L3

Contents

1 Language Features

This document gives a overview of the L3 specification language. The language is imperative, first-order, case-sensitive and strongly typed.

1.1 Commenting

Block comments are delimited by '`{-`' and '`-}`'. Inline comments start with '`--`' and extend to the end of the current line. All commented code is simply ignored. Block comments can be nested, which means that it is possible to comment out code that already contains comments.

1.2 Primitive Types

The primitive types of the language are:

```
unit  bool  char  nat  int  bits(·)  rounding .
```

Types can be combined into tuples (with the '`*`' operator) and maps (with the '`->`' operator). Finite set, option and list types can be formed with the '`set`', '`option`' and '`list`' postfixes. An *equality type* is any type that excludes map types. The equality comparison operation '`==`' can only be used to compare members of equality types. Furthermore, sets can only be constructed over equality types.

The infix '`::`' is used for type annotation. For example, '`1::bits(16)`' denotes a 16-bit word with value one. For convenience, this bit-vector annotation can be shortened to '`1`16`'.

Synonyms The '`type`' construct can be used to introduce new *type synonyms*; for example, the following are valid type declarations:

```
type word = bits(32)
type mem = bits(32) -> bits(8)
type foo = word set * mem * bool
```

Type synonyms can be used interchangeably with their defining type. In the context of the type declarations above, we can introduce a new constant '`four`' as follows:

```
word four = 4::bits(32)
```

The extra type annotation is unnecessary here but it helps demonstrate that '`bits(32)`' and the synonym '`word`' are effectively one and the same.

1.3 Literals

The '`unit`' type has a single element '`()`'. The '`bool`' type has two elements: '`true`' and '`false`'. The string type '`string`' is a pseudonym for '`char list`'. String literals are represented using double quotes and character literals are preceded by a hash; for example, '`"hello"`' is a string and '`#"a"`' is a character. The type '`rounding`' has four elements: '`roundTiesToEven`', '`roundTowardPositive`', '`roundTowardNegative`' and '`roundTowardZero`'.

The language supports three numeric types: '`nat`', '`int`' and '`bits(·)`'. Number literals can be expressed using binary, decimal or hexadecimal notation; as indicated by the prefixes '`0b`', '`0d`' and '`0x`' respectively.[1] An additional prefix character ('`n`', '`i`' or '`w`') can be used to avoid explicit type annotation; for example, '`0i15`' and '`0ixF`' both denote the integer value fifteen. A number without any prefixes is treated as a decimal bit-vector, wherein leading zeroes are ignored. Bit-vectors can be represented using a special quotation syntax; for example, '`'0101'`' is equivalent to '`0b101`4`'.

Bit-string literals L3 allows lists of Booleans (bit-strings) to be treated as numeric values, with the head of the list representing the most significant bit. No constraint is placed on the length of bit-strings.[2] This makes bit-strings distinct from bit-vectors, which are fixed width (as constrained by the type). The prefix character '`s`' may be used to identify bit-string literals. Leading zeroes are significant when bit-string values are presented in binary or hexadecimal formats; for example, the expressions '`0b00001101::bool list`' and '`0sx0D`' both represent the list

```
list {false, false, false, false, true, true, false, true}
```

.

Consequently, the bit-string values '`0sxD`' and '`0sx0D`' are not equal. Note that leading zeroes are ignored in *decimal* bit-string expressions; for example, '`029::bool list`', '`0sd29`' and '`0s11101`' all represent the same bit-string. When the prefix '`s`' is used on its own, the expression is parsed as a binary value (this means that '`0s20`' will fail to parse).

[1] Octal is not supported.

[2] The bit-string type represents a countably infinite set of values.

1.4 Global State

Global variables are introduced using the '`declare`' construct, which adds elements to the global *state-space*. The following represent valid variable declarations:

```
declare number :: nat
declare { integer :: int, boolean :: bool }  -- two declarations
declare { pc :: bits(32)  r1 :: bits(64) }   -- commas are optional
```

At the point of declaration, it is not possible to specify initial values. Users can specify (and later call) their own initialisation procedures if required.

The '`<-`' operator is used to assign values to *mutable* variables; for example

```
pc <- 0xF -- an assigment
```

assigns value '`0xF`' to our declared global variable '`pc`'. The bit-string and bit-vector types also permit assignment to *bit-ranges*; for example

```
{ pc<3:0> <- 0xF; pc<31> <- true } -- a sequence of assigments
```

will set the bottom four bits and then the most-significant bit of '`pc`'.

1.5 Tuples

Tuples can be formed using the pairing constructor '`,`', which associates to the right. Bracketing is optional (but recommended); for example, all of the following expressions are semantically identical:

```
"one", true, false
"one", (true, false)
("one", true, false) -- preferred form
("one", (true, false))
```

These expressions have type '`string * bool * bool`'.[3] Note that the expression

```
(("one", true), false)
```

has type '`(string * bool) * bool`', which makes it strictly type distinct from (incompatible with) the previous expression.

Note that (in expressions) commas are the loosest binding operator, which means that

```
x, y == 3, 4
```

will parse as

```
(x, ((y == 3), 4)) -- fully elaborated form
```

and not as

```
(x, y) == (3, 4) -- this could be what was intended
```

It is advisable that users only drop brackets when there is little chance of ambiguity.

[3] This is the same as '`string * (bool * bool)`'.

1.6 Maps

Maps are used to represent total functions.[4] For example, the declaration

```
type map :: nat -> int   -- a map type pseudonym
```

introduces a type pseudonym for maps with domain '`nat`' and codomain '`int`'. Maps and single map entries can be assigned to; for example, given the declaration

```
declare -- three global maps
{
   m1 :: map
   m2 :: map
   m3 :: nat -> nat
}
```

the statement

```
{ m1 <- m2; m1(1) <- 2 }
```

will set map '`m1`' to be '`m2`' and this map is then updated so that '`m1(1)`' takes value '`2`'.

The language encourages the use of simple, *un-curried* maps, which means that the type

```
type map :: bool * nat -> int
```

should be used in preference to

```
type nmap :: nat -> int
type map :: bool -> nmap
```

In particular, the language does not permit the syntax '`m(true)(2)`' whereas '`m(true, 2)`' is fine.

The language does not provide a mechanism for specifying *anonymous functions* or *anonymous maps*.[5] Furthermore, maps and operations (either primitive or user defined) are considered to be distinct. As such, it is not possible to make assignments of the form '`m3 <- Log2`'.

1.7 Sum Types

Enumerated and sum (tagged union) types can be introduced using the '`construct`' keyword. For example, the declaration:

```
construct enumerated { Zero One Two Three } -- a new type called "enumerated"
```

introduces a new type '`enumerated`' that consists of four elements (new constants). Sum types can be declared in a similar manner; for example:

```
construct bool_option
   { BoolNone, BoolSome :: bool } -- an unnecessary variant of "bool option"
```

The elements of this new type are: '`BoolNone`', '`BoolSome(false)`' and '`BoolSome(true)`'.[6] Every *constructor* of the sum type must map from a pre-existing type (or be nullary) and this precludes the definition of new recursive or mutually recursive datatypes.

Enumerated types (where every constructor is nullary) admit casting to/from numeric types.[7] For example, '`[One]::nat`' is equal to '`0n1`' and '`[0n1]::enumerated`' is equal to '`One`'.

[4] Maps must have an equality type as their *domain*. They can be viewed as a form of array and are typically used to model memories and register banks.

[5] The function '`fn x => x + 1`' is an example of an anonymous function in Standard ML.

[6] The brackets are needed here; for example, '`BoolSome true`' will not parse.

[7] The same does not hold for general sum types.

1.8 Records

Record types can be introduced using the 'record' construct. The following is an example of a record declaration:

```
record rec
{
  number :: nat
  integer :: int
  boolean :: bool
}
```

Every value of type 'rec' has three components, which may be accessed using *dot* notation; for example, if we have a global variable

```
declare rec :: rec
```

then the sub-elements are 'rec.number', 'rec.integer' and 'rec.boolean'. The language does not provide a syntax for record *literals*. As such, records must be built up incrementally; for example, the following is a sequence of assignments to components of the variable 'rec':

```
{ rec.number <- 1; rec.integer <- 2; rec.boolean <- true }
```

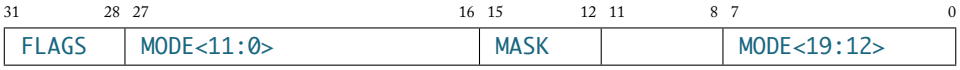
It is not possible to express this with a single assignment to 'rec'.

1.9 Registers

Register types can be introduced using the 'register' construct; for example:

```
register sreg :: bits(32)
{
  31-28:      FLAGS
  7-0, 27-16: MODE
  15-12:      MASK
}
```

This declares a new type 'sreg' that represents the following 32-bit register:



Note that the bit-field 11..8 of 'sreg' does not correspond with a named component. Also note that fields are allowed to be non-contiguous, which is the case for the 'MODE' field. Register types are essentially special forms of record types, with the automatic specification of bijections to a bit-vector representation. Each named bit-field can be accessed using the *dot* notation; for example, if we have

```
declare status :: sreg
```

then the fields 'status.FLAGS' and 'status.MASK' have type 'bits(4)' and 'status.MODE' has type 'bits(20)'. Unlike records, register types automatic admit casting to/from an underlying bit-vector type. The expression '&status' gives the value of the register (with type 'bits(32)') and 'sreg(0xFFFF0000)' constructs a register of type 'sreg'. It is possible to use this notation to update arbitrary bit-fields of a register; for example, the statement:

```
&status<20:7> <- '11101101100111'
```

will update the bit-field 20..7 of 'status' using the 14-bit literal value.

1.10 Polymorphic Types

Users cannot declare their own polymorphic types. Furthermore, all user defined operations and variables must be monomorphic. The language does, however, provide support for three built-in polymorphic datatypes, which may be used under the proviso that these types are always fully type instantiated (free of type variables).

1.10.1 Lists

The language provides limited support for working with lists. Lists can be specified as follows:

```
list { 0n0, 2, 4 }                  -- a list of naturals (length three)
0n0 @ 2 @ 3 @ Nil                   -- expanded version
Cons (0n0, Cons (2, Cons (4, Nil))) -- equivalent version
```

These expressions have type '`nat list`'. Given the first-order nature of L3, only simple list operations are provided (for example: '`Head`', '`Tail`' and '`Length`'). Higher-order list operations, such as *filter*, *map* and *fold*, are not available.

1.10.2 Finite Sets

The language provides basic support for finite sets. Sets can be specified as follows:

```
set { 0n0, 2, 4 }                   -- a three element set (of naturals)
0n0 insert 2 insert 4 insert set {} -- an expanded version
SetOfList (list { 0n0, 2, 4 })      -- a version built from a list
```

These expressions have type '`nat set`'. The infix operators '`in`' and '`notin`' test for set membership; for example, the expressions

```
0n0 in set { 0, 2, 4 }     -- test for membership
0n1 notin set { 0, 2, 4 }  -- test for non-membership
```

both evaluate to '`true`'. Sets can only be defined over equality types. It is not possible, for example, to construct the type '`(bool -> bool) set`'.

1.10.3 Option Types

The option type is a polymorphic sum type that has two constructors: '`None::`α `option`' and '`Some::`$\alpha \to \alpha$ `option`'. Option types are helpful when specifying partial operations.

1.11 Statements and Expressions

This section presents the statements and expressions of the language. Statements can be grouped into blocks using the syntax

```
{
  statement1; -- do this
  statement2; -- then do this
  ...
}
```

Each block, and sub-statement, has type '`unit`'. The operator '`;`' is strictly binary: trailing semi-colons not permitted at the end of a sequence of statements. Expressions are distinct from statements, in particular expressions preclude assignments.

1.11.1 Nothing

The statement '`nothing`' has no effect. Its primary purpose is to facilitate the specification of control flows wherein some paths do something (update the state) and others do nothing.[8]

1.11.2 Conditionals

The language supports *if-then-else* statements and expressions, and *when-do* statements. The latter is provided as a convenient shorthand; that is to say, the statement

```
when p do x
```

is equivalent to the statement

```
if p then x else nothing
```

Note that the language does not support layout syntax (the off-side-rule).

1.11.3 For Loops

The language provides support for simple *for-loop* statements. On each iteration of a for-loop, a natural number index variable is either incremented or decremented (by one). This process is repeated until the index value reaches a pre-computed upper or lower bound; for example, the loop statements

```
for i in 1 .. 3 do proc(i)
for i in 3 .. 1 do proc(i)
```

are respectively equivalent to

```
{ proc(1); proc(2); proc(3) }
{ proc(3); proc(2); proc(1) }
```

The initial and final loop index values may be arbitrary natural number expressions. As such, it may be the case that the loop direction will only be known once these bound expressions are computed.

In addition to regular *for* loops, the language also supports *foreach* loops, which operate over lists. For example, the following code makes three calls to '`proc`':

```
foreach i in list {0n2, 4, 6} do proc(i)
```

Note that this construct will also work for bit-strings and for normal strings.

There is no support for *do while* or *repeat until* loops.

[8] The '`nothing`' statement is essentially syntactic sugar for '`()`'.

1.11.4 Exceptions, Unknown and Undefined

The keyword '`UNKNOWN`' is used to represent an *unspecified* element, which may be of any given type. The language does not stipulate whether or not this value should be chosen deterministically. If the value were chosen deterministically then the expression '`UNKNOWN == UNKNOWN`' will always be '`true`'. However, if the choice is non-deterministic then there is no way of knowing if an occurrence of this expression is '`true`' or '`false`'.

The language supports user defined exceptions, which may be raised but not caught. The '`exception`' keyword is used to declare new exception categories; for example

```
exception ASSERT :: string
```

introduces a new exception '`ASSERT`' that takes a single string as an argument. This exception can be raised in expressions and statements as follows:

```
#ASSERT ("an error has occurred")
```

Thus, exceptions can be used to flag the occurrence of a variety of different errors. The L3 interpreter will stop when an exception is raised. Exceptions can be raised within expressions and are not constrained by return type.

The behaviour of various operations is *undefined* on some inputs; for example the arithmetic expression 'x `div 0n0`' is *undefined*. For any given undefined expression, the language permits three possible interpretations (implementations):

- A deterministic value can be chosen. For example: 'x `div 0n0`' is equal to '`0n0`' is a valid interpretation of this undefined expression.
- A non-deterministic value can be chosen. In particular, is not necessary for the expression 'x `div 0n0 ==` x `div 0n0`' to be '`true`'.
- An exception can be raised.

It is expected that specifications will be designed with some preferred treatment of undefined behaviour in mind. In this example one possible way to manage undefined cases is by taking care to explicitly write code of the form '`if n == 0 then ... else .. x div n ..` '. This will ensure that undefined cases are *unreachable*.

1.11.5 Procedure Calls

A *procedure* is any user defined operation with return type '`unit`'. Procedures typically provide functionality by means of updating the global state (through side effects). Procedures can be called (and exceptions can be raised) in statement blocks; for example:

```
{
  ...
  proc (1, true); -- this is a procedure call
  ...
}
```

The following code is valid but not recommended:

```
{
  ...
  #ASSERT "error"; -- raise an exception
  proc (1, true)   -- the L3 interpreter will never reach this code
}
```

In the above, the exception is unconditionally raised, so the remaining code should be regarded as unreachable (redundant).

1.11.6 Local Variables

The L3 language supports two varieties of variables: *mutable* and *immutable*. All global variables are mutable. The keyword '`var`' is used to declare new mutable variables with local scope. Local variables can be introduced anywhere within a statement block and these variables can be assigned to using the '`<-`' operator. Immutable variables are introduced within a block using the syntax '*x* `=` *expr*`;` ...' and this is treated as a *let*-statement. An immutable value cannot be updated, however it can fall out of scope by declaring a new immutable variable with the same name. The operators '`<-`' and '`=`' are not permitted in expressions.

The specification below uses two mutable variables '`x`' and '`y`':

```
{
  var y = z;
  var x = y;
  if p then { x<3:0> <- '1111'; y <- x + 1 }
       else { x<7:4> <- '1111'; y <- 0 };
  return (x + y)
}
```

This could be respecified using immutable variables, for example

```
{
  y = z;
  x = y;
  x, y = if p then (x || '00001111', x || '00001111' + 1)
              else (x || '11110000', 0);
  return (x + y)
}
```

It is not always easy to eliminate mutable variables. In this case the second specification is more cumbersome because the bit-field update has been replace by an explicit mask (bitwise-or) operation. This operations has been repeated in specify the value of '`y`' because assignments are not permitted in expressions.

Mutable variables can be declared without an initial assignment; for example, '`var x`' is equivalent to '`var x = UNKNOWN`'.

1.11.7 Pattern Matching

The language supports *match* statements and expressions. Pattern matching is supported with respect to: literals (see Section 1.3), constructors,[9] (immutable) variables, wildcards (anonymous variables) and bit-patterns.

[9] This includes matching on tuples, as well as '`None`', '`Some`', '`Cons`' and '`Nil`'. Matching on finite sets and record fields is not possible.

Given the type declarations:

```
construct enum { One Two Three }
construct data
{
  NoData
  Data1 :: string * bits(8)
  Data2 :: enum * bits(16)
}
```

the following illustrates a match expression:

```
match y
{
  case -1, NoData or -2, NoData      => 0`4   -- note use of "or"
  case _, Data1 ("hello", '11 x 11') => x + 1 -- use of bit-pattern
  case i, Data2 (Three, '1111 y')    => [i] + y<3:0>
  case 1, Data2 (_, 8)               => 4
  case _                             => 8
}
```

The body (right-hand side of '`=>`') of the first case line with a *matching* pattern is selected. The symbol '`_`' is a wildcard and will match any value. The '`or`' construct provides a shorthand for writing blocks of clauses with differing patterns but the same body.

The terms '`'11 x 11'`' and '`'1111 y'`' are called *bit-patterns* and they will match against a set of bit-vector values. From the type declaration of '`data`', it is known that the variable '`x`' must represent a 4-bit value because it occurs in the middle of an 8-bit word (the match requires bits '`11`' as a prefix and as a suffix). Likewise, '`y`' must be a 12-bit value. Multiple variables can occur in a bit-patterns, for example, '`'1 x 1 y 0'`' is a valid bit-pattern. The type checker will often be able to infer the type of pattern variables and if this fails type annotations can be added.

The '`pattern`' construct can help avoid situations where explicit annotations are repeatedly required (causing clutter in patterns). If, for example, the variable '`imm5`' is frequently used to match a 5-bit value then the following declaration can be made:

```
pattern imm5 :: bits(5)
```

In subsequent code the default type for pattern variable '`imm5`' will be '`bits(5)`'. The '`pattern`' construct only influences type checking. If a particular type hint is no longer appropriate then it can be dropped; for example, one can insert the declaration '`clear pattern imm5`'.

1.12 Defining Constants and Operations

Users can construct operational semantics style formal specifications in L3 by defining their own constants and operations. There is no module system or facility for local definitions; all user definitions are made at the *top-level* with global scope. User constants and operations cannot be overloaded or redefined.

A *pure constant* is a nullary (zero argument) operation that is state independent; for example, the following code declares a new constant:

```
bits(32) UINT_MAX = 0xFFFFFFFF
```

The following code declares operations that are nullary but state dependent:

```
declare number :: Nat
nat numberPlus1 = return (number + 1)
unit incNumber = number <- numberPlus1
```

Users can also declare new unary operations, which may or may not be state dependent; for example:

```
nat numberMinusN (n::nat) = return (number + n)  -- state dependent
unit decNumber () = number <- numberMinusN (1)
nat mla (m::nat,n::nat,a::nat) = return (m * n + a)  -- pure
```

The operation '`mla`' is unary because it takes a single triple as an argument. To enhance clarity and avoid misunderstandards, users may elect to specify nullary state dependent operations using a single '`unit`' argument. Given the declarations above, the expression '`decNumber()`' gives the clear impression that a call is being made (with possible side-effects), wheres the expression '`incNumber`' does not.

At the point of declaration, operation arguments must be fully type annotated and only monomorphic types are allowed.[10] It is, nevertheless, possible to declare bit-vector operations that are valid with respect to some given set of bit widths, for example, the following are valid declarations:

```
--- valid for any bit-vector width N ---
bits(N) bv_mla (m::bits(N), n::bits(N), a::bits(N)) = return (m * n + a)

--- valid for sufficiently wide bit-vectors (N greater than eight) ---
bool bit8 (x::bits(N)) with N > 8 = return (x<8>)
```

An L3 type error will arise if ever an application of the operation '`bit8`' violates the bit width constraint. Note that vector width variables (such as '`N::nat`' above) can be used as if they were a regular variables in the definitional body of an operation.

1.13 Components

When specifying instruction set architectures, the two most significant programmer's model components are typically the *main memory* and a collection of *general purpose registers*. In the MIPS architecture, for example, there are thirty-two general purpose registers, which may be represented using the following map:

```
declare gpr :: bits(5) -> bits(64)
```

The statement

```
gpr(d) <- gpr(r)
```

[10] An compile time error will arise if the type checker is not able to infer all types: the final definition must be free of type variables.

then expresses moving a value from register 'r' to register 'd'. However, register a memory access is frequently more complicated than this; in particular, registers can be banked and memory can be complex.[11] As such, it is often necessary to specify collections of operations for accessing (reading and writing) these components. With the MIPS example, register zero is special (it is a constant value) and so one could define the following operations:

```
bits(64) read_gpr (n::bits(5)) = if n == 0 then 0 else gpr (n)
unit write_gpr (v::bits(64), n::bits(5)) = when n <> 0 do gpr (n) <- v
```

Thus, the previous register transfers would now be specified as follows:

```
write_gpr (read_gpr (n), d)
```

This precise style of specification is not entirely satisfying though, since it departs in appearance from the pseudo code found in MIPS reference manuals. The L3 language provides a special mechanism for specifying a pair of operations that read and write state components. With the MIPS example, the following declaration can be made:

```
component GPR (n::bits(5)) :: bits(64)
{
   value = if n == 0 then 0 else gpr(n)
   assign value = when n <> 0 do gpr(n) <- value
}
```

One can then write:

```
GPR(d) <- GPR(r)
```

which is closer to the syntax used in the MIPS reference manuals. The 'component' construct effectively provides a controlled form of operator overloading, whereby the 'assign' operation[12] is select when the operator name appears on the left-hand side of an assignment ('<-') and the 'value' operation[13] is applied otherwise. Semantically, these two operations are exactly the same as the manually defined versions 'write_gpr' and 'read_gpr'. The "write" operation will always have a 'unit' return type. Users are free to specify the "read" and "write" operations in any manner that they deem appropriate; however, the "write" operation cannot be defined recursively, nor can the "read" operation call the "write" operation.

1.14 Recursion

The language permits the definition of recursive operations. It is not possible to define mutually recursive operations.

The L3 language has been designed to act as an authoring front-end for the HOL4 theorem prover; facilitating the task of writing instruction set specifications. In order to successfully import an L3 specification, it is necessary for the HOL4 system to prove that every user defined

[11] ARM registers are banked according to processor mode. Memory can often be accessed with big- and little-endian byte ordering. Special treatment may be needed for unaligned memory addresses, for example, when reading a 16-bit value from an odd address. There may also be address translation (virtualisation) and memory hierarchies (caching).

[12] In this example the "write" operations is internally called 'write'GPR'.

[13] Internally the "read" operation is simply called 'GPR'.

operation is total (terminates on all inputs). If an L3 function is non-terminating then it will be impossible to export that definition to HOL4. If a function is terminating but the standard automation of HOL4 is unable to construct a termination proof, then it may be necessary to provide the prover with a hint in the form of a *measure*.

Consider the following specification:

```
declare m :: bits(8) -> bits(8)

unit load (a::bits(8), l::bits(8) list) =
    match l
    {
       case Nil => nothing
       case Cons (h, t) => { m(a) <- h; load (a + 1, t) }
    }
```

The '`load`' operation recurses over the list '`l`', loading values into the map '`m`'. At the time of writing this documentation, trying to build the exported HOL4 version of this specification gives rise to the following error message:

```
Initial goal:

∃R.
  WF R ∧
  ∀state a l h t.
    (l = h::t) ⇒
    R ((a + 1w,t),state with m := (a =+ h) state.m) ((a,l),state)

Exception raised at TotalDefn.Define:
between beginning of frag 0 and end of frag 0:
at TotalDefn.defnDefine:

Unable to prove termination!

Try using "TotalDefn.tDefine <name> <quotation> <tac>".
```

This occurs because HOL4 is unable to automatically prove that '`load`' always terminates. To get around this problem, the original L3 specification can be modified to contain an annotation as follows:

```
unit load (a::bits(8), l::bits(8) list) measure Length(l) = ...
```

The metadata '`measure Length(l)`' tells HOL4 to examine the length of the list '`l`', since this value will decrease upon each recursive call to '`load`'. When the length of the list is zero (the '`Nil`' case) there are no recursive calls, so termination is guaranteed.

The expression occurring after the '`measure`' keyword must be of type '`nat`'. After providing a measure, it is still possible that HOL4 will fail to prove termination, either because the supplied measure is incorrect or because the termination proof is non-trivial (requires human guidance and/or custom lemmas). If this occurs then users should either simplify/rewrite their specifications or, if necessary, report the problem.

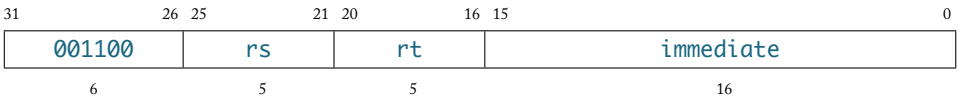
1.15 Instruction Set Definitions

When working with instruction set models, it is helpful to construct a datatype that represents the set of machine (low-level) instructions. This datatype constitutes a simple form of *abstract syntax tree* (AST) and it can be used to build: instruction *decoders* (identifying machine-code instructions and splitting opcodes into sub-fields); instruction *encoders* (generating opcodes for a given instruction); *next-state functions* (applying a semantics function for a given instruction) and *assembly code parsers*.

By way of an example, consider the MIPS instruction `ANDI`: this instruction has the assembly code syntax

```
ANDI rt, rs, immediate
```

and the following 32-bit machine-code encoding:



The semantics of this instruction can be expressed with the operation:

```
unit run_ANDI (rs::bits(5), rt::bits(5), imm::bits(16)) =
   GPR(rt) <- GPR(rs) && ZeroExtend (imm)
```

One could use this operation to directly specify a next-state function, which would be of the form:

```
unit Next =
   match Fetch -- fetches a 32-bit opcode from memory
   {
      ...
      case '001100 rs rt imm' => run_ANDI (rs, rt, imm)
      ...
   }
```

This style of specification is direct and concise; however, it does lack some of the flexibility that comes with using an instruction datatype. The L3 language provides support for an slightly different style of specification. This alternative approach is explained by first examining how one would specify things manually (explicitly) and then showing how L3 can streamline this process.

Firstly, one could explicitly define an AST datatype for MIPS instructions; it would be of the form:

```
construct instruction
{
   ...
   ANDI :: bits(5) * bits(5) * bits(16)
   ...
}
```

This datatype provides a suitable constructor for the ANDI instruction. It is now possible to define a decoder, which maps opcodes onto AST entries; for example:

```
instruction Decode (opc::bits(32)) =
   match opc
   {
      ...
      case '001100 rs rt imm' => ANDI (rs, rt, imm)
      ...
   }
```

This decoder can be used to define our next-state function, that is:

```
unit Next = Run (Decode (Fetch))
```

where 'Run' is defined by

```
unit Run (i::instruction) =
   match i
   {
      ...
      case ANDI (args) => run_ANDI (args)
      ...
   }
```

When working with full-scale architecture specifications (where the AST type is large), there are some clear overheads associated with implementing this approach (in comparison with the AST free version). It becomes challenging to maintain the AST datatype declaration and to keep it correctly synchronised with the instruction semantics operations (such as 'run_ANDI') and the definition of 'Run'. Note, however, that there is great uniformity in the AST datatype declaration and also in the definition of 'Run'. As such, L3 is able to automatically define both of these for the user.

To make use of this automation, the declaration of 'run_ANDI' is simply modified to:

```
define ANDI (rs::bits(5), rt::bits(5), imm::bits(16)) =
   GPR(rt) <- GPR(rs) && ZeroExtend (imm)
```

The 'define' keyword tells L3 to define a semantics function[14] (as before) and to also create a new 'instruction' datatype AST entry; furthermore, the two will be linked with the automatic construction of a 'Run' function. The 'instruction' type and 'Run' function become defined (in scope) following the declaration 'define Run'. After this declaration it is no longer possible to use the 'define' construct again; however, it does become possible to define functions such as 'Decode' and 'Next'.

Specifications that are based on using 'define' gain the benefits of declaring an instruction AST datatype without the downside of laborious coding overheads. In the example above, the overhead amounts to one extra line of code; this comes from defining a next-state function that calls the dedicated decode function, followed by a call to 'Run'.

One of the key advantages of declaring an instruction datatype is that it makes it considerably easier to develop tools that work with assembly code syntax (performing parsing, encoding

[14] This function is internally called 'dfn'ANDI' instead of 'run_ANDI'.

and pretty-printing). In this case, it is relatively easy, for example, to map between the MIPS instruction syntax `"andi $5, $6, 1024"` and the expression '`ANDI (5, 6, 1024)`'. Defining an encoder in L3 is relatively straightforward; for example:

```
bits(32) Encode (i::instruction) =
   match i
   {
      ...
      case ANDI (rs, rt, imm) => '001100' : rs : rt : imm
      ...
   }
```

Such a function can be used to write a MIPS assembler, which would map the assembly code syntax `"andi $5, $6, 1024"` onto the machine-code value `0x30c50400`.

2 Tutorial

This section gives a brief overview of how to install and use L3.

Installation The sources for L3 can be downloaded from the following webpage:

```
www.cl.cam.ac.uk/~acjf3/l3
```

The file `l3.tar.bz2` can be unpacked with the command:

```
$ tar xvjf l3.tar.bz2
```

This will create a directory of the form `L3-YYYY-MM-DD`, where the year, month and day will correspond with the release date. The current version of L3 depends upon Poly/ML 5.7, which can be obtained from the site `polyml.org`. If Poly/ML is installed then L3 can be built as follows:

```
$ cd L3-YYYY-MM-DD
$ make
```

Once built, the following should appear when you start L3:[15]

```
$ l3
<< L3 >>
>
```

Users should consider running L3 with the command `rlwrap l3`.[16]

[15] This assumes that the directory `L3-YYYY-MM-DD/bin` has been added to the `PATH` environment variable.
[16] Details on `rlwrap` utility can be found at the site `freecode.com/projects/rlwrap`.

Basic Interaction The command prompt for L3 is essentially the read-eval-print loop (REPL) of Poly/ML. As such, users are free to write arbitrary Standard ML code. The structure `Runtime` enables interaction with L3. The following shows an arithmetic evaluation in Standard ML, as well as in the L3 interpreter (`evalQ`):

```
> 1 + 2;
val it = 3: int
> Runtime.evalQ `0i1 + 2`;
val it = `0i3`: Term.term
```

Various error messages can occur when using the interpreter; for example:

```
> Runtime.evalQ `1 + 2`;
Exception- Fail "Could not infer all types" raised

> Runtime.evalQ `0i2 div 0`;
Exception- Except ("Div", SOME ("div" (int, [`0i2`, `0i0`]))) raised
```

The first error occurs because unannotated number literals are regarded as bit-vector values and here the vector width cannot be inferred.

L3 declarations can be made using the function `loadQ`. The following sequence shows the declaration and manipulation of a global variable:

```
> Runtime.loadQ `declare n :: nat`;
<VAR> n
val it = (): unit

> Runtime.evalQ `n`;
val it = `UNKNOWN::nat`: Term.term

> Runtime.evalQ `{ n <- 3; n }`;
val it = `0n3`: Term.term
```

The global state can be examined using the function `show`; for example:

```
> Runtime.show();
val it = [("n", `0n3`)]: Eval.term_stack
```

The state can be "wiped" clean using the function `reset` and all declarations can be wiped using '`resetAll`'; for example:

```
> Runtime.reset();
val it = (): unit
> Runtime.show();
val it = [("n", `UNKNOWN::nat`)]: Eval.term_stack

> Runtime.resetAll();
val it = (): unit
> Runtime.show();
val it = []: Eval.term_stack
```

The function '`LoadQ`' is similar to '`loadQ`' but it performs a '`resetAll`' before making declarations.

Constants The structure `Consts` provides various functions for working with L3 constants. Many of the operation in the `Consts` structure will be of little interest to end-users, however it can be used to display some helpful information.

The function `Consts.lookupConst` can be used to lookup an operation; for example:

```
> Consts.lookupConst "Log2";
val it = Primitive [(bits(a) -> bits(a), -), (nat -> nat, -)]: const
```

Since '`Log2`' is a primitive operation, only the valid types (ordered by overload priority) are show. With user operations, the internal representation (definition) of the operation will be shown; for example:

```
> Runtime.LoadQ `unit demo () = ()`;
> Consts.lookupConst "demo";
val it = Definition (Abs ([("_", (unit, -))], unit -> unit, `()`), NONE, 1):
   Consts.const
```

The functions `Consts.listDefinitions` and `Consts.listExceptions` provide details of all user operations and exceptions respectively. The function '`Consts.stats`' gives a summary of all the current declarations:

```
> Consts.stats();
Primitives ............... 188
Destructors............... 5
Constructors.............. 0
Accessors ................ 0
Exceptions................ 0
User operations.......... 1
Instruction definitions.. 0
```

Here the constructors for primitive types (such as '`Cons`') are counted under "primitives".

Types The `Types` structure can be used to display information relating to L3 types. In particular, this structure provides the functions `Types.lookupConst` and `Types.listConsts`, which are demonstrated below:

```
> Runtime.loadQ `construct enum {One Two Three}`;
> Types.listConsts();
val it =
  [("enum",
    {eq = true, ast = false, def = Constructors (Enum <dict(3)>), num = 0})]:
  (string * typeconst) list

> Types.lookupConst "int";
val it = SOME {eq = true, ast = false, def = BaseType, num = ~1}:
  typeconst option
```

Loading and Exporting Specifications When writing an L3 specification, it is expected that user declarations will be contained in an L3 source file. A sample instruction set specification (`tiny.spec`) can be found in the L3 distribution subdirectory `examples`. This specification is based on Thacker's pedagogical Tiny 3 architecture; it can be loaded as follows:

```
> Runtime.LoadF "tiny.spec";
Loading... tiny.spec
<TYP> regT
<TYP> wordT
...
<FUN> test_prog
Done.
```

The string argument to the functions `Runtime.loadF` and `Runtime.LoadF` can be in the form of a comma separated list of files, which will be loaded in order. That is, the call

```
Runtime.loadF "a.spec, b.spec"
```

will load `a.spec` followed by `b.spec`.

The function `HolExport.export` will export the current L3 specification in the form of an HOL4 script; for example:

```
> HolExport.export "tiny";
raise'exception ............................ ok.
function ................................... ok.
...
m'extend ................................... skip.
m'unextend ................................. skip.
Created file: tinyScript.sml
Created file: tinyLib.sml
Created file: tinyLib.sig
Done.
```

The function `HolExport.spec` provides a convenient shortcut for performing `LoadF` and `export` using just a single call; for example, `HolExport.spec ("tiny.spec", "tiny")` will perform the load and export presented above.

L3 is capable of exporting two styles of HOL specification: an explicitly monadic version (based on `state_transformerTheory`); and a version that directly manipulates state using let-expressions (which is the default style). For example, a monadic version of the Tiny specification can be exported using the command `HolExport.spec ("tiny.spec", "tiny monadic")`. Other export options include: `sigdocs`/ `nosigdocs` (for setting HOL's `TheoryPP.include_docs` trace variable), `bigrecords`/ `nobigrecords` (for controlling HOL's `Datatype.big_record_size`) and `underflowbefore`/ `nounderflowbefore` (for selecting the required IEEE 754 underflow behaviour).

Generated HOL4 specifications are not designed to be manually edited (or human readable). To build an L3 specification script, the HOL library `Import` is needed. This library is currently located in the directory `$HOL_DIR/examples/l3-machine-code/common` and so adding this path to the `INCLUDES` variable of a local `Holmakefile` is recommended. The `examples/l3-machine-code` directory contains various examples of using L3 generated specifications.

Testing Specifications Having written an L3 specification, it is fairly easy to write code (in Standard ML) that runs test vectors. Code for running the `tiny.spec` example can be found in the file `run-tiny.spec`. In this test harness, the Tiny state is initialised, and a small test program is loaded, simply by evaluating the expression '`initialize (test_prog)`'.

It is also possible to evaluate (run) models that have been exported to HOL4. The file `run-tiny.sml` shows how this can be accomplished for the Tiny example.

A Primitive Types and Operations

The primitive types are as follows:

```
unit bool char nat int bits(·) rounding
```

There are two infix type constructors: '`*`' for pairs, and '`->`' for maps. There are also postfix type constructors '`set`', '`option`' and '`list`'; for example, sets of strings are represented by the type '`string set`'. The following sections describe these primitive types, together with some details of associated operations. The following constants and operations are available:

nullary	`() Nil None false roundTiesToEven roundTowardNegative roundTowardPositive roundTowardZero true`
prefix	`! - ~ Abs Cardinality Cons Difference Drop Element FP32_Abs FP32_Add FP32_Equal FP32_IsNan FP32_LessThan FP32_Mul FP32_Neg FP32_Sub FP64_Abs FP64_Add FP64_Equal FP64_IsNan FP64_LessThan FP64_Mul FP64_Neg FP64_Sub FromBinString FromDecString FromHexString Fst Head IndexOf InitMap Intersection IsAlpha IsAlphaNum IsDigit IsHexDigit IsLower IsMember IsSome IsSubset IsUpper Length Log2 Max Min Msb QuotRem Remove RemoveDuplicates RemoveExcept Reverse SetOfList SignExtend SignedMax SignedMin Size Snd Some Tail Take ToLower ToUpper Union Update ValOf ZeroExtend not`
infix	`!= && * ** + , - : < <+ <= <=+ <> == > >+ >= >=+ ?? @ ^ || >> << >>+ #<< #>> and div in insert mod notin or quot rem`
`sdiv`	
`smod`	
other	`·<·> ·<·:·> [·] fields splitl splitr tokens`

Many arithmetic operations are *overloaded*. Unless otherwise stated, the order of selection is: '`bits(·)`', '`nat`', '`int`' and then '`bool list`'.

A.1 Unit

The type '`unit`' represents the singleton set containing the value '`()`'. The primary use of this type is as the return type for procedures.

A.2 Bool

The type '`bool`' represents the two element set $\mathbb{B} = \{\text{`true'}, \text{`false'}\}$. The following standard Boolean operations are available:

negation	`not`, `!`	$\texttt{bool} \to \texttt{bool}$
conjunction	`and`	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$
disjunction	`or`	$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$
equality	`==`	$\underline{\alpha} \times \underline{\alpha} \to \texttt{bool}$
non-equality	`<>`, `!=`	$\underline{\alpha} \times \underline{\alpha} \to \texttt{bool}$

The infix operators '`and`' and '`or`' are interpreted with *short-circuit evaluation*; for example, in the expression '`false and f()`' the value of '`f()`' is not computed, as the overall expression will always evaluate to false. This short-circuiting influences the side-effects that occur during expression evaluation.

There are two prefix symbols for logical negation (these are semantically identical) with '`!`' binding slightly tighter than '`not`'. This distinction is becomes significant when working with equalities; for example: '`not a == b`' is taken to be $\neg(a = b)$, whereas '`!a == b`' is $(\neg a) = b$.

Equality is only defined with respect to *equality types*, represented here by the type variable $\underline{\alpha}$. An equality type is any type built without the used of the map operator '`->`'.

A.3 Nat

The type '`nat`' represents the natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$. The following operations are available:

logarithm (base two)	`Log2`	`nat` $\to$ `nat`
minimum value	`Min`	`nat` $\times$ `nat` $\to$ `nat`
maximium value	`Max`	`nat` $\times$ `nat` $\to$ `nat`
less than	`<`	`nat` $\times$ `nat` $\to$ `bool`
greater than	`>`	`nat` $\times$ `nat` $\to$ `bool`
less than or equal to	`<=`	`nat` $\times$ `nat` $\to$ `bool`
greater than or equal to	`>=`	`nat` $\times$ `nat` $\to$ `bool`
addition	`+`	`nat` $\times$ `nat` $\to$ `nat`
subtraction	`-`	`nat` $\times$ `nat` $\to$ `nat`
multiplication	`*`	`nat` $\times$ `nat` $\to$ `nat`
exponentiation	`**`	`nat` $\times$ `nat` $\to$ `nat`
division	`div`	`nat` $\times$ `nat` $\to$ `nat`
modulus	`mod`	`nat` $\times$ `nat` $\to$ `nat`

The '`div`' and '`mod`' operations satisfy the following property:

$$\forall n.\ 0 < n \implies \forall k.\ k = (k \texttt{ div } n) * n + (k \texttt{ mod } n) \wedge (k \texttt{ mod } n) < n\ .$$

The expressions '`Log2(0)`', 'x `div 0`' and 'x `mod 0`' are all *undefined*.[17]

A.4 Int

The type '`int`' represents the integers $\mathbb{Z} = \mathbb{N} \cup \{-1, -2, -3, \dots\}$. The following operations are available:

[17] The interpreter will raise an exception when attempting to evaluate them.

unary minus	`-`	`int` → `int`
absolute value	`Abs`	`int` → `int`
minimum value	`Min`	`int` × `int` → `int`
maximum value	`Max`	`int` × `int` → `int`
less than	`<`	`int` × `int` → `bool`
greater than	`>`	`int` × `int` → `bool`
less than or equal to	`<=`	`int` × `int` → `bool`
greater than or equal to	`>=`	`int` × `int` → `bool`
addition	`+`	`int` × `int` → `int`
subtraction	`-`	`int` × `int` → `int`
multiplication	`*`	`int` × `int` → `int`
exponentiation	`**`	`int` × `nat` → `int`
quotient and remainder	`QuotRem`	`int` × `int` → `int` × `int`
quotient	`quot`	`int` × `int` → `int`
remainder	`rem`	`int` × `int` → `int`
division	`div`	`int` × `int` → `int`
modulus	`mod`	`int` × `int` → `int`

The operation '`Abs`' gives an positive integer value; for example, '`Abs(-2)`' = '`2`'. The integer operations '`quot`' and '`div`' are related to natural number division as follows:

$$i \texttt{ quot } j = \begin{cases} \mathrm{z}(\mathrm{n}(i) \texttt{ div } \mathrm{n}(j)) & 0 < j \wedge 0 \leq i \\ -\mathrm{z}(\mathrm{n}(-i) \texttt{ div } \mathrm{n}(j)) & 0 < j \wedge i < 0 \\ -\mathrm{z}(\mathrm{n}(i) \texttt{ div } \mathrm{n}(-j)) & j < 0 \wedge 0 \leq i \\ \mathrm{z}(\mathrm{n}(-i) \texttt{ div } \mathrm{n}(-j)) & j < 0 \wedge i < 0 \\ \textit{undefined} & \mathrm{j} = 0 \end{cases}$$

and

$$i \texttt{ div } j = \begin{cases} \mathrm{z}(\mathrm{n}(i) \texttt{ div } \mathrm{n}(j)) & 0 < j \wedge 0 \leq i \\ -\mathrm{z}(\mathrm{n}(-i) \texttt{ div } \mathrm{n}(j)) & 0 < j \wedge i < 0 \wedge \mathrm{n}(-i) \texttt{ mod } \mathrm{n}(j) = 0 \\ -\mathrm{z}(\mathrm{n}(-i) \texttt{ div } \mathrm{n}(j)) - 1 & 0 < j \wedge i < 0 \wedge \mathrm{n}(-i) \texttt{ mod } \mathrm{n}(j) \neq 0 \\ -\mathrm{z}(\mathrm{n}(i) \texttt{ div } \mathrm{n}(-j)) & j < 0 \wedge 0 \leq i \wedge \mathrm{n}(i) \texttt{ mod } \mathrm{n}(-j) = 0 \\ -\mathrm{z}(\mathrm{n}(i) \texttt{ div } \mathrm{n}(-j)) - 1 & j < 0 \wedge 0 \leq i \wedge \mathrm{n}(i) \texttt{ mod } \mathrm{n}(-j) \neq 0 \\ \mathrm{z}(\mathrm{n}(-i) \texttt{ div } \mathrm{n}(-j)) & j < 0 \wedge i < 0 \\ \textit{undefined} & \mathrm{j} = 0 \end{cases}$$

where $\mathrm{z} : \mathbb{N} \to \mathbb{Z}$ and $\mathrm{n} : \mathbb{Z} \to \mathbb{N}$ are value preserving maps. The '`rem`' and '`mod`' operations satisfy the equations:

$$i \texttt{ rem } j = \begin{cases} i - (i \texttt{ quot } j) * j & j \neq 0 \\ \textit{undefined} & \mathrm{j} = 0 \end{cases} \qquad i \texttt{ mod } j = \begin{cases} i - (i \texttt{ div } j) * j & j \neq 0 \\ \textit{undefined} & \mathrm{j} = 0 \,. \end{cases}$$

A.5 Bit-vectors

The type '`bits(`n`)`' represents fixed-width (n-bit) words, also known as bit-vectors; for example, `bits(32)` is the type of 32-bit machine words.[18] The bits type can be viewed as representing

[18] These are called *double-words* (DWORD) in the x86 architecture and simply *words* in the ARM architecture.

the set $\mathbb{Z}_n = \{0, 1, \ldots, 2^{n-1}\}$, for some $0 < n$. At the *type system level* no distinction is made between signed and unsigned words; however, some operations (such as orderings) have signed and unsigned variants.

The following bit-vector operations are available:

1's complement	`~`	`bits(`n`)` → `bits(`n`)`
2's complement	`-`	`bits(`n`)` → `bits(`n`)`
absolute value	`Abs`	`bits(`n`)` → `bits(`n`)`
logarithm (base two)	`Log2`	`bits(`n`)` → `bits(`n`)`
zero-extend	`ZeroExtend`	`bits(`m`)` → `bits(`n`)`
sign-extend	`SignExtend`	`bits(`m`)` → `bits(`n`)`
reverse bits	`Reverse`	`bits(`n`)` → `bits(`n`)`
minimum value (*u*)	`Min`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
maximum value (*u*)	`Max`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
minimum value (*s*)	`SignedMin`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
maximum value (*s*)	`SignedMax`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
width/size	`Size`	`bits(`n`)` → `nat`
most-significant (sign) bit	`Msb`	`bits(`n`)` → `bool`
bit value	`·<·>`	`bits(`n`)` × `nat` → `bool`
bit field	`·<·:·>`	`bits(`m`)` × `nat` × `nat` → `bits(`n`)`
concatenation	`:`	`bits(`m`)` × `bits(`n`)` → `bits(`p`)`
duplication	`^`	`bits(`m`)` × `nat` → `bits(`n`)`
less than (*s*)	`<`	`bits(`n`)` × `bits(`n`)` → `bool`
greater than (*s*)	`>`	`bits(`n`)` × `bits(`n`)` → `bool`
less than or equal to (*s*)	`<=`	`bits(`n`)` × `bits(`n`)` → `bool`
greater than or equal to (*s*)	`>=`	`bits(`n`)` × `bits(`n`)` → `bool`
less than (*u*)	`<+`	`bits(`n`)` × `bits(`n`)` → `bool`
greater than (*u*)	`>+`	`bits(`n`)` × `bits(`n`)` → `bool`
less than or equal to (*u*)	`<=+`	`bits(`n`)` × `bits(`n`)` → `bool`
greater than or equal to (*u*)	`>=+`	`bits(`n`)` × `bits(`n`)` → `bool`
addition	`+`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
subtraction	`-`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
multiplication	`*`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
quotient (*s*)	`quot`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
remained (*s*)	`rem`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
division (*u*)	`div`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
modulus (*u*)	`mod`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
division (*s*)	`sdiv`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
modulus (*s*)	`smod`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
bitwise conjunction	`&&`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
bitwise disjunction	`||`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`
bitwise exclusive-or	`??`	`bits(`n`)` × `bits(`n`)` → `bits(`n`)`

Unsigned operations are labelled with (*u*) and signed operations are labelled with (*s*).

The shift operations support shifting by either a bit-vector value or by a natural number value. The overloading is resolved such that the '`nat`' variant is selected over the '`bits(·)`' version. The shift operations are:

left-shift	`<<`	`bits(`n`) × nat → bits(`n`)` `bits(`n`) × bits(`n`) → bits(`n`)`
right-shift (*u*)	`>>+`	`bits(`n`) × nat → bits(`n`)` `bits(`n`) × bits(`n`) → bits(`n`)`
right-shift (*s*)	`>>`	`bits(`n`) × nat → bits(`n`)` `bits(`n`) × bits(`n`) → bits(`n`)`
rotate left	`#<<`	`bits(`n`) × nat → bits(`n`)` `bits(`n`) × bits(`n`) → bits(`n`)`
rotate right	`#>>`	`bits(`n`) × nat → bits(`n`)` `bits(`n`) × bits(`n`) → bits(`n`)`

A number of operations are *undefined* depending on the types of their domain and range. This is detailed in the following table:

Operation	*Undefined* when
`ZeroExtend(`x`)`m	$m < n$
`SignExtend(`x`)`m	$m < n$
`(`x`)<`p`>`	$m \leq p$
`(`x`)<`h`:`l`>`n	$n \neq h + 1 - l \vee h < l \vee m \leq h$
`((`x`'`m`) : (`y`'`n`))`p	$p \neq m + n$
`((`x`)^`n`)`p	$p \neq m \cdot n$ (which covers $n = 0$)

The type checker will fail when these conditions are obviously violated. Alternatively, the interpreter may raise a runtime error when these conditions are not met.

As before the expressions '`Log2(0)`', 'x `quot 0`', 'x `rem 0`', 'x `div 0`' and 'x `mod 0`' are all *undefined*.

A.6 Bit-strings

Various bit-string operation are defined over the type '`bool list`'. These are listed below:

bit value	`·<·>`	`bool list × nat → bool`
bit field	`·<·:·>`	`bool list × nat × nat → bool list`
concatenation	`:`	`bool list × bool list → bool list`
duplication	`^`	`bool list × nat → bool list`
addition	`+`	`bool list × bool list → bool list`
bitwise conjunction	`&&`	`bool list × bool list → bool list`
bitwise disjunction	`||`	`bool list × bool list → bool list`
bitwise exclusive-or	`??`	`bool list × bool list → bool list`
left-shift	`<<`	`bool list × nat → bool list`
right-shift	`>>+`	`bool list × nat → bool list`
rotate-right	`#>>`	`bool list × nat → bool list`

The bit value operation gives the truth value of a bit-string at a given bit position, for example:

'`0s1101<0>`' = '`true`', '`0s1101<1>`' = '`false`', '`0s1101<2>`' = '`true`',
'`0s1101<3>`' = '`true`', '`0s1101<4>`' = '`false`', '`0s1101<5>`' = '`false`'.

The least-significant bit (position zero) occurs to the right of the bit-string. The bit value operation gives '`false`' when the bit position is greater than the size of the bit-string.

The bit field operation extracts a range of bits from a bit-string. For example:

'`0s1101<3:1>`' = '`0s110`' and '`0s1101<5:2>`' = '`0s0011`' .

In the bit-string expression x`<`h`:`l`>` the size of the result is always $h + 1 - l$ and the result is *undefined* when $h < l$.

Bit-string addition does not require the arguments to be of the same size and it gives a non-truncated result. For example,

'`0s101 + 0s1`' = '`0s110`' and '`0s101 + 0s11`' = '`0s1000`' .

The operation `<<` performs a left-shift; for example, '`0s1101 << 2`' = '`0s110100`'. The operation '`:`' is bit-string concatenation and '`^`' is duplication; for example:

'`0s101 : 0s11`' = '`0s10111`' and '`0s101 ^ 3`' = '`0s101101101`' .

Regular list operations work as one would expect over bit-strings; for example, '`Head (0s101)`' is a valid expression with value '`true`'.

A.7 Characters and Strings

The following built-in functions are available for characters and strings:

is alphabetic	`IsAlpha`	`char → bool`
is alpha-numeric	`IsAlphaNum`	`char → bool`
is decimal digit	`IsDigit`	`char → bool`
is hexadecimal digit	`IsHexDigit`	`char → bool`
is lowercase	`IsLower`	`char → bool`
is uppercase	`IsUpper`	`char → bool`
parse binary	`FromBinString`	`string → num option`
parse decimal	`FromDecString`	`string → num option`
parse hexadecimal	`FromHexString`	`string → num option`
convert to lowercase	`ToLower`	`string → string`
convert to uppercase	`ToUpper`	`string → string`

The following string operations are useful for parsing:

split from the left	`splitl`	`(char → bool) × string → string × string`
split from the right	`splitr`	`(char → bool) × string → string × string`
split into fields	`fields`	`(char → bool) × string → string list`
split into tokens	`tokens`	`(char → bool) × string → string list`

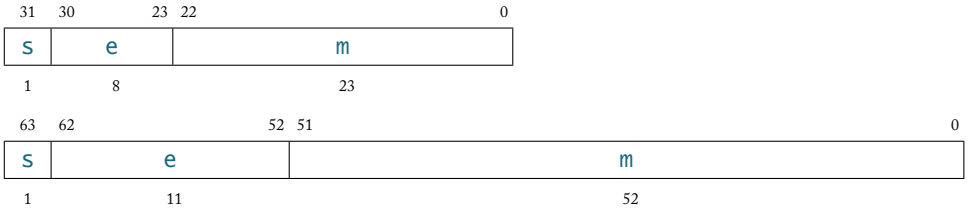
These are based on the SML functions `Substring.splitl`, `Substring.splitr`, `String.fields` and `String.tokens`. Character predicates are declared using special syntax; for example, the expression:

```
splitl space or hexdigit and not in set {#"a", #"f"} (" 12bcab")
```

gives the result '`(" 12bc", "ab")`'.

A.8 Floating-point

The L3 language does not provide specialised types for representing floating-point numbers. However, 32-bit (single precision) and 64-bit (double precision) bit-vector values can be treated as floating-point numbers; the following IEEE compliant encoding formats are assumed:



Here '`s`' is the sign bit (set to '`1`' for negative numbers), '`e`' is a biased exponent value and '`m`' is the significand. The following constants and operations are available (where '`fp`' stands for `bits(32)` or `bits(64)`, depending on the name of the operation):

rounding modes	`roundTiesToEven`, `roundTowardPositive`, `roundTowardNegative`, `roundTowardZero`	`ieee_rounding`
flags (record)	`DivideByZero`, `InvalidOp`, `Overflow`, `Precision`, `Underflow`	`ieee_flags→bool`
comparison values	`FP_LT`, `FP_EQ`, `FP_GT`, `FP_UN`	`ieee_compare`
convert format	`FP32_ToFP64`	`fp32→fp64`
convert format	`FP64_ToFP32`	`ieee_rounding × fp64→fp32`
convert format	`FP64_ToFP32_`	`ieee_rounding × fp64→ ieee_flags × fp32`
from integer	`FP{32|64}_FromInt`	`ieee_rounding × int→fp`
to integer	`FP{32|64}_ToInt`	`ieee_rounding × fp→int option`
from string	`FP{32|64}`	`string→fp`
negative zero	`FP{32|64}_NegZero`	`fp`
positive zero	`FP{32|64}_PosZero`	`fp`
negative infinity	`FP{32|64}_NegInf`	`fp`
positive infinity	`FP{32|64}_PosInf`	`fp`
a quiet NaN	`FP{32|64}_qNan`	`fp`
a signalling NaN	`FP{32|64}_sNan`	`fp`
is integral	`FP{32|64}_IsIntegral`	`fp→bool`
is finite	`FP{32|64}_IsFinite`	`fp→bool`
is NaN	`FP{32|64}_IsNan`	`fp→bool`
is signalling NaN	`FP{32|64}_IsSignallingNan`	`fp→bool`
is normal	`FP{32|64}_IsNormal`	`fp→bool`
is sub-normal	`FP{32|64}_IsSubnormal`	`fp→bool`
is zero	`FP{32|64}_IsZero`	`fp→bool`

comparison	`FP{32|64}_Compare`	`fp` × `fp`→`ieee_compare`
equality	`FP{32|64}_Equal`	`fp` × `fp`→`bool`
less than	`FP{32|64}_LessThan`	`fp` × `fp`→`bool`
less than or equal	`FP{32|64}_LessEqual`	`fp` × `fp`→`bool`
greater than	`FP{32|64}_GreaterThan`	`fp` × `fp`→`bool`
greater than or equal	`FP{32|64}_GreaterEqual`	`fp` × `fp`→`bool`

absolute	`FP{32|64}_Abs`	`fp`→`fp`
negation	`FP{32|64}_Neg`	`fp`→`fp`
square root	`FP{32|64}_Sqrt`	`ieee_rounding` × `fp`→`fp`
(with flags)	`FP{32|64}_Sqrt_`	`ieee_rounding` × `fp`→`ieee_flags` × `fp`
addition	`FP{32|64}_Add`	`ieee_rounding` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_Add_`	`ieee_rounding` × `fp` × `fp`→ `ieee_flags` × `fp`
subtraction	`FP{32|64}_Sub`	`ieee_rounding` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_Sub_`	`ieee_rounding` × `fp` × `fp`→ `ieee_flags` × `fp`
multiplication	`FP{32|64}_Mul`	`ieee_rounding` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_Mul_`	`ieee_rounding` × `fp` × `fp`→ `ieee_flags` × `fp`
fused multiply-add	`FP{32|64}_MulAdd`	`ieee_rounding` × `fp` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_MulAdd_`	`ieee_rounding` × `fp` × `fp` × `fp`→ `ieee_flags` × `fp`
fused multiply-sub	`FP{32|64}_MulSub`	`ieee_rounding` × `fp` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_MulSub_`	`ieee_rounding` × `fp` × `fp` × `fp`→ `ieee_flags` × `fp`
division	`FP{32|64}_Div`	`ieee_rounding` × `fp` × `fp`→`fp`
(with flags)	`FP{32|64}_Div_`	`ieee_rounding` × `fp` × `fp`→ `ieee_flags` × `fp`

The L3 interpreter supports evaluation for 32-bit and 64-bit floating-point expressions but only when running on an x86 machine that supports SSE2 extensions (Pentium 4 and later). The fused multiply-add/sub operations can only be interpreted on machines supporting the FMA extension, which came in with the 4th generation (Haswell) Intel processors. These restrictions also apply to any Standard ML code that is generated by L3.

The IEEE 754 standard permits a number of different ways to flag *underflow*. L3 predominantly supports the x86 interpretation, which detects underflow *after rounding*. However, when exporting to HOL4 it is possible to choose detection *before rounding* which, for example, is suited to ARM instruction set models.

A.9 Miscellaneous operations

The following operations are available:

pair constructor	`,`	$\alpha \times \beta \to \alpha \times \beta$
first of pair	`Fst`	$\alpha \times \beta \to \alpha$
second of pair	`Snd`	$\alpha \times \beta \to \beta$
casting	`[·]`	$\hat{\alpha} \to \hat{\beta}$
initialise a map	`InitMap`	$\alpha \to (\beta \to \alpha)$

option constructor	`Some`	$\alpha \rightarrow \alpha$ `option`
option projection	`ValOf`	α `option` $\rightarrow \alpha$
option test	`IsSome`	α `option` $\rightarrow$ `bool`

set membership	`in`	$\underline{\alpha} \times \underline{\alpha}$ `set` $\rightarrow$ `bool`
set non-membership	`notin`	$\underline{\alpha} \times \underline{\alpha}$ `set` $\rightarrow$ `bool`
set insertion	`insert`	$\underline{\alpha} \times \underline{\alpha}$ `set` $\rightarrow \underline{\alpha}$ `set`
set union	`Union`	$\underline{\alpha}$ `set` $\times \underline{\alpha}$ `set` $\rightarrow \underline{\alpha}$ `set`
set intersection	`Intersect`	$\underline{\alpha}$ `set` $\times \underline{\alpha}$ `set` $\rightarrow \underline{\alpha}$ `set`
set difference	`Difference`	$\underline{\alpha}$ `set` $\times \underline{\alpha}$ `set` $\rightarrow \underline{\alpha}$ `set`
set subset test	`IsSubset`	$\underline{\alpha}$ `set` $\times \underline{\alpha}$ `set` $\rightarrow$ `bool`
set cardinality	`Cardinality`	$\underline{\alpha}$ `set` $\rightarrow$ `nat`
set from a list	`SetOfList`	$\underline{\alpha}$ `list` $\rightarrow \underline{\alpha}$ `set`

empty list	`Nil`	α `list`
list constructor	`Cons`, `@`	$\alpha \times \alpha$ `list` $\rightarrow \alpha$ `list`
list head	`Head`	α `list` $\rightarrow \alpha$
list tail	`Tail`	α `list` $\rightarrow \alpha$ `list`
list length	`Length`	α `list` $\rightarrow$ `nat`
list concatenation	`:`	α `list` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list flatten	`Concat`	α `list list` $\rightarrow \alpha$ `list`
list element	`Element`	`nat` $\times \alpha$ `list` $\rightarrow \alpha$
list prefix	`Take`	`nat` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list drop prefix	`Drop`	`nat` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list pad left	`PadLeft`	$\alpha \times$ `nat` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list pad right	`PadRight`	$\alpha \times$ `nat` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list update	`Update`	$\alpha \times$ `nat` $\times \alpha$ `list` $\rightarrow \alpha$ `list`
list difference	`Remove`	$\underline{\alpha}$ `list` $\times \underline{\alpha}$ `list` $\rightarrow \underline{\alpha}$ `list`
list intersection	`RemoveExcept`	$\underline{\alpha}$ `list` $\times \underline{\alpha}$ `list` $\rightarrow \underline{\alpha}$ `list`
list without duplicates	`RemoveDuplicates`	$\underline{\alpha}$ `list` $\rightarrow \underline{\alpha}$ `list`
list membership	`IsMember`	$\underline{\alpha} \times \underline{\alpha}$ `list` $\rightarrow$ `bool`
list member position	`IndexOf`	$\underline{\alpha} \times \underline{\alpha}$ `list` $\rightarrow$ `nat option`

Sets are only defined over equality types, see Appendix A.2. The casting map is defined over *cast-able* types (as represented by the polymorphic type variables $\hat{\alpha}$ and $\hat{\beta}$): these are the primitive types (with the exception of '`unit`') and user defined enumerated types. Assume '`Enum`' and '`Enum2`' are declared with:

```
construct Enum { One, Two, Three }
construct Enum2 { Four, Five, Six, Seven }
```

The cast operations are described in Tables 1 to 7.

Table 1: Cast to ‘`bool`’

From	Operation	Examples
`bool`	identity	‘`[true]::bool`’ = ‘`true`’
`string`	parse bool	‘`["true"]::bool`’ = ‘`true`’ ‘`["false"]::bool`’ = ‘`false`’ otherwise *undefined*
`nat`	is not zero	‘`[0n3]::bool`’ = ‘`true`’ ‘`[0n0]::bool`’ = ‘`false`’
`int`	is not zero	‘`[-0i1]::bool`’ = ‘`"true"`’ ‘`[0i0]::bool`’ = ‘`false`’
`bool list`	is not zero	‘`[0s11]::bool`’ = ‘`true`’ ‘`[0s0]::bool`’ = ‘`false`’
`bits(·)`	is not zero	‘`[3`3]::bool`’ = ‘`true`’ ‘`[0`3]::bool`’ = ‘`false`’
`Enum`	is not the first element	‘`[One]::bool`’ = ‘`false`’ ‘`[Two]::bool`’ = ‘`true`’

Table 2: Cast to ‘`string`’

From	Operation	Examples
`bool`	name	‘`[true]::string`’ = ‘`"true"`’
`string`	identity	‘`["hello"]::string`’ = ‘`"hello"`’
`nat`	decimal	‘`[0n3]::string`’ = ‘`"3"`’
`int`	signed decimal	‘`[-0i1]::string`’ = ‘`"~1"`’
`bool list`	binary	‘`[0s11]::string`’ = ‘`"11"`’
`bits(·)`	unsigned hexadecimal	‘`[139`8]::string`’ = ‘`"8B"`’
`Enum`	name	‘`[One]::string`’ = ‘`"One"`’

Table 3: Cast to '`nat`'

From	Operation	Examples
`bool`	bit value	'`[false]::nat`' = '`0n0`' '`[true]::nat`' = '`0n1`'
`string`	parse decimal	'`["123"]::nat`' = '`0n123`' '`[""]::nat`' is *undefined* '`["-1"]::nat`' is *undefined* '`["hello"]::nat`' is *undefined*
`nat`	identity	'`[0n123]::nat`' = '`0n123`'
`int`	value	'`[0i123]::nat`' = '`0n123`' '`[-0i1]::nat`' is *undefined*
`bool list`	value	'`[0s11]::nat`' = '`0n3`'
`bits(·)`	unsigned value	'`[139`8`]::nat`' = '`0n139`'
`Enum`	enumerate (count from zero)	'`[One]::nat`' = '`0n0`' '`[Two]::nat`' = '`0n1`'

Table 4: Cast to '`int`'

From	Operation	Examples
`bool`	bit value	'`[false]::int`' = '`0i0`' '`[true]::int`' = '`0i1`'
`string`	parse decimal	'`["123"]::int`' = '`0i123`' '`["-123"]::int`' = '`-0i123`' '`["~123"]::int`' = '`-0i123`' '`[""]::int`' is *undefined* '`["hello"]::int`' is *undefined*
`nat`	value	'`[0n123]::int`' = '`0i123`'
`int`	identity	'`[0i123]::int`' = '`0i123`'
`bool list`	value	'`[0s11]::int`' = '`0i3`'
`bits(·)`	signed value	'`[123`8`]::int`' = '`0i123`' '`[130`8`]::int`' = '`-0i126`'
`Enum`	enumerate	'`[One]::int`' = '`0i0`'

Table 5: Cast to '`bit-string`'

From	Operation	Examples
`bool`	bit value	'`[false]::bool list`' = '`0s0`' '`[true]::bool list`' = '`0s1`'
`string`	parse binary	'`["101"]::bool list`' = '`0s101`' '`["123"]::bool list`' is *undefined*
`nat`	value	'`[0n123]::bool list`' = '`0s1111011`'
`int`	value	'`[0i123]::bool list`' = '`0s1111011`' '`[-0i1]::bool list`' is *undefined*
`bool list`	identity	'`[0s11]::bool list`' = '`0s11`'
`bits(·)`	unsigned value	'`[123`8]::bool list`' = '`0s01111011`'
`Enum`	enumerate	'`[One]::bool list`' = '`0s0`'

Table 6: Cast to '`bits(`n`)`'

From	Operation	Examples
`bool`	bit value	'`[false]`8`' = '`0x0`8`' '`[true]`8`' = '`0x1`8`'
`string`	parse hexadecimal	'`["8B"]`8`' = '`0x8B`8`' '`["-A"]`8`' = '`0xF6`8`' '`["100"]`8`' is *undefined* (too big)
`nat`	unsigned value $\pmod{2^n}$	'`[0n123]`8`' = '`0x7B`8`' '`[0n256]`8`' = '`0x0`8`'
`int`	signed value $\pmod{2^n}$	'`[0i123]`8`' = '`0x7B`8`' '`[0i256]`8`' = '`0x0`8`' '`[-0i1]`8`' = '`0xFF`8`' '`[-0i133]`8`' = '`0x7B`8`'
`bool list`	unsigned value $\pmod{2^n}$	'`[0s1111]`8`' = '`0xF`8`' '`[0s1000000001]`8`' = '`0x1`8`'
`bits(`p`)`	identity / zero extend ($p \leq n$) truncate ($n < p$)	'`[0x81`8]`10`' = '`0x81`10`' '`[0xFF0`12]`8`' = '`0xF0`8`'
`Enum`	enumerate	'`[One]`8`' = '`0s0`' '`[Three]`1`' is *undefined* (too big)

Table 7: Cast to '`Enum`'

From	Operation	Examples
`bool`	type error	
`string`	parse	'`["One"]::Enum`' = '`One`' '`["hello"]::Enum`' is *undefined*
`nat`	select	'`[0n1]::Enum`' = '`Two`' '`[0n3]::Enum`' is *undefined*
`int`	select	'`[0i1]::Enum`' = '`Two`' '`[0i3]::Enum`' is *undefined* '`[-0i1]::Enum`' is *undefined*
`bool list`	select	'`[0s1]`8`' = '`Two`' '`[0s11]::Enum`' is *undefined*
`bits(·)`	select	'`[1`8]::Enum`' = '`Two`' '`[3`8]::Enum`' is *undefined*
`Enum`	identity	'`[One]::Enum`' = '`One`'
`Enum2`	select	'`[Four]::Enum`' = '`One`' '`[Seven]::Enum`' is *undefined*

B Syntax

Reserved words

The names of all primitive types, operators and constants are reserved, see Appendix A. Other reserved words are:

```
assign case clear component construct declare define do else exception for
foreach if list match nothing pattern patterns record register return set then
type var when RAO! RAZ! UNK! UNKNOWN
```

Comments

Inline comments start with '`--`'. Block comments are delimited by '`{-`' and '`-}`', and can be nested. All commented code is ignored (not parsed).

Number literals

In the following, no white-space is permitted.

⟨*bin*⟩ ::= '`0`' | '`1`'

⟨*dec*⟩ ::= ⟨*bin*⟩ | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

⟨*hex*⟩ ::= ⟨*dec*⟩ | '`a`' | '`b`' | '`c`' | '`d`' | '`e`' | '`f`' | '`A`' | '`B`' | '`C`' | '`D`' | '`E`' | '`F`'

⟨*binary*⟩ ::= ⟨*bin*⟩ | ⟨*bin*⟩ ⟨*binary*⟩ | ⟨*bin*⟩ '`_`' ⟨*binary*⟩

⟨*decimal*⟩ ::= ⟨*dec*⟩ | ⟨*dec*⟩ ⟨*decimal*⟩ | ⟨*dec*⟩ '`_`' ⟨*decimal*⟩

⟨*hexadecimal*⟩ ::= ⟨*hex*⟩ | ⟨*hex*⟩ ⟨*hexadecimal*⟩ | ⟨*hex*⟩ '`_`' ⟨*hexadecimal*⟩

⟨*number*⟩ ::= ⟨*decimal*⟩ | '`0b`' ⟨*binary*⟩ | '`0d`' ⟨*decimal*⟩ | '`0x`' ⟨*hexadecimal*⟩
| '`0n`' ⟨*decimal*⟩ | '`0nb`' ⟨*binary*⟩ | '`0nd`' ⟨*decimal*⟩ | '`0nx`' ⟨*hexadecimal*⟩

⟨*numeric*⟩ ::= ⟨*number*⟩
| '`0i`' ⟨*decimal*⟩ | '`0ib`' ⟨*binary*⟩ | '`0id`' ⟨*decimal*⟩ | '`0ix`' ⟨*hexadecimal*⟩
| '`0s`' ⟨*decimal*⟩ | '`0sb`' ⟨*binary*⟩ | '`0sd`' ⟨*decimal*⟩ | '`0sx`' ⟨*hexadecimal*⟩
| '`0w`' ⟨*decimal*⟩ | '`0wb`' ⟨*binary*⟩ | '`0wd`' ⟨*decimal*⟩ | '`0wx`' ⟨*hexadecimal*⟩

Identifiers

⟨*alpha*⟩ ::= '`a`' .. '`z`' | '`A`' .. '`Z`'

⟨*alphanum*⟩ ::= ⟨*alpha*⟩ | ⟨*dec*⟩ | '`_`'

⟨*alphanums*⟩ ::= ⟨*alphanum*⟩ | ⟨*alphanum*⟩ ⟨*alphanums*⟩

⟨*name*⟩ ::= ⟨*alpha*⟩ | ⟨*alpha*⟩⟨*alphanums*⟩

Types

In the following, white-space is permitted.

⟨*type*⟩ ::= ⟨*tuple-type*⟩ | ⟨*tuple-type*⟩ '`->`' ⟨*tuple-type*⟩

⟨*tuple-type*⟩ ::= ⟨*base-type*⟩ | ⟨*base-type*⟩ '`*`' ⟨*tuple-type*⟩

⟨*base-type*⟩ ::= '`unit`' | '`bool`' | '`string`' | '`nat`' | '`int`'
| '`bits`' '`(`' ⟨*bits-type*⟩ '`)`'
| ⟨*base-type*⟩ '`list`'
| ⟨*base-type*⟩ '`option`'
| ⟨*base-type*⟩ '`set`'
| ⟨*name*⟩
| '`(`' ⟨*type*⟩ '`)`'

⟨*bits-type*⟩ ::= ⟨*name*⟩ | ⟨*number*⟩

⟨*annotation*⟩ ::= '`::`' ⟨*type*⟩ | '`` ` ``' ⟨*bits-type*⟩

⟨*a-name*⟩ ::= ⟨*name*⟩ ⟨*annotation*⟩

⟨*opt-a-name*⟩ ::= ⟨*name*⟩ | ⟨*a-name*⟩

Specifications

⟨*specification*⟩ ::= ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*specification*⟩

⟨*declaration*⟩ ::= ⟨*definition*⟩
| '`type`' ⟨*name*⟩ '`=`' ⟨*type*⟩
| '`construct`' ⟨*name*⟩ '`{`' ⟨*constructs*⟩ '`}`'
| '`record`' ⟨*name*⟩ '`{`' ⟨*arguments*⟩ '`}`'
| '`register`' ⟨*a-name*⟩ '`{`' ⟨*fields*⟩ '`}`'
| '`declare`' ⟨*globals*⟩
| '`exception`' ⟨*opt-a-name*⟩
| '`pattern`' ⟨*hints*⟩
| '`clear`' '`pattern`' ⟨*name-list*⟩
| '`clear`' '`patterns`'

⟨*constructs*⟩ ::= ⟨*opt-a-name*⟩
| ⟨*opt-a-name*⟩ ⟨*constructs*⟩
| ⟨*opt-a-name*⟩ '`,`' ⟨*constructs*⟩

⟨*globals*⟩ ::= ⟨*a-name*⟩ | '`{`' ⟨*arguments*⟩ '`}`'

⟨*arguments*⟩ ::= ⟨*a-name*⟩
| ⟨*a-name*⟩ ⟨*arguments*⟩
| ⟨*a-name*⟩ '`,`' ⟨*arguments*⟩

⟨*hints*⟩ ::= ⟨*hint*⟩ | '`{`' ⟨*hint-sequence*⟩ '`}`'

⟨*hint-sequence*⟩ ::= ⟨*hint*⟩ | ⟨*hint*⟩ ⟨*hint-sequence*⟩ | ⟨*hint*⟩ '`,`' ⟨*hint-sequence*⟩

⟨*hint*⟩ ::= ⟨*names*⟩ ⟨*annotation*⟩

⟨*names*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ ‘,’ ⟨*names*⟩ | ⟨*name*⟩ ⟨*names*⟩

⟨*name-list*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ ‘,’ ⟨*name-list*⟩

Definitions

⟨*definition*⟩ ::= ‘`define`’ ⟨*define*⟩
| ‘`component`’ ⟨*component*⟩
| ⟨*function*⟩

⟨*define*⟩ ::= ⟨*ast*⟩ ⟨*define-sig*⟩ ‘`=`’ ⟨*block*⟩

⟨*ast*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ ‘`>`’ ⟨*ast*⟩

⟨*component*⟩ ::= ⟨*name*⟩ ⟨*component-sig*⟩ ‘`{`’ ‘`value`’ ‘`=`’ ⟨*block*⟩ ‘`assign`’ ⟨*name*⟩ ‘`=`’ ⟨*block*⟩ ‘`}`’

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*name*⟩ ⟨*function-sig*⟩ ‘`=`’ ⟨*block*⟩

⟨*define-sig*⟩ ::= ⟨*single-constraint*⟩
| ‘`(`’ ‘`)`’
| ‘`(`’ ‘`)`’ ⟨*single-constraint*⟩
| ‘`(`’ ⟨*args*⟩ ‘`)`’
| ‘`(`’ ⟨*args*⟩ ‘`)`’ ⟨*single-constraint*⟩

⟨*function-sig*⟩ ::= ⟨*constraints*⟩
| ‘`(`’ ‘`)`’
| ‘`(`’ ‘`)`’ ⟨*constraints*⟩
| ‘`(`’ ⟨*args*⟩ ‘`)`’
| ‘`(`’ ⟨*args*⟩ ‘`)`’ ⟨*constraints*⟩

⟨*component-sig*⟩ ::= ⟨*type-constraint*⟩
| ‘`(`’ ‘`)`’ ⟨*type-constraint*⟩
| ‘`(`’ ⟨*args*⟩ ‘`)`’ ⟨*type-constraint*⟩

⟨*type-constraint*⟩ ::= ‘`::`’ ⟨*type*⟩
| ‘`::`’ ⟨*type*⟩ ⟨*constraints*⟩

⟨*args*⟩ ::= ⟨*a-name*⟩ | ⟨*a-name*⟩ ‘,’ ⟨*args*⟩

Blocks and Statements

⟨*block*⟩ ::= ⟨*statement*⟩
| ‘`{`’ ⟨*sequence*⟩ ‘`;`’ ⟨*return*⟩ ‘`}`’
| ⟨*return*⟩

⟨*statement*⟩ ::= ‘`{`’ ⟨*sequence*⟩ ‘`}`’ | ⟨*command*⟩

⟨*return*⟩ ::= ‘`return`’ ⟨*expression*⟩ | ⟨*expression*⟩

⟨*sequence*⟩ ::= ⟨*command*⟩ | ⟨*command*⟩ ‘`;`’ ⟨*sequence*⟩

⟨*command*⟩ ::= '#' ⟨*name*⟩
| '#' ⟨*name*⟩ ⟨*expression-literal*⟩
| ⟨*name*⟩ ⟨*expression-literal*⟩
| 'if' ⟨*expression*⟩ 'then' ⟨*block*⟩ 'else' ⟨*block*⟩
| 'when' ⟨*expression*⟩ 'do' ⟨*statement*⟩
| 'match' ⟨*expression*⟩ '{' ⟨*cases*⟩ '}'
| 'for' ⟨*name*⟩ 'in' ⟨*expression*⟩ '..' ⟨*expression*⟩ 'do' ⟨*statement*⟩
| 'foreach' ⟨*name*⟩ 'in' ⟨*expression*⟩ 'do' ⟨*statement*⟩
| 'nothing'
| 'var' ⟨*name*⟩
| 'var' ⟨*name*⟩ '=' ⟨*expression*⟩
| ⟨*reference*⟩ '<-' ⟨*expression*⟩
| ⟨*tuple*⟩ '=' ⟨*expression*⟩

⟨*tuple*⟩ ::= ⟨*tuple-literal*⟩ | ⟨*tuple-literal*⟩ ',' ⟨*tuple*⟩

⟨*tuple-literal*⟩ ::= ⟨*tuple-lit*⟩ | ⟨*tuple-lit*⟩ ⟨*annotation*⟩

⟨*tuple-lit*⟩ ::= ⟨*name*⟩ | '_' | '(' ⟨*tuple*⟩ ')'

⟨*cases*⟩ ::= ⟨*case*⟩ | ⟨*case*⟩ ⟨*cases*⟩

⟨*case*⟩ ::= 'case' ⟨*patterns*⟩ '=>' ⟨*block*⟩

⟨*patterns*⟩ ::= ⟨*pattern*⟩ | ⟨*pattern*⟩ 'or' ⟨*patterns*⟩

Expressions

⟨*expression*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*binary-op*⟩ ⟨*expression*⟩

⟨*expr*⟩ ::= ⟨*clause*⟩ | ⟨*clause*⟩ ⟨*annotation*⟩

⟨*binary-op*⟩ ::= ',' | 'or' | 'and' | '==' | '!=' | '<>' | '<' | '<+' | '<=' | '<=+' | '>' | '>+' | '>=' | '>=+' | 'in' | 'notin' | 'insert' | '@' | ':' | '-' | '+' | '||' | '??' | '*' | 'quot' | 'rem' | 'div' | 'mod' | '&&' | '<<' | '>>' | '>>+' | '#>>' | '#<<' | '^' | '**'

⟨*clause*⟩ ::= ⟨*unary-op*⟩ ⟨*expression*⟩
| ⟨*reference*⟩
| ⟨*expression-literal*⟩
| '#' ⟨*name*⟩
| '#' ⟨*name*⟩ ⟨*expression-literal*⟩
| 'if' ⟨*expression*⟩ 'then' ⟨*expression*⟩ 'else' ⟨*expression*⟩
| 'match' ⟨*expression*⟩ '{' ⟨*expression-cases*⟩ '}'
| 'set' '{' ⟨*expression*⟩ '}'
| 'list' '{' ⟨*expression*⟩ '}'

⟨*unary-op*⟩ ::= 'not' | '!' | '-' | '~'

⟨*expression-literal*⟩ ::= '(' ')'
| '(' ⟨*expression*⟩ ')'
| '[' ⟨*expression*⟩ ']'

| ‘'’ ⟨*binary*⟩ ‘'’
| ‘"’ ⟨*string*⟩ ‘"’
| ‘#"’ ⟨*character*⟩ ‘"’
| ⟨*numeric*⟩ | ‘`true`’ | ‘`false`’ | ‘`UNKNOWN`’

⟨*expression-cases*⟩ ::= ⟨*expression-case*⟩ | ⟨*expression-case*⟩ ⟨*expression-cases*⟩

⟨*expression-case*⟩ ::= ‘`case`’ ⟨*patterns*⟩ ‘`=>`’ ⟨*expression*⟩

⟨*reference*⟩ ::= ‘`&`’ ⟨*entity*⟩
| ‘`&`’ ⟨*entity*⟩ ⟨*index*⟩
| ⟨*dotted*⟩
| ⟨*dotted*⟩ ⟨*index*⟩
| ⟨*dotted*⟩ ‘`.`’ ‘`&`’ ⟨*entity*⟩
| ⟨*dotted*⟩ ‘`.`’ ‘`&`’ ⟨*entity*⟩ ⟨*index*⟩

⟨*dotted*⟩ ::= ⟨*entity*⟩ | ⟨*entity*⟩ ‘`.`’ ⟨*dotted*⟩

⟨*entity*⟩ ::= ⟨*name*⟩
| ⟨*name*⟩ ‘`(`’ ‘`)`’
| ⟨*name*⟩ ‘`(`’ ⟨*expression*⟩ ‘`)`’

⟨*index*⟩ ::= ‘`<`’ ⟨*expression*⟩ ‘`>`’

Patterns

⟨*pattern*⟩ ::= ⟨*pat*⟩ | ⟨*pat*⟩ ‘`,`’ ⟨*pattern*⟩ | ⟨*pat*⟩ ‘`@`’ ⟨*pattern*⟩

⟨*pat*⟩ ::= ⟨*pattern-literal*⟩ | ⟨*pattern-literal*⟩ ⟨*annotation*⟩

⟨*pattern-literal*⟩ ::= ⟨*name*⟩ ‘`(`’ ‘`)`’
| ⟨*name*⟩ ‘`(`’ ⟨*pattern*⟩ ‘`)`’
| ⟨*pattern-lit*⟩
| ⟨*bit-pattern*⟩
| ‘`-`’ ⟨*number*⟩
| ‘`(`’ ⟨*pattern*⟩ ‘`)`’

⟨*pattern-lit*⟩ ::= ⟨*name*⟩ | ⟨*number*⟩ | ‘`_`’ | ‘`true`’ | ‘`false`’ | ‘`UNKNOWN`’

Bit patterns

⟨*bit-pattern*⟩ ::= ‘'’ ⟨*bit-pat*⟩ ‘'’

⟨*bit-pat*⟩ ::= ⟨*bit-pattern-lit*⟩
| ⟨*bit-pattern-lit*⟩ ⟨*bit-pat*⟩
| ⟨*bit-pattern-lit*⟩ ‘`:`’ ⟨*bit-pat*⟩

⟨*bit-pattern-lit*⟩ ::= ⟨*pattern-lit*⟩
| ⟨*pattern-lit*⟩ ‘`` ` ``’ ⟨*bits-type*⟩
| ‘`(`’ ⟨*binary*⟩ ‘`)`’

Constraints

⟨*constraints*⟩ ::= '`with`' ⟨*constraint-list*⟩

⟨*single-constraint*⟩ ::= '`with`' ⟨*name*⟩ '`in`' ⟨*number-list*⟩

⟨*constraint-list*⟩ ::= ⟨*constraint*⟩ | ⟨*constraint*⟩ '`and`' ⟨*constraint-list*⟩

⟨*constraint*⟩ ::= ⟨*name*⟩ '`in`' ⟨*number-set*⟩ | ⟨*name*⟩ ⟨*ordering*⟩ ⟨*number*⟩

⟨*ordering*⟩ ::= '`<`' | '`>`' | '`<=`' | '`>=`'

⟨*number-list*⟩ ::= ⟨*number*⟩ | ⟨*number*⟩ '`,`' ⟨*number-list*⟩

⟨*number-set*⟩ ::= ⟨*range*⟩ | ⟨*range*⟩ '`,`' ⟨*number-set*⟩

⟨*range*⟩ ::= ⟨*number*⟩ | ⟨*number*⟩ '`-`' ⟨*number*⟩ | ⟨*number*⟩ '`...`'

Register specifications

⟨*fields*⟩ ::= ⟨*field*⟩ | ⟨*field*⟩ ⟨*fields*⟩ | ⟨*field*⟩ '`,`' ⟨*fields*⟩

⟨*field*⟩ ::= ⟨*bit-ranges*⟩ '`:`' ⟨*field-name*⟩

⟨*bit-ranges*⟩ ::= ⟨*bit-range*⟩ | ⟨*bit-range*⟩ '`,`' ⟨*bit-ranges*⟩

⟨*bit-range*⟩ ::= ⟨*number*⟩ | ⟨*number*⟩ '`-`' ⟨*number*⟩

⟨*field-name*⟩ ::= '`UNK!`' | '`RAZ!`' | '`RAO!`' | ⟨*name*⟩