is given in its declaration, so this definition *in intension* can be finitely unfolded (since the dimension is finite) to obtain the equivalent definition *in extension*; *i.e.* the previous definitions can be continued:

```
...
  'a[n-2]' := n-2;
  'a[n-1]' := n-1;
```

An array access out of the bounds is not defined, it takes the exceptional undefined value nil (see section 14.11).

Another example, let `odd` be an array of size n such that `odd[i]` contains value `TRUE` if `i` is odd and `FALSE` otherwise. This array can be defined by:

```
  odd[i] := i%2=1;
```

It can also be defined by the recursive definition:

```
  odd[i] := if   i = 0 then FALSE
            elif i = 1 then TRUE
            else            odd[i - 2];
```

These definitions are correct and both define the same array content. The second one is recursive in the sense that the definition of the i^{th} element is based on the $(i-2)^{th}$ one. It can be finitely unfolded considering a lazy interpretation of the `if-then-else` expression, *i.e.* if one can prove that the condition is always true or always false, the unfolding can ignore the unselected branch (see 10.3 for the list of lazy operators). This allows to introduce base cases (0 and 1 in this example) in order to build a well founded recursive definition.

The conjunction `&` and the disjunction `#` can also be used to introduce these base cases. This means that in this unfolding operation, `&` (resp. `#`) is interpreted as a sequential *and then* (resp. *or else*) operator. For instance, the previous example can be rewritten as:

```
  odd[i] := i = 1 # i >= 2 & odd[i - 2];
```

Last, the implication `->` is also interpreted lazily in the unfolding process. For instance, `odd` can be rewritten as:

```
  odd[i] := ~(i <> 0 -> (i >= 1 -> odd[i - 1]));
```

An array can also have a memory definition, this is thus an array of memories. An example mixing recursive definition and array of memories is the one of a *sliding window* on a stream. A sliding window of size `N` on an input `a` is an array `SW` containing previous values of `a` (the one at index `0` is the value of `a` at the previous step). It is defined by:

```
constants:
  int N := 10;

inputs:
  bool a;

declarations:
  bool SW[N];

definitions:
  SW[i] := false,
           if i = 0 then a else SW[i-1];

outputs:
  SW;
```

Note that `SW` memories contain `false` value as long as it refers to a previous cycle that did not exists yet.

In order to have a powerful language for such recursive definitions, HLL semantics on streams does not consider arrays; it is defined on scalars and array definitions are considered lazily (on demand), when a particular array item is needed for some outputs, constraints or proof obligations.

Array declarations can be of one of the two equivalent forms:

- `bool A[10, 20];` or
- `bool^(10,20) A;`

10.2 Function

The notion of arrays indexed by integers is extended to the one of functions taking as parameters any scalar values (values of type `bool`, `int`, enumerated types, and sorts). A function `f` is characterized by the property: *for each cycle,* `a = b` $\Rightarrow$ `f(a) = f(b)` *regardless of the history (past or future values) of* `a` *and* `b`.

In this sense, `f` can be understood as a stream of combinatorial functions. Another way to understand functions in HLL is to see them as generalized truth table (not only for the boolean case), as a table indexed by the values of its parameters (remember the domains are finite); then the function corresponds to a stream of *truth table*.

A function is defined, like an array, by a combinatorial definition or a memory definition. A function returns a single stream *i.e.* all the functions have a type: `t f(t1, t2, ..., tn)` where `t` is any HLL type and all the `ti` are scalar types.

A function access out of the domains of its parameters is not defined, it takes the exceptional undefined value nil (see section 14.11).

Example of a function definition:

```
declarations:
  int Fibonacci(int);

definitions:
  Fibonacci (i) := if i <= 2 then 1 else Fibonacci (i - 1) + Fibonacci (i - 2);
```

A function declaration can be of one of the two equivalent forms:

- `bool f(T1, int);` or
- `(T1  * int -> bool) A;`

A consequence of the fact a function cannot access the past or future of the streams it applies to is the impossibility to write, for instance, a *rising edge detection* using a function. One could be tempted by this formulation:

```
declarations:
  bool bad_rising_edge(bool);

definitions:
  bad_rising_edge(a) := ~a != ~X(a);
```

The formal parameter `a` in the *expression-body* of the function is considered static (see typing rule (lhs-parameters), the static flag of the i_k is 1 which means static). Thus `X(a)` is the same stream as `a`

and `bad_rising_edge(a)` is always *false* and cannot detect rising edges. This function looks like a stream operator one could want to implement, but it's not a good use of functions.

10.3 Making recursive definitions terminate

We have seen that arrays and functions can be defined by recursion *i.e.* the stream they represent for a given effective value v (projection index or parameter value) may depend on the stream they represent at another point. To effectively define a stream, such a recursive definition must terminate *i.e.* admit a finite unfolding for any finite effective parameter.

In a declarative language such as HLL, this unfolding can terminate only if there exist some operators that can provide a value without the need to have all their parameters values (so-called *lazy operators*).

We provide here a table containing the HLL operators allowing to cut definitions with a tag • on the parameters which value is always needed (*strict tag*) and a tag ◦ for those that may not be known (*lazy tag*):

operator name	• strict / ◦ lazy tag profile
logical and	• `&` ◦
logical or	• `#` ◦
logical implication	• `->` ◦
if-then-else	`if` • `then` ◦ `else` ◦
switch-case	`(` • `|` • `=>` ◦ `|` • `=>` ◦ `...)`

10.4 Note about causality in the presence of mappings

This split between the arrays and the scalar streams avoid defining causality on the full HLL language (see Section 15). Such a definition would have been impossible on the full language without introducing raw restrictions on the accepted array definition schemes making modelling with HLL harder. There is a drawback in the fact that the capacity to implement lazy strategies is tool dependent, thus any model containing array definitions must be interpreted as: *there exists an expansion of the array definitions present in the model such that we can build a logically equivalent unfolded one (thus on scalar streams).* This *existential* quantification is resolved in practice by a preprocessing of the model (that we call array expansion) that may fail to produce the scalar model, but the global approach to make proofs is still safe since a tool cannot deduce erroneous facts on a model that it fails to expand.

11 DEFINING COMPOSITES USING COLLECTIONS

Collections can be used to define composites, following the assignability rules defined in section 6.10. Tuples, structures, arrays and functions for which all parameters have a sized and ordered type can be defined using collections.

The rules defined below apply recursively when defininining composites of composites using collections of collections.

11.1 Tuples

A tuple is an ordered collection of n streams $s_0, s_1, \ldots, s_{n-1}$. The order of the fields in a tuple is the order in which the elements are parsed. Such a tuple is defined by a collection of n streams $\{c_0, c_1, \ldots, c_{n-1}\}$ with $s_i := c_i$ $(0 \leq i < n)$.

11.2 Structures

A struct is an ordered collection of n fields $f_0, f_1, \ldots, f_{n-1}$. The order of the fields in a structure is the order in which the fields are parsed. Such a structure is defined by a collection of n streams $\{c_0, c_1, \ldots, c_{n-1}\}$ with $f_i = c_i$ $(0 \leq i < n)$.

11.3 Arrays

The assignment semantics for arrays are defined in an inductive way on the number of dimensions.

If A is a one dimensional array of size n, A is defined by a collection of n streams $\{c_0, c_1, \ldots, c_{n-1}\}$ with $A[i] := c_i$ $(0 \leq i < n)$.

If A is an array with N finite dimensions of sizes $d_0, d_1, \ldots, d_{N-1}$ and $N > 1$, A is defined by a collection of d_0 arrays $\{A_0, A_1, \ldots, A_{d_0-1}\}$, where each A_i $(0 \leq i < d_0)$ is an $(N-1)$-dimensional array of dimensions $d_1, d_2, \ldots, d_{N-1}$, with the relation $A[i_0, i_1, \ldots, i_{N-1}] = A_{i_0}[i_1, i_2, \ldots, i_{N-1}]$. The A_i "sub arrays" can themselves be defined by collections following the same rule.

11.4 Functions

The assignment semantics for functions are defined in an inductive way on the number of parameters.

Functions that can be defined by collections have parameter types which are scalar, bounded and ordered. The ordering rules for the various types of HLL are described in section 6.11. We can for each parameter type define a ranking function. If T is a bounded and ordered type, $rank_T$ is its ranking function, that is, $rank_T$ is a bijection between T and the integer range $[0, Card(T) - 1]$ such that $\forall (i, j) \in T \times T, i < j \Leftrightarrow rank_T(i) < rank_T(j)$.

If f is a function of one parameter p of type T, f is defined by a collection of $Card(T)$ streams $\{c_0, c_1, \ldots, c_{Card(T)-1}\}$ by the relation $f(p) = c_{rank_T(p)}$.

If f is a function of N parameters $p_0, p_1, \ldots, p_{N-1}$ of types $T_0, T_1, \ldots, T_{N-1}$ with $N > 1$, f is defined by a collection of $Card(T_0)$ functions $\{f_0, f_1, \ldots, f_{Card(T_0)-1}\}$, where each f_i is a function of $N-1$ parameters, with the relation $f(p_0, p_1, \ldots, p_{N-1}) = f_{rank_{T_0}(p_0)}(p_1, p_2, \ldots, p_{N-1})$. The f_i "sub functions" can themselves be defined by collections following the same rule.

11.5 Example

The following example illustrates using collections for defining various kinds of composites.

```
Types:
  tuple {bool, int} TupleType;
  tuple {TupleType, TupleType} NestedTupleType;
  struct {a: bool, b: int} StructType;
  int^(2) Array1D;
  int^(2, 2) Array2D;
  (bool -> int) Func1;
  (bool * bool -> int) Func2;

Declarations:
  TupleType t;
  NestedTupleType nt;
  StructType s;
  Array1D arr1d;
  Array2D arr2d;
  Array1D nested_arr1d[2];
  Func1 f1;
  Func2 f2;
  Func1 f3(bool);

Definitions:
  t := {false, 3};
  nt := {{true, 5}, t};
  s := {true, 7};
  arr1d := {1, 2};
  arr2d := {arr1d, {3, 4}};
  nested_arr1d := {arr1d, {5, 6}};
  f1 := {0, 1};
  f2 := {f1, {2, 3}};
  f3 := {f1, {2, 3}};
```

12 LOCAL BINDERS

HLL-26

HLL allows to specify the quantification of a variable over finite domains, providing a compact way to write a formula. Syntactically a quantification is a local binder that introduces an *index identifier*, a *domain* specifying the values the index can take, a *sub-expression* and an *operation*. The domain can be given by a mapping stream, as described in 12.4.

HLL provides three kinds of quantifiers:

- boolean quantifiers;
- integer quantifiers;
- the selection operator.

In boolean and integer quantifiers, the operator combines the value the sub-expression takes on each point of the domain. The selection operator, instead of returning a boolean or integer value, returns a value in the domain of the indexes.

In this document, *quantification* will designate the usual boolean quantification as well as the other kinds of quantification.

12.1 Quantifying over scalar types (built-in and user-defined)

It is possible to quantify over any scalar type (built-in or user-defined) provided that it is finite. The quantification ranges over every value of the given type.

In the case of a finite integer type, this quantification is equivalent to a quantification over the range of this type, for instance:

```
Types:
int [0,9] my_int_0_9;
Outputs:
SUM i:int signed 8 (i);
SUM i:int [0,9] (i);
SUM i:my_int_0_9 (i);
```

is equivalent to:

```
Outputs:
SUM i:[-128,127] (i)
SUM i:[0,9] (i)
SUM i:[0,9] (i)
```

Semantic of the quantification over integer ranges is described in section 12.2.

In the case of a named enumeration or sort, this quantification is equivalent to a quantification on the actual enumeration or sort, described in section 12.3.

In all other cases, the quantification ranges over every value of the given type (necessarily finite).

12.2 Quantifying over integer ranges

This section gives, using examples, the principles of quantification in HLL. As a first example, let us define the boolean expression that is `true` if the array `A` of size `10` contains an even integer can be written in the following way :

```
contains_even := SOME i:[0,9] (A[i] % 2 = 0);
```

this corresponds to an *existential* quantification. This equation could be rewritten without quantifier:

```
contains_even :=
  (A[0] % 2 = 0) # (A[1] % 2 = 0) # (A[2] % 2 = 0) # (A[3] % 2 = 0) #
  (A[4] % 2 = 0) # (A[5] % 2 = 0) # (A[6] % 2 = 0) # (A[7] % 2 = 0) #
  (A[8] % 2 = 0) # (A[9] % 2 = 0);
```

Another example is the boolean that is `true` if all the even indices of `A` contains an even integer value:

```
evens_contain_even := ALL i:[0,9] (i % 2 = 0 -> A[i] % 2 = 0);
```

That is equivalent to:

```
evens_contain_even :=
  (A[0] % 2 = 0) & (A[2] % 2 = 0) & (A[4] % 2 = 0) &
  (A[6] % 2 = 0) & (A[8] % 2 = 0);
```

The examples illustrate the two quantifiers `SOME` (∃) and `ALL` (∀), that are standard in logic. They correspond, as we can see in the example, to an iteration of the boolean *or* (`#`) for the first and *and* (`&`) for the second. HLL provides synonyms that help capturing user intentions in formulas : `CONJ` for `ALL` and `DISJ` for `SOME`.

In the case when the quantification domain is empty, the result is the neutral element of the iterated boolean operator, *i.e.* `true` for `ALL` and `CONJ` and `false` for `SOME` and `DISJ`.

12.3 Quantifying over enumerations

In the previous examples, the quantified variable iterates over an integer range, it is also possible to make it iterate over the values of an enumeration or a sort. For instance checking that all the values of an enumeration are present in an array can be expressed in the following way:

```
types:
  enum {green, yellow, red} cool_color;

inputs:
  cool_color A[10];

definitions:
  has_all_color := ALL c:cool_color SOME i:[0,9] (A[i] = c);

outputs:
  has_all_color;
```

It is also allowed to use a sort identifier as a quantifier domain specification, in this case, the values to consider are all the values defined for this sort and all its subsorts.

Here is an example with a hierarchy of sorts. If `picture` is a square matrix of colors (or pixels), we specify here a property of this matrix that expresses the fact *below the first diagonal all the pixels are black, white or grey and above all the bright colors appear at least once*:

```
types:
  sort {green, blue, red}       < cool_color;
  sort {black, grey, white}     < bw_color;
  sort {yellow, cyan}           < light_color;
  sort bw_color, bright_color   < color;
  sort light_color, cool_color  < bright_color;

inputs:
  color picture[10, 10];

definitions:
  picture_property :=
    ALL i:[0,9] (ALL j:[0,i] SOME c:bw_color (picture[i, j] = c))
   &
    ALL c:bright_color (SOME i:[0,9] SOME j:[i+1,9] (picture[i, j] = c));

outputs:
  picture_property;
```

12.4 Quantifying over mapping streams

It is possible to quantify on a named mapping stream (*i.e.* an array or a function) provided that it has a finite number of components. This condition is always valid for an array, and for functions it is true if and only if each of its parameters ranges over a finite domain. The quantification ranges over the arguments of the mapping stream.

For instance, with the declaration `bool A[10]` , the expression: `ALL i:[0,9] (A[i]);` is equivalent to `ALL a:$items(A) (a);`

Same goes with finite domain functions: with the declaration `bool F(OBJECT1,OBJECT2)` , the expression: `ALL o1:OBJECT1, o2:OBJECT2 (F(o1,o2))` is equivalent to `ALL f:$items(F) (f)`

12.5 Arithmetic extensions

There are standard binders that apply on arithmetic expressions in a way similar to boolean quantification. HLL provides the following operators: `SUM`, `PROD`, `$min` and `$max`. `SUM` (resp. `PROD`) admits the neutral element `0` (resp. `1`) and thus can be used even when the quantification domain is empty.

Operators `$max` and `$min` don't have such neutral values; as a consequence they cannot be applied when the quantification domain is empty. The quantification domain is either a type name (sort or enum) or a static range, checking whether the domain is empty or not is a static property.

For instance, given a two-dimensional array `V` of sizes N and M, computing the sum of the max of each column (with the convention that the second dimension is the column) is quite easy:

```
sum_of_max_col := SUM j:[0,M-1] ($max i:[0,N-1] (V[i, j]));
```

12.6 Selection operator

The selection operator is a quantifier evaluating to a tuple formed by the iteration domain elements for which a given predicate is true.

When several such tuples exist, the result is undefined (see below for a specificity of selection over mapping streams).

If none of the iterated tuples satisfies the given predicate the result is undefined unless a default value has been provided to the select operator, in which case the result is this default value. The default value shall be a tuple of compatible type.

In both cases where the result is undefined, the select takes the exceptional undefined value nil (see section 14.11).

In the special case where the iteration is done on a single variable, the type of the result (and of the default value if any) is directly that of the domain (instead of a singleton tuple).

Apart from the various checks, when correctly defined the selection operator is semantically equivalent to a nested if-then-else over the elements of the iterated domain(s), as illustrated below.

```
//unrolled SELECT
   SELECT i:[0,4] (P(i)) == if   P(0) then 0
                            elif P(1) then 1
                            elif P(2) then 2
                            elif P(3) then 3
                            else           4

//unrolled SELECT with default
   SELECT i:[0,4] (P(i),d) == if   P(0) then 0
                              elif P(1) then 1
                              elif P(2) then 2
                              elif P(3) then 3
                              elif P(4) then 4
                              else           d
```

Note on mapping streams: The `SELECT` quantifier over a mapping stream can be understood as a `SELECT` over the arguments of the mapping stream, meaning that if several mapped values satisfy the `SELECT` condition, the unicity constraint will be violated. For this reason, in the following model, both the `s1` selection and the `s2` selection are incorrect.

```
Declarations:
 int A[3];
Definitions:
 A  := {1,1,2}
```

```
s1 := A[SELECT i:[0,2] (A[i]==1)];
s2 := SELECT a:A (a==1);
```

12.7 Summary of quantifier semantics

Apart from the `SELECT` quantifier which has a specific semantic described in section 12.6, the semantics of the quantifiers is summed up in this table:

Quantifier	Corresponding associative/commutative HLL **binary operator**	Value for **empty domain**
`ALL`	`&`	`true`
`CONJ`	`&`	`true`
`SOME`	`#`	`false`
`DISJ`	`#`	`false`
`SUM`	`+`	`0`
`PROD`	`*`	`1`
`$max`	`$max`	*undefined*
`$min`	`$min`	*undefined*

12.8 Anonymous function and array definition (`lambda`)

Another kind of local binder is the *anonymous* definition of an array or function in HLL. Section 10 introduces arrays and function definition, with the need to explicitly declare and name the mapping and then define it.

```
declarations:
  int A[10];
definitions:
  A[i] := 2 * i;
```

defines an array `A` of size `10` such that its i^{th} component contains the value `2 * i`. The array value represented by expression `A` in this context can be specified by:

```
lambda [10]:[i] := 2 * i
```

without the need of any preliminary declaration nor definition. The definition of `A` can also be rewritten:

```
definitions:
  A := lambda [10]:[i] := 2 * i;
```

The same kind of expression is allowed for functions; `lambda (int):(i) := 2 * i` is equivalent to the function `twice` defined by:

```
declarations:
  (int -> int) twice;
definitions:
  twice(i) := 2 * i;
```

A `lambda` expression can introduce several dimension arrays, multi-parameters functions and mix array and function as one can do with an HLL definition. For instance:

```
lambda (bool)[8,8]:(b)[i,j] := if b
                               then (i + j) % 2 = 0
                               else (i + j) % 2 = 1;
```

This expression is an anonymous function that, maps a boolean value to a chessboard (using a convention that associates boolean values with *black* and *white*); changing the boolean parameter makes the square colors alternate.

A semantically equivalent formulation is given here:

```
lambda (bool):(b) := if b
                     then lambda [8,8]:[i,j] := (i + j) % 2 = 0
                     else lambda [8,8]:[i,j] := (i + j) % 2 = 1
```

This form highlights the fact that the value `b` selects one or the other chessboard definition.

There are below some examples with array expressions corresponding to simple operations on arrays that can be expressed with an anonymous definition, thus without the need to name the constructed arrays:

```
constants:
  int N := 3;
  int M := 7;
  int k1 := 1;
  int k2 := 2;

inputs:
  bool A[N];
  bool B[M];
  bool e;

outputs:
  // creates an array with all the components equal
  lambda [N]:[i] := e;

  // array slice A[k1..k2]
  lambda [k2 - k1 + 1]:[i] := A[i + k1];

  // concatenation of arrays A and B
  lambda [M + N]:[i] := if i < N then A[i] else B[i-N];

  // A in a reverse order
  lambda [N]:[i] := A[N - i - 1];
```

12.9 The *pigeon-hole* example

This example corresponds to the HLL formalisation of the *pigeon hole* problem that we can formulate by: *it is not possible to put N pigeons in (N-1) holes with, at most one pigeon per hole.*

```
constants:
  int NOF_PIGEONS := 10;
  int NOF_HOLES := NOF_PIGEONS - 1;

inputs:
  bool P[NOF_PIGEONS, NOF_HOLES];

definitions:
  /* For each hole there is just one pigeon */
  a := ALL i:[0, NOF_HOLES-1], j:[0, NOF_PIGEONS-1] ALL k:[j+1, NOF_PIGEONS-1]
        (P[j, i] -> ~P[k, i]);

  /* For each pigeon there is at least one hole */
  b := ALL i:[0, NOF_PIGEONS-1] SOME j:[0, NOF_HOLES-1] (P[i,j]);

proof obligations:
  ~(a & b);
```

12.10 The *sudoku* example

We provide here an example that, while addressing a quite popular problem, illustrates the powerfulness of HLL quantification. The goal is to define in HLL the criterion that a *sudoku* grid must be satisfied when entirely filled. Based on this expression, it's easy to use a proof engine for HLL in order to complete a given partially filled grid, provided it can be done at all:

```
inputs:
  int [1,9] grid [9,9];

definitions:
  satisfy :=
    ALL value:[1,9] (
        ALL line:[0,8] SOME col:[0,8] (grid[line, col]=value)
      & ALL col:[0,8] SOME line:[0,8] (grid[line, col]=value)
      & ALL subregion_line:[0,2], subregion_col:[0,2]
          SOME s_line:[0,2], s_col:[0,2]
      (grid[subregion_line * 3 + s_line, subregion_col * 3 + s_col]=value));
```

13 ARITHMETICS IN HLL

HLL-14

Arithmetics in HLL is both *bounded* and *exact*. This is possible thanks to the fact that all the inputs and memories must be explicitly bounded in the model. The definitions contain only a finite number of operations. So any integer value in the model is a finite combination of bounded values, thus it is itself a bounded value.

In this context, all the arithmetic operators must be understood with their mathematical definition. Values are explicitly cast with the cast operator or (less explicitly) cast when used to define a named stream declared with a sized type (provided that the value to store in the stream fits in its declared type, see sections 14.3 and 14.9).

14 STREAM SEMANTICS

HLL is a language for the definition of streams. A stream `s` denotes an infinite sequence of values that we will represent by the following table:

HLL stream expression	sequence of values
`s`	$s_0\ s_1\ s_2\ s_3\ s_4\ \ldots$

Using this notation we specify the semantics of the HLL temporal primitives.

In this section we provide the semantics of the streams defined by a causal (see Section 15) HLL model.

Note: This causality notion is important because it gives a sufficient condition to ensure that at any step of a stream, the value it takes does not depend on itself (s_n is not defined as a solution to a fixpoint equation on the form $s_n = f(s_n)$).

14.1 Input streams

HLL-15

Input streams can be declared either in an `inputs` section or in a `declarations` section. In this second case, a stream is considered as an input if it has no definition at all (neither *combinatorial* nor *initial* nor *next* nor *memory*). If a declared stream is only defined for the *next values*, it's initial value is considered as an initial input.

An input stream represents any sequence of values in its declared type.

14.2 Throw-away definition

The definition `_ := e` is a "throw-away" definition, it can be safely ignored.

14.3 Combinatorial definition

HLL-16

The definition `v := e` means that `v` represents the same sequence as `e`:

`e`	e_0	e_1	e_2	e_3	e_4	$\ldots$
`v`	e_0	e_1	e_2	e_3	e_4	$\ldots$

HLL is declarative and thus has a *substitution principle* that holds at the level of the combinatorial definition. This principle can be expressed as : *if a stream variable* `v` *has a combinatorial definition (*`v := e`*), any occurrence of* `v` *can be substituted with the expression that defines it (*`e`*) without affecting the semantics of the* HLL *model.*

The integer case: HLL requires all the inputs and memories to have a sized type. This is a key point for the arithmetics (see Section 13). However nothing is required for the variables defined by a combinatorial definition. If such a variable is declared with a plain integer, the variable will be large enough to accommodate for the values to be stored. On the other hand if the variable is declared with a sized type (in particular with a sized integer), and it is defined with a value that doesn't fit into that type, the variable will take the exceptional undefined value nil and the substitution principle is broken (see section 14.11).

14.4 Memory definition

HLL-18

The definition `v := e, f` means that `v` takes its first value from stream `e` then from stream `f` shifted one step to the right:

e	e_0	e_1	e_2	e_3	e_4	...
f	f_0	f_1	f_2	f_3	f_4	...
v	e_0	f_0	f_1	f_2	f_3	...

14.5 Initial and next definitions

HLL-19

A memory definition can be split into its two components that are:

1. its *initial* value defined by `I(v) := e` meaning that `v` takes its first value from stream `e`;
2. its *next* value is defined by `X(v) := f` .

The resulting stream is :

e	e_0	e_1	e_2	e_3	e_4	...
f	f_0	f_1	f_2	f_3	f_4	...
v	e_0	f_0	f_1	f_2	f_3	...

14.6 Next definition only

HLL-17

The definition `X(v) := e` means that `v` represents the same sequence as `e` shifted one step to the right. The value of `v` at the first instant is considered as an implicit input (stream `I(v)`), unless `I(v)` is explicitly used in an input section, as specified in 14.1.

e	e_0	e_1	e_2	e_3	e_4	...
I(v)	iv_0	iv_1	iv_2	iv_3	iv_4	...
v	iv_0	e_0	e_1	e_2	e_3	...

14.7 Next expression : `X(e)`

HLL-20

The expression `X(e)` (pronounce *next of e*) represents the same stream as e, shifted one step to the left:

e	e_0	e_1	e_2	e_3	e_4	...
X(e)	e_1	e_2	e_3	e_4	e_5	...

14.8 Unit delay expression : `pre(e)`

HLL-25

The expression `pre(e)` represents the same stream as e, shifted one step to the right:

e	e_0	e_1	e_2	e_3	e_4	...
pre(e)	nil	e_0	e_1	e_2	e_3	...
i	i_0	i_1	i_2	i_3	i_4	...
pre(e, i)	i_0	e_0	e_1	e_2	e_3	...

Where nil represents an undefined exceptional value (see section 14.11).

When the delayed stream takes values in a type that contain integers, a type can be specified in the operator in order to give these integers a size. The syntactic forms of this case are: `pre <T>(e)` or `pre <T>(e, i)` where `T` is a type.

Note that the semantics of `pre` can also be given by its translation in terms of a memory. The stream represented by `pre <T>(e, i)` is the same as the one represented by the memory `m` of type `T` defined by: `m := i, e;`

14.9 Definition of a data memory

HLL-21

Sections 14.4, 14.5, 14.6 and 14.8 describe the principle of the definition of a memory state. Because memories allow definitions of a stream as a function of its previous values, the definition of an integer memory may be diverging in the sense that the values it can take cannot be statically bounded. For this reason the type system requires that any memory (a stream variable defined by a *memory* or a *next* definition) must have a sized type (see Definition 3) and the expression that defines a memory must fit in the declared type of the memory. Below we define the semantics of an integer memory definition of type `T` (where `T` is a constrained integer type).

```
declarations:
  T v;
...
definitions:
  v := e, f;
```

If e_0 fits in type `T` (in the range if it specifies a range or in the specified finite representation otherwise) and $\forall i \in \mathbb{N}$, f_i fits in `T`, the stream `v` is defined by:

`e`	e_0	e_1	e_2	e_3	e_4	$\ldots$
`f`	f_0	f_1	f_2	f_3	f_4	$\ldots$
`v`	e_0	f_0	f_1	f_2	f_3	$\ldots$

When either e_0, or any of the f_i for $i \in \mathbb{N}$ do not fit into the type `T`, the memory takes the exceptional undefined value nil at that depth (see section 14.11).

14.10 Array definitions

HLL-22

Definitions of the form `v[i] := e` follow the same semantics pointwisely by replacing each index `i` by its value taken in the range of legal indices for array `v` given by its declaration.

14.11 Determinism, sanity, and nil values in HLL

HLL-27

In the present section we have seen that a number of HLL constructs may introduce unspecified values in a stream: the so called nil. It is not, in general, a problem to have a stream carrying nil values as long as it is not a stream we are observing (*i.e.* those that appear in `outputs`, `constraints` or

`proof obligations` sections). A nil value in an observed stream leads to different issues, depending on the section it appears in:

- in a `proof obligations` section, the concerned proof obligation cannot be proved because nil is not `true`;
- in an `outputs` section, the HLL model becomes globally non deterministic, it is not even equivalent to itself (the comparison of two instances of nil is also a nil);
- in a `constraints` section, having a possible nil corresponds to an unsatisfiable constraint.

Accepting the non-determinism introduced by the nil in the presence of the uninitialized `pre` in the language would invalidate the substitution principle given in 14.3. This principle holds again if the model is proved to be deterministic in the sense discussed above.

For all these reasons, a semantic tool implementing HLL has to reject non-deterministic models. Different strategies and proof capabilities can be used to reach this goal (rejecting more or less correct models); they are not part of the language specification and must be defined in the tool specifications.

15 CAUSALITY

HLL-23

This section defines the causality in an HLL model and what a *causal* (correct regarding causality) model is. The semantics of streams presented in Section 14 is defined only for causal models. This section considers scalar streams only for the reasons discussed in Section 10. A model containing array definitions should first be expanded and then the question of causality is considered on the scalar definitions as described below.

Restriction 10 (model causality)**.** *A correct* HLL *model shall be causal i.e. all the streams it defines and that contribute to the production of an* output*, a* proof obligation *or a* constraint *shall be causal in the sense defined below.*

15.1 Temporal dependencies between scalar streams

To be well founded, a stream definition must be causal (in other words *non-cyclic*). Here is a first intuitive and informal definition of this notion: *a stream definition is causal if:*

- *each value of the stream is defined by an expression that does not depend (directly or through other streams) on itself and*
- *for inductive definitions if the inductive case (next definition or second member of a memory definition) does not depend directly or indirectly on values that are after in the stream.*

To formalise this relation, we will distinguish the dependencies on the first instant from the other ones; a represents any value of stream a, $I(a)$ its initial value and $X(a)$ any value but the initial one. The dependency relation is defined between terms of the following grammar:

$$\begin{array}{rcl} depterm & ::= & streamexpr \\ & | & I(streamexpr) \\ streamexpr & ::= & identifier \\ & | & \mathsf{op}(streamexpr, \ldots, streamexpr) \\ & | & X(streamexpr) \end{array}$$

where $streamexpr$ represents a stream expression, as defined in the concrete syntax by `<expr>`.

Definition 21 (dependency relation)**.** *The dependency relation denoted* $a :- b$ *(*a depends on b*) is defined by:*

1. $v :- a$ *for a definition* `v := a`*;*
2. $I(v) :- I(a)$ *for an initial definition* `I(v) := a`*;*
3. $X(v) :- a$ *for a next definition* `X(v) := a`*;*
4. $I(v) :- I(b)$ *and* $X(v) :- c$ *for a memory definition* `v := b,c`*;*
5. $\mathtt{pre} < t > (a, b) :- b$*;*
6. $\forall i \in [1..n],\ \mathtt{op}(a_1, \ldots, a_n) :- a_i$*;*
7. transitivity*:* $a :- b \wedge b :- c \ \Rightarrow a :- c$*;*
8. monotony *of* $X()$*:* $a :- b \Rightarrow X(a) :- X(b)$*;*
9. monotony *of* $I()$*:* $a :- b \Rightarrow I(a) :- I(b)$*;*
10. $X(X(a)) :- X(a)$*.*

where op *designates any* n*-ary (*$n > 0$*) combinatorial function and* a_i *are dependency terms (*$depterm$*).*

The rules 1 to 9 define the dependency relation. After application of these 9 rules, a system is said to

be causal if the relation does not contain any pairs of the form:

$$\underbrace{X(X(\ldots(X(a))\ldots))}_{n \text{ next, with } n\geq 1} :- \underbrace{X(X(\ldots(X(a))\ldots))}_{p \text{ next, with } p\geq n}$$

nor

$$I(a) :- I(a)$$

Which means that a system is causal if none of the streams it defines depends, for its next definition, on itself or on its next values.

Rule 10 is added to transform any of these pairs into a cycle in the dependency relation and make the causality criterion easier to implement by reducing it to a cycle search in a graph.

15.2 Composite types, mappings and causality

The causality relation for HLL is defined in 15.1 for a scalar model and thus does not cover the overall language. When a stream is composite, causality is defined component by component which allows to have one array element depending on another element of the same array. Taking this point of view, all the streams are scalar and arrays are *arrays of streams*, tuples are *tuples of streams* etc. . .

16 PREDEFINED COMBINATORIAL OPERATOR SEMANTICS

A combinatorial operator on streams is built from an operator on the values carried by the streams by pointwise application. For instance if (x_n) and (y_n) represent two streams given by their sequence of values, the sum of these streams $(x_n)+(y_n)$ is the stream of the sum (z_n) defined by $\forall n,\ z_n = x_n + y_n$.

This pointwise extension can be defined for any operator op of arity $k \geq 1$ by :

$$\forall p, (op((x_n^1), \ldots, (x_n^k)))_p = op(x_p^1, \ldots, x_p^k)$$

We can write it using the tabular notation we introduced before:

HLL	sequence			
x^1	x_0^1	x_1^1	x_2^1	$\ldots$
x^2	x_0^2	x_1^2	x_2^2	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
x^k	x_0^k	x_1^k	x_2^k	$\ldots$
$op(x^1, \ldots, x^k)$	$op(x_0^1, \ldots, x_0^k)$	$op(x_1^1, \ldots, x_1^k)$	$op(x_2^1, \ldots, x_2^k)$	$\ldots$

Thus to define combinatorial functions on streams from their original operation on values (boolean, integers, structures, arrays), it suffices to define them on values (instead of streams) to capture the whole semantics of the extension to streams.

16.1 Logical operators

These operators apply on boolean values, they are defined below by their truth tables:

conjunction :

a	b	$a\&b$
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

disjunction :

a	b	$a\#b$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

equivalence :

a	b	a<->b
FALSE	FALSE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

exclusive or :

a	b	$a\#!b$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

implication :

a	b	a->b
FALSE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

negation :

a	$\sim a$
FALSE	TRUE
TRUE	FALSE

16.2 Population count

HLL provides various n-ary operators taking a variable number of boolean streams and a static integer value to easily express complex conditions about the number of streams taking the value `true` at a given step. Let's define the combinatorial function $population$ that applies on a finite list of boolean values and returns the number of `true` values among these booleans:

$$population(b_0, b_1, \ldots, b_n) \ = \ \sum_{k=0}^{n} (\text{if } b_k \text{ then } 1 \text{ else } 0)$$

In particular when the list of boolean streams is empty, this function is the constant 0 ($population() = 0$).

With this function, given a static (see the type system in Section 7 for a definition of static) integer value N we define the population count operators by:

$$\texttt{population\_count\_eq}(b_0, b_1, \ldots, b_n, N) \equiv population(b_0, b_1, \ldots, b_n) = N$$
$$\texttt{population\_count\_lt}(b_0, b_1, \ldots, b_n, N) \equiv population(b_0, b_1, \ldots, b_n) < N$$
$$\texttt{population\_count\_gt}(b_0, b_1, \ldots, b_n, N) \equiv population(b_0, b_1, \ldots, b_n) > N$$

16.3 Polymorphic comparison operators =, ==, !=, <>

The polymorphic comparison operators apply on any type (provided that the type has a finite domain (see definition 4) when they share the same structure (same dimensions with same sizes, as specified by the type system).

- both = and == represent the equality operator;
- both != and <> represent the inequality operator.

The following equivalences hold: `a <> b` ≡ `~(a = b)` ≡ `a != b` ≡ `~(a == b)`

The definition of equality on scalars is standard and extends to structured types in the following way: *two structured values are equal if all their corresponding elements are pairwise equal*.

The equality and disequality over finite domain function follows the extension of these operators on structured types: *two functional values are equal if all their corresponding elements are pairwise equal*.

16.4 Shift operators «, »

The shift operators are defined on both signed and unsigned representation of integer values.

If a represents an integer and n a static positive value ($n \geq 0$), then:

- a « n is an n bit shift to the left. From an arithmetical point of view, it corresponds to a multiplication by 2^n. If a is encoded in binary with an N-bit word, a « n requires an $(N + n)$ bits representation.
- a » n is an n bit shift to the right. This operation corresponds to the floor division a /> 2^n If a is encoded in binary with an N bits word, a » n requires $min(N - n, 1)$ bits representation.

The shifts are not defined if the second parameter is a negative value.

16.5 Arithmetic operators +, -, * and unary minus -

Exact implementation of arithmetics, see Section 13 for a discussion about exact bounded arithmetics.

16.6 Integer comparison operators >, >=, <, <=

These operators represent predicates corresponding to the standard order relation on integers. They produce a boolean value `true` when the relation holds and `false` otherwise.

16.7 Maximum $max

Returns the maximum of its two arguments:

$$\$\mathtt{max}(a, b) = \begin{cases} a \text{ if } a \geq b \\ b \text{ otherwise.} \end{cases}$$

16.8 Minimum $min

Returns the minimum of its two arguments:

$$\$\mathtt{min}(a, b) = \begin{cases} b \text{ if } a \geq b \\ a \text{ otherwise.} \end{cases}$$

16.9 Absolute value $abs

This operator takes one integer parameter and produces its absolute value, it is defined by:

$$\$\mathtt{abs}(v) = \begin{cases} v \text{ if } v \geq 0 \\ -v \text{ otherwise.} \end{cases}$$

16.10 Euclidian division /

If a and b are two positive integers, a/b is the result of the Euclidian division and is such that:

$$a = b * (a/b) + r \text{ where } r \text{ is an integer such that } 0 \leq r < b.$$

This operation is not defined when $b = 0$, in which case the result takes the exceptional undefined value nil (see section 14.11).

if a or b is negative, the absolute value of the result is given by the application of the positive (see before) case to the absolute values of a and b and the sign is given by the standard rules:

a	b	a/b
> 0	> 0	> 0
< 0	> 0	< 0
> 0	< 0	< 0
< 0	< 0	> 0

16.11 Remainder $\%$

If a and b are positive integers, then $a\%b$ represents the remainder r of the Euclidian division (see Section 16.10). This operation is not defined when $b = 0$, in which case the result takes the exceptional undefined value nil (see section 14.11)

if a or b is negative, the absolute value of the result is given by the application of the positive (see before) case to the absolute values of a and b and the sign is the sign of a.

16.12 Floor division />

This operator implements the *floor* of the exact division; i.e. `a/>b` represents the biggest integer smaller than or equal to the rational a/b.

This operation is not defined when $b = 0$, in which case the result takes the exceptional undefined value nil (see section 14.11).

It can be expressed using the division operator by:

$$a/>b = \begin{cases} a/b \text{ if } sign(a) = sign(b) \text{ or } a\%b = 0 \\ (a/b) - 1 \text{ otherwise.} \end{cases}$$

where $sign(.)$ is the sign function on integers defined by:

$$sign(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ -1 \text{ otherwise.} \end{cases}$$

16.13 Ceiling division /<

This operator implements the *ceiling* of the exact division; i.e. `a/<b` represents the smallest integer bigger than or equal to the rational a/b. More formally, if a and b are two integers, $a/>b$ is such that:

$$a = b * (a/<b) + r \text{ where } r \text{ is an integer such that } -|b| < r \leq 0.$$

$|b|$ represents the absolute value of b. This operation is not defined when $b = 0$, in which case the result takes the exceptional undefined value nil (see section 14.11).

It can be expressed using the division operator by:

$$a/<b = \begin{cases} (a/b) + 1 \text{ if } sign(a) = sign(b) \text{ and } a\%b \neq 0 \\ a/b \text{ otherwise.} \end{cases}$$

16.14 Bitwise logical operators: `$not`, `$and`, `$or`, `$xor`

All the bitwise operators are defined on signed integers, meaning that applying them on an unsigned value introduces an implicit conversion from unsigned to signed.

At a bit representation level, a signed value can be seen as an infinite boolean word:

$\underbrace{\ldots s\, s \ldots s}_{\infty} b_n b_{n-1} \ldots b_0$ where s is the sign bit

The bitwise operations are the pointwise extension of the logical operators on these infinite words.

16.15 Power (^)

If a and b are two integers then $a\hat{}b$ is equal to:

- $\underbrace{a * a * \cdots * a}_{b \text{ times}}$ if $b > 0$;
- $1/(\underbrace{a * a * \cdots * a}_{|b| \text{ times}})$ if $b < 0$ and $a \neq 0$;
- 1 if b is equal to zero (in particular we take the convention that $0^0 = 1$);
- undefined when $b < 0$ and $a = 0$, in which case the result takes the exceptional undefined value nil (see section 14.11).

16.16 Cast

The cast (`cast<t>(e)`) allows to interpret an integer expression (here `e`) as a value of a specified implementation type (`t`) by considering its binary representation and the binary implementation of `t`. The binary representation is based on two's complement for signed values and standard binary for unsigned ones.

The following table specifies the result of this cast depending on the representation of `e` and `t`.

Let $b_{n_1} \ldots b_2 b_1$ be a binary representation of the value taken by `e`.

representation of `e`	representation of `t`	size condition	cast expression value in binary
`int unsigned n1`	`int unsigned n2` or `int signed n2`	when $n_1 \leq n_2$	fill with zeros $\underbrace{0 \dots 0 b_{n_1} \dots b_2 b_1}_{n_2 \text{ bits}}$
`int signed n1`	`int unsigned n2` or `signed int n2`	when $n_1 \leq n_2$	fill with the MSB $\underbrace{b_{n_1} \dots b_{n_1} b_{n_1-1} \dots b_2 b_1}_{n_2 \text{ bits}}$
`int unsigned n1` or `int signed n1`	`int unsigned n2` or `int signed n2`	when $n_1 > n_2$	ignore extra bits $\underbrace{b_{n_2} \dots b_2 b_1}_{n_2 \text{ bits}}$

Then this binary value is interpreted following a representation of `t` (`unsigned`: positive integer value represented in base 2; `signed`: two's complement).

16.17 `bin2u`

If w is an array of boolean values and n a constant expression, `bin2u`(w, n) is the integer whose unsigned binary representation is given by the first n bits ($w[n-1] \dots w[0]$) of w (where $w[0]$ is the *Least Significant Bit*). n must be statically less than or equal to the size of w.

16.18 `bin2s`

If w is an array of boolean values and n a constant expression, `bin2s`(w, n) is the integer which signed binary representation is given by the first n bits ($w[n-1] \dots w[0]$) of w (where $w[0]$ is the *Least Significant Bit*). n must be statically less than or equal to the size of w.

16.19 `u2bin`

If v is a positive integer value and n a constant expression, `u2bin`(v, n) is the boolean array containing the bit values of the n bits unsigned binary representation of the integer value v. If the representation does not fit within n bits, the array will contain the n first bits of this representation. The resulting array is such that the item at index 0 contains the *Least Significant Bit* of this representation.

For the case where v is negative. It is defined by:

$$\texttt{u2bin}(v, n) = \texttt{u2bin}(\texttt{cast} < \texttt{int unsigned n} > (v), n)$$

16.20 `s2bin`

If v is an integer value and n a constant expression, `s2bin`(v, n) is the boolean array containing the bit values of the n bits signed binary representation of the integer value v. If the representation does not fit within n bits, the array will contain the n first bits of this representation. The resulting array is such

that the item at index 0 contains the *Least Significant Bit* of this representation and the item at index $n-1$ contains the sign bit (provided than n is large enough).

Note:

the primitives `s2bin` and `u2bin` give the same boolean array for any given integer stream. HLL provides two primitives for convenience only.

16.21 If-then-else

```
a := if c then e1 else e2;
```

Selects the value of the expression present in the `then` branch (`e1`) or in the `else` branch (`e2`, depending on the boolean value taken by `c`.

HLL offers a shortcut for a cascade of `if-then-else`:

```
a := if c1
     then e1
     elif c2
     then e2
     elif c3
    ...
     then en
     else e
```

This syntactic form is equivalent to :

```
a := if c1
     then e1
     else if c2
          then e2
          else if c3
             ...
                       then en
                       else e
```

16.22 Array projection

The array projection `A[x]` returns the value contained in the array at position `x` if `x` is within the declared bounds of `A`. A model such that the definition of one of its *outputs*, *proof obligations* or *constraints* requires the access out of an array bounds is incorrect.

A projection involves a structured stream and an index, as said in 10 and in 15.1, the stream semantics is given on scalar. The definition of an array projection can be explained on scalars using the following expression equivalent to stream `A[x]`:

```
(x
| 0     => A[0]
| 1     => A[1]
| 2     => A[2]
...
| N - 1 => A[N - 1])
```

where `N` is the size of array `A` and `A[0]`, `A[1]`, ... are streams that can be projected statically. Based on this equivalent form, it is possible to transform an HLL model with arrays and projection into a model involving scalars only.

16.23 Function application

The function application `f(x)` returns the value of function `f` at point `x`. if `x` is in the declared domain of `f`. A model such that the definition of one of its *outputs*, *proof obligations* or *constraints* requires the value of a function out of its declared domain is incorrect.

As for array projection, the function application `f(x)`, if `f` has type `bool -> int` is equivalent to:

```
if x then f(true) else f(false)
```

In the case the domain of `f` is not finite, `int -> int` for instance, it is possible to build a finite expression of this kind, based on the fact inputs and memories can take a finite set of values, the domain of a function application at its application point can always be restricted to the possible values its argument can take. If a model has bounded inputs and memories and contains function, it is always possible to translate it into a finite model without functions.

16.24 (... with ... := ...)

Example:

```
b := (a with .m[1].5 := e);
```

`b` is componentwisely equal to `a` except for the component specified by the path `.m[1].5` in the structure that is equal to `e`; the following invariant holds: `b.m[1].5` = `e`.

16.25 Elementhood : `a:D`

The operator `a : D` is a predefined predicate that, given a *stream expression* `a` and a *domain* produces `true` when the expression takes a value that is an element of the specified domain.

A domain can be either:

- a *sort*, in this case the predicate expresses the elementhood of the value to the set of the possible values for the specified sorts (those defined for this sort and all its subsorts);
- a *range e.g.* `a : [-8, 7]` (equivalent to the expression `a >= -8 & a <= 7`).

– a finite scalar type *e.g.* `a : int signed 4` (equivalent to the expression `a >= -8 & a <= 7`), or `c:bool` (which is always true).

Remark: even-though the grammar allows for the elementhood construct `a:$item(b)`, this is not a valid expression.

APPENDIX A: LIST OF REQUIREMENTS

Here is the list of requirements attached to the present document. A tool that intends to implement the HLL language shall cover these requirements.

HLL-1
HLL-2
HLL-3
HLL-4
HLL-5
HLL-6
HLL-30
HLL-7
HLL-8
HLL-10
HLL-32
HLL-11
HLL-12
HLL-32
HLL-28
HLL-29
HLL-31
HLL-13
HLL-26
HLL-14
HLL-15
HLL-16
HLL-18
HLL-19
HLL-17
HLL-20
HLL-25
HLL-21
HLL-22
HLL-27
HLL-23

APPENDIX B: LIST OF RESERVED KEYWORDS

Below is a list of reserved HLL words that cannot be used as identifiers.

```
ALL
bin2s
bin2u
Blocks
blocks
bool
cast
```

```
CONJ
constants
Constants
constraints
Constraints
declarations
Declarations
definitions
Definitions
DISJ
elif
else
enum
false
False
FALSE
I
if
inputs
Inputs
int
lambda
namespaces
Namespaces
obligations
Obligations
outputs
Outputs
population_count_eq
population_count_gt
population_count_lt
pre
PRE
PROD
proof
Proof
s2bin
SELECT
signed
SOME
sort
struct
SUM
then
true
True
TRUE
tuple
types
Types
```

```
u2bin
unsigned
with
X
```

END OF DOCUMENT