The Sail instruction-set semantics specification language

Kathryn E. Gray, Peter Sewell, Christopher Pulte, Shaked Flur, Robert Norton-Wright

March 15, 2017

Contents

1 Introduction 2
2 A tutorial example 4
3 Running Sail 8
4 The Sail language: types 9
  4.1 Basic types and the corresponding expression forms 9
  4.1.1 The type grammar 14
  4.2 Dependent types for vector lengths, start indices, and integer ranges 15
  4.2.1 Types with Nat constants 15
  4.2.2 Types with Nat variables and Nat-polymorphic functions 15
  4.2.3 Types with Nat arithmetic 16
  4.2.4 Types with constrained quantification of Nat variables 16
  4.2.5 Type definitions with quantified Nat variables 16
  4.2.6 Implicit Nat arguments 17
  4.3 Order polymorphism 17
  4.4 Parametric type polymorphism 18
  4.5 Effects 18
  4.6 Kinds, type constructors, and type applications 19
  4.7 Type schemes 19
  4.8 Flow-sensitive constraints 20
  4.9 Implicit coercions 20
5 The Sail language: expressions 20
  5.1 Pattern matching 20
  5.2 Control flow 21
  5.3 Assignment and l-values 22
  5.4 Undefined values 23
  5.5 Undefined behaviour 24
  5.6 The full expression grammar 24
6 The Sail language: files and top-level definitions 25
  6.1 Nat abbreviations 25
  6.2 Type definitions 25
  6.3 Function definitions 27
  6.4 Value definitions 27
  6.5 Type constraints 27
  6.6 Default kind and type assumptions 29
  6.7 Scattered definitions 29
  6.8 Register declarations 29
1 Introduction

Sail is a language for expressing the instruction-set architecture (ISA) semantics of processors. Vendor architecture specification documents typically describe the sequential behaviour of their ISA with a combination of prose, tables, and pseudocode for each instruction. They vary in how precise that pseudocode is: in some it is just suggestive, while in others it is close to a complete description of the envelope of architecturally allowed behaviour for sequential code. For x86 [1], the Intel pseudocode is just suggestive, with embedded prose, while the AMD descriptions [2] are prose alone. For IBM Power [3], there is reasonably detailed pseudocode for many instructions in the manual, but it has never been machine-parsed. For ARM [4], there is detailed pseudocode, which has recently become machine-processed [5]. For MIPS [6, 7] there is also reasonably detailed pseudocode.

The behaviour of concurrent code is often described less well. In a line of research from 2007–2017 we have developed mathematically rigorous models for the allowed architectural envelope of concurrent code, for x86, IBM Power, and ARM, that have been reasonably well validated by experimental testing and by discussion with the vendors and others [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. In the course of this, we have identified a number of subtle issues relating to the interface between the intra-instruction semantics and the inter-instruction concurrency semantics [13, 14, 15, 16, 17]. For example, the concurrency models, rather than treating each instruction execution as an atomic unit, require exposed register and memory events, knowledge of the potential register and memory footprints of instructions, and knowledge of changes to those during execution. Our early work in this area had hand-coded instruction semantics for quite small fragments of the instruction sets, just enough for concurrency litmus tests and expressed in various ad hoc ways. As our models have matured, we have switched to modelling the intra-instruction semantics more completely and in a style closer to the vendor-documentation pseudocode, and Sail was developed for this.

Sail is intended:
• to support precise definition of real-world ISA semantics;
• to be accessible to engineers familiar with existing vendor pseudocodes, with a similar style to the pseudocodes used by ARM and IBM Power (modulo minor syntactic differences);
• to expose the structure needed to combine the sequential ISA semantics with the relaxed-memory concurrency models we have developed;
• to provide an expressive type system that can statically check the bitvector length and indexing computation that arises in these specifications, to detect errors and to support code generation, with type inference to minimise the required type annotations;
• to support execution, for architecturally complete emulation automatically based on the definition;
• to support automatic generation of theorem-prover definitions, for mechanised reasoning about ISA specifications; and
• to be as minimal as possible given the above, to ease the tasks of code generation and theorem-prover definition generation.

A Sail definition will typically define an abstract syntax type (AST) of machine instructions, a decode function that takes binary values to AST values, and an execute function that describes how each of those behaves at runtime, together with whatever auxiliary functions and types are needed. Given such a definition, the Sail implementation can typecheck it and generate:

• an internal representation of the fully type-annotated definition (a deep embedding of the definition) in a form that can be executed by the Sail interpreter. These are both expressed in Lem [18, 19], a language of type, function, and relation definitions that can be compiled into OCaml and various theorem provers. The Sail interpreter can also be used to analyse instruction definitions (or partially executed instructions) to determine their potential register and memory footprints.

• a shallow embedding of the definition, also in Lem, that can be executed or converted to theorem-prover code more directly. Currently this is aimed at Isabelle/HOL or HOL4, though the Sail dependent types should support generation of idiomatic Coq definitions (directly rather than via Lem).

Sail does not currently support description of the assembly syntax of instructions, or the mapping between that and instruction AST or binary descriptions.

Sail has been used to develop models of parts of several architectures:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv8 (hand)</td>
<td>hand-written</td>
</tr>
<tr>
<td>ARMv8 (ASL)</td>
<td>automatically derived from ARM-internal ASL spec</td>
</tr>
<tr>
<td>IBM Power</td>
<td>extracted/patched from IBM Framemaker XML</td>
</tr>
<tr>
<td>MIPS</td>
<td>hand-written</td>
</tr>
<tr>
<td>CHERI</td>
<td>hand-written</td>
</tr>
</tbody>
</table>

The ARMv8 (hand) and IBM Power models cover all user-mode instructions except vector, float, and load-multiple instructions, without exceptions; for ARMv8 this is for the A64 fragment. The ARMv8 (hand) model is hand-written based on the ARMv8-A specification document [4, 16], principally by Flur. The Power model is based on an automatic extraction of pseudocode and decoding data from an XML export of the Framemaker document source of the IBM Power manual [3, 15], with manual patching as necessary, principally by Kerneis and Gray. The ARMv8 (ASL) model is based on an automatic translation of an ARM-internal definition of their pseudocode. Currently it covers a similar fragment. The MIPS model is hand-written based on the MIPS64 manual version 2.5 [6, 7], but covering only the features in the BERI hardware reference [20], which in turn drew on MIPS4000 and MIPS32 [21, 22]. The CHERI model is based on that and the CHERI ISA reference manual version 5 [23]. These two are both principally by Norton-Wright; they cover all basic user and kernel mode MIPS features sufficient to boot FreeBSD, including a TLB, exceptions and a basic UART for console interaction. ISA extensions such as floating point are not covered. The CHERI model supports either 256-bit capabilities or 128-bit compressed capabilities.
Fig. 1 shows a side-by-side comparison of vendor-documentation and Sail instruction descriptions, for sample IBM Power and ARM instructions, to demonstrate the readability of the Sail versions for those familiar with the vendor-documentation versions. For the Power `stdu` instruction:

- decoding is specified in Sail with a clause of the `decode` function using a bitvector concatenation pattern, rather than the diagram of the manual.

- the Sail gives an explicit AST tagged-union clause `Stdu` containing three bitfields for the opcode parameters. They are not named in the type, but the pattern matches of the `execute` and `invalid` function clauses use consistent names as in the manual.

- general-purpose register accesses in the Sail involve an explicit indexing into the array `GPR` of registers, e.g. `GPR[R郜]`, rather than a mention of the register number `RA` (as an lvalue) or a parenthesised `(RA)` (for its value) in the manual.

- the local variable `EA`, used to calculate the effective address, has an explicit type annotation and initialisation in the Sail. Other idioms are also possible: the initialisation could have been omitted, putting the type annotation on the line below, or `EA` could have been given a default type.

- in the Sail, the memory write is split into two actions, one to announce the effective address and one to specify the value. It is important in the concurrency model for the former to be before the register reads (here `GPR[RS]`) required for the latter.

- in the Sail, the writeback of register `GPR[RA]` with the calculated effective address is before the memory write; this is also important for the concurrency model.

- the identification of the invalid `RA==0` case is in prose in the manual but a clause of another function, `invalid`, in the Sail.

The ARM `ADD (immediate)` instruction execution function body is very similar in the manual and in Sail, modulo minor syntactic differences. The main difference is that the sizes of bitvectors are specified by a manifestly compile-time variable `’R` in the Sail.

Sail is available on bitbucket (https://bitbucket.org/Peter_Sewell/sail/) under a BSD licence. The ARMv8 (hand), MIPS, and CHERI models are also available there. Sail has been principally designed and implemented by Gray, with contributions from Flur, Kell, Kerneis, Norton-Wright, Pulte, and Sewell.

We note various possible Sail changes and defects in footnotes.

2 A tutorial example

We introduce Sail with a small example: Fig. 2 shows an extract of our Power ISA model. This is a self-contained type-correct Sail file, modulo the elided register declarations on Line 4.

The file starts by declaring general-purpose registers `GPR0..GPR31`, each of which holds a value of type `bit[64]`, i.e. a bitvector of length 64.

On Lines 7–10 it declares a vector `GPR` of the names of these registers. The vector has length 32, indexed in increasing order from left to right, starting at 0.

Lines 12–15 declare three primitive functions, `MEMr`, `MEMw_EA`, and `MEMw`, used to read and write memory in this specification. These functions are external as far as Sail is concerned; calls to them are converted to actual memory accesses by whatever whole-system-semantics harness is used around the Sail instruction semantics. In a sequential model, memory accesses can be modelled with simple accesses to a byte-array view of memory; in a realistic concurrency model, they have to be more involved. Each of these primitive functions is polymorphic in a size `’n`; `MEMw_EA` requires this to be in the set `{1,2,4,8,16}`, whereas the other two leave it unconstrained as any natural number. Then each has a type:

- `MEMr`, used for memory reads, has a function type `(bit[64], [:’n:]) -> (bit[8 * ’n])`, taking a pair of the address and the number of bytes to read (a pair of a bitvector of size 64 and an integer which must be exactly `’n`), and returning the bytes read (a bitvector of size 8 * `’n`).
IBM Power Store Doubleword with Update (stdu) Instruction
Power 2.06 manual

<table>
<thead>
<tr>
<th>Store Doubleword with Update</th>
<th>DS-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdu RS,DS(RA)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>62</th>
<th>6</th>
<th>11</th>
<th>16</th>
<th>30 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RS</td>
<td>RA</td>
<td>DS</td>
<td>1</td>
</tr>
</tbody>
</table>

EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA

Let the effective address (EA) be the sum (RA)+(DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.
If RA=0, the instruction form is invalid.

Special Registers Altered:
None

ARM ADD (immediate) (ADD) Instruction
ARMv8-A manual

bits(datasize) result;
bits(datasize) operand1 =
  if n == 31 then SP[] else X[n];
bits(datasize) operand2 = imm;
bits(4) nzcv;
bit carry_in;

if sub_op then
  operand2 = NOT(operand2);
carry_in = '1';
else
  carry_in = '0';

(result, nzcv) = AddWithCarry(operand1,operand2,carry_in);
if setflags then
  PSTATE.<N,Z,C,V> = nzcv;

if d == 31 && !setflags then
  SP[] = result;
else
  X[d] = result;

Figure 1: Comparison of vendor documentation and Sail instruction descriptions. For the Power stdu instruction we show the AST clause, decode, and execute functions; for the ARM ADD (immediate) instruction we show just the body of the execution functions.
(* Fixed-point registers *)

register (bit[64]) GPR0
register (bit[64]) GPR1
(* ... *)
register (bit[64]) GPR31

let (vector <0, 32, inc, (register<(bit[64])>) >) GPR =
[ GPR0, GPR1, GPR2, GPR3, GPR4, GPR5, GPR6, GPR7, GPR8, GPR9, GPR10,
 GPR11, GPR12, GPR13, GPR14, GPR15, GPR16, GPR17, GPR18, GPR19, GPR20,
 GPR21, GPR22, GPR23, GPR24, GPR25, GPR26, GPR27, GPR28, GPR29, GPR30, GPR31 ]

(* primitive external functions to read and write memory *)
val extern forall Nat 'n. (bit[64], [:'n:]) -> (bit[8 * 'n]) effect {rmem} MEMr
val extern forall Nat 'n, 'n IN {1,2,4,8,16}. (bit[64], [:'n:]) -> unit effect {eamem} MEMw.EA
val extern forall Nat 'n. (bit[64], [:'n:], bit[8*'n]) -> unit effect {wmv} MEMw

scattered typedef ast = const union
val bit[32] -> option<ast> effect pure decode
scattered function option<ast> decode
class scattered function unit execute

(* add immediate instruction *)
union ast member (bit[5], bit[5], bit[16]) Addi

function clause decode
(0b001110 :
 (bit[5]) RT :
 (bit[5]) RA :
 (bit[16]) SI as instr) =
Some(Addi(RT,RA,SI))

function clause execute (Addi (RT, RA, SI)) =
  if RA == 0 then GPR[RT] := EXTS(SI)
  else GPR[RT] := GPR[RA] + EXTS(SI)

(* load halfword zero instruction *)
union ast member (bit[5], bit[5], bit[16]) Lhz

function clause decode
(0b101000 :
 (bit[5]) RT :
 (bit[5]) RA :
 (bit[16]) D as instr) =
Some(Lhz(RT,RA,D))

function clause execute (Lhz (RT, RA, D)) = {
  (bit[64]) b := 0;
  (bit[64]) EA := 0;
  if RA == 0 then b := 0 else b := GPR[RA];
  EA := b + EXTS(D);
  GPR[RT] := 0b000000000000000000000000000000000000000000000000 : MEMr(EA,2) }

(* decode failure clause *)
function clause decode _ = None

end execute
end decode
end ast
• MEMw.EA, used to announce the effective address of an upcoming memory write, has a function type  
(bit[64], [:{'n':1}] -> unit, taking a pair of the address and the number of bytes to write (a pair of a  
bitvector of size 64 and an integer that must be exactly ‘n) and returning the unit value. In ARM and  
Power concurrency models, it can be programmer-observable that a memory write address becomes  
determined before the value of the write is available, so the interface between the ISA semantics and  
concurrency semantics has to make this explicit.

• MEMw, used to actually perform a memory write, has a type (bit[64], [:{'n':1}, bit[8*‘n’]) -> unit,  
taking a tuple of the address, the number of bytes to write, and the data to write (a tuple of a  
bitvector of size 64, an integer which must be exactly ‘n, and a bitvector of length 8*‘n). It too returns the unit  
value. Each is also labelled with a primitive effect, rmem, eamem, or wmw. Sail infers the set of primitive effects of all  
functions, letting one easily see, for example, whether a function is pure.

The remainder of the file, Lines 17–57, defines:

• the ast type: a union type of the machine instructions;

• the decode function: a function taking bitvectors of size 32 to option<ast> values, which are either  
Some(v) for an ast value v, if decoding is succesful, or None, if not; and

• the execute function: a function taking an ast value to its behaviour.

These three are all scattered definitions, meaning that their clauses can be interleaved, with each other and  
with other definitions. Sometimes this is useful, as in this file, to group the different parts of an instruction  
definition together. Scattered declarations are introduced as on Lines 17–20, with an optional val specification  
that also gives the result type and effect, as on Line 18; they are terminated as on Lines 55–57. Semantically,  
scattered definitions of types appear at the start of their definition, and scattered definitions of functions  
appear at the end. As far as Sail is concerned, there is nothing special about the names ast, execute, and  
decode, but our whole-system harnesses around the Sail instruction semantics do assume that definitions  
with these names and types are present.

The body of these definitions defines two Power instructions: add immediate addi RT,RA,SI (Lines 22-33)  
and load halfword and zero lhz RT,D(RA) (Lines 35-50).

Each begins by declaring a clause of the ast union type to represent the instruction (Lines 23 and 36).  
Each of these has a constructor (essentially a union tag), Addi and Lhz, and a tuple of arguments. In this  
specification those are the bitvector opcode fields of the instruction, eg the (bit[5], bit[5], bit[16]) of  
Addi; in other specifications (eg. ARM), the arguments are sometimes higher-level computed quantities —  
this is a matter of style, not forced by Sail. One could also define two types and the mapping between them  
within Sail. The opcode fields are not explicitly named in Sail, though we do use consistent naming of pattern  
variables in the functions that act on the ast type, eg the RT, RA, and SI in the decode and execute clauses  
for Addi.

Looking at the decode clause for Addi, Lines 25–30, this pattern-matches its 32-bit bitvector argument  
against a vector concatenation pattern

(0b001110 : (bit[5]) RT : (bit[5]) RA : (bit[16]) SI)

which starts with a constant bitvector 0b001110 and then has three variables of bitvector type, of sizes 5, 5,  
and 16 respectively. If the input 32-bit bitvector does match this, the clause constructs the Addi(RT,RA,SI)  
value of the ast type, and wraps that in the Some(...) constructor of the option<ast> type to indicate success.  
If the input doesn’t match, Sail will try the next clause of decode. If none match, the default clause (with a  
wildcard pattern .) on Lines 52–53 will match, and decode will return the None constructor of the option<ast>  
type to indicate failure.

Turning to the execute clause for Addi, Lines 32–33, it takes a value Addi (RT, RA, SI) of the ast type,  
binds variables RT, RA, and SI to the opcode arguments, and executes the body of the clause on Line 33.  
The body is more-or-less standard imperative code, with a conditional test on RA, reads and writes of GPR  
registers indexed by RT and RA, and sign-extension and addition using Sail built-in functions EXT15 and +.  
These have flexible definitions which we return to later; the Sail type inference and overloading resolution  
chooses appropriate instances. Analogous to the usual behaviour of C variables, a register name, such as the  

value of \texttt{GPR[RT]} (the RT element of the vector of register names \texttt{GPR}) can appear on the left of an assignment, as an lvalue, or within an expression, in which case the register is implicitly dereferenced to the value of the contents of that register name.

The \texttt{execute} clause for \texttt{Lhz}, Lines 45–50, is broadly similar. It introduces two local variables, \texttt{b} and \texttt{EA}, with their types and initial values; local variables can similarly be assigned to and dereferenced. This function body also calls the \texttt{MEMr} external function, with \texttt{MEMr(EA,2)}, to do a memory read of 2 bytes from the effective address computed in \texttt{EA}. The result is concatenated with a 48-bit zero constant (which could have been written more compactly, in several ways).

3 Running Sail

Sail is a command-line tool, analogous to a compiler: one gives it a list of input Sail files; it type-checks those and generates translated output. There are three main options. Given the Fig. 2 \texttt{power_fragment.sail}:

1. running \texttt{sail power_fragment.sail -lem_ast} will generate a Lem file containing a deep embedding of the source: Lem representation of its fully type-annotated abstract syntax tree, in a form that can be executed by the Sail interpreter:

\begin{verbatim}
power_fragment.lem
\end{verbatim}

2. running \texttt{sail -lem -lem_lib Power_extras_embed power_fragment.sail} will generate a Lem shallow embedding of the source:

\begin{verbatim}
power_fragment_embed.lem
power_fragment_embed_sequential.lem
power_fragment_embed_types.lem
\end{verbatim}

3. running \texttt{sail power_fragment.sail -ocaml} will generate an OCaml shallow embedding of the source:

\begin{verbatim}
power_fragment.ml
\end{verbatim}

Typically one would use these by linking them into a build of our \texttt{ppcmem} tool, for exploring concurrent executions, or one of the \texttt{sail/src/lem_interp run_with_elf*} tools, for standalone sequential execution.

The full command-line options are:

\texttt{Sail}

\texttt{usage: sail <options> <file1.sail> .. <fileN.sail>}

\begin{verbatim}
-o <prefix> select output filename prefix
-lem_ast output a Lem AST representation of the input
-lem output a Lem translated version of the input
-ocaml output an OCaml translated version of the input
-lem_lib <filename> provide additional library to open in Lem output
-ocaml_lib <filename> provide additional library to open in OCaml output
-print_initial_env print the built-in initial type environment and terminate
-verbose (debug) pretty-print the input to standard output
-skip_constraints (debug) skip constraint resolution in type-checking
-v print version
-help Display this list of options
--help Display this list of options
\end{verbatim}

The \texttt{-verbose} option pretty prints the sources after coercion insertion, implicit resolution, and some but not all of the rewriting. The output is not exactly well-formed Sail source, for minor reasons:

- implicit arguments have been inserted but function declarations still have them implicit
- there are special casts ((\texttt{length})(\ldots)) which tell subsequent Sail passes to re-index decreasing vectors to start from \texttt{length} – 1. Many of these are are inserted, but then after constraint solving it becomes obvious that many aren’t needed, and we take them back out.
4 The Sail language: types

We describe Sail types and the related main definition and expression forms in this section; the Sail expression language in §5, and Sail top-level definitions in §6. The standard library is given in §7. The full grammar is in Appendix A.

Sail has an expressive static type system, to check consistency and detect errors in Sail definitions at type-checking time, and to provide type information for the generation of emulator code and theorem-prover definitions, while supporting the idioms that are used in the existing vendor pseudocode, and without requiring excessive type annotation or partiality.

The basic types (§4.1) are unit, bits, booleans, integer ranges, vectors, tuples, lists, option types, strings, mutable registers, mutable local variables, user-defined record, tagged-union, and enumeration types, and first-order top-level functions.

This is integrated with lightweight dependent types (§4.2): ISA descriptions typically elaborate manipulation of bitvectors, with computed indices and lengths, and registers are indexed from various start-index values. To check these we use a type system in which types can be dependent on simple arithmetic expressions, and in which functions can be polymorphic on integer quantities.

The type system also supports parametric polymorphism (§4.4), and indexing-order polymorphism (§4.3). Sail function types can be annotated by sets of effects (§4.5), to identify whether they are pure or can have register or memory side-effects, and there is also simple effect polymorphism. To do all this uniformly, types are classified by kinds (§4.6), in the same way that expressions are classified by types. We also collect the full grammar of type schemes, which are types together with quantified variables of various kinds and constraints on Nat-kindged variables (§4.7). Constraints can be flow-sensitive (§4.8).

Sail also provides implicit type coercions (§4.9) between numbers, vectors, individual bits, booleans, and registers, again keeping the specification readable.

4.1 Basic types and the corresponding expression forms

The basic types are as follows. We give the corresponding expression forms for constructing and deconstructing values of these types, sometimes with excerpts of the Sail grammar.

- unit unit: a type with a single value (). Sail does not syntactically distinguish between expressions and statements: a statement is just an expression of type unit.

- bits bit. This has values bitzero and bitone, but one usually writes constants of bit vector or integer range types instead.\(^1\)

- booleans bool, with values true and false. Currently this is equivalent to bit, and those are equivalent to bitzero and bitone.\(^2\)

- integer ranges range<\(nexp_1, nexp_2\)>: the type of the range of integers from \(nexp_1\) to \(nexp_2\) inclusive. Sail supports bounded integer ranges so that type-checking can guarantee that vector indexing and assignment to bitvectors are within bounds. The numeric expressions \(nexp_1\) and \(nexp_2\) can be constants or from a moderately expressive grammar; we return to this below (§4.2). There are several built-in abbreviations for particular ranges:
  - nat: the natural numbers, 0,…
  - int: all the integers
  - uint8: the range range<\(0, 2^8 - 1\)>
  - uint16: the range range<\(0, 2^{16} - 1\)>
  - uint32: the range range<\(0, 2^{32} - 1\)>
  - uint64: the range range<\(0, 2^{64} - 1\)>
  - [\(\lfloor nexp_1 : nexp_2 \rfloor\)]: the range range<\(nexp_1, nexp_2\)>

\(^1\)Perhaps syntax \#0 and \#1 would be preferable.
\(^2\)We may revisit this decision, to match the practice of the ARM ARM.
Vectors can be constructed and accessed in several ways:

\[
\text{exp} ::= \ldots \\
| \quad [ \text{exp}_1, \ldots, \text{exp}_n ] \quad \text{vector (indexed from 0)} \\
| \quad [ \text{num}_1 = \text{exp}_1, \ldots, \text{num}_n = \text{exp}_n \text{ opt } \text{ default } ] \quad \text{vector (indexed consecutively)} \\
| \quad \text{exp} [ \text{exp}' ] \quad \text{vector access} \\
| \quad \text{exp} [ \text{exp}_1 \ldots \text{exp}_2 ] \quad \text{subvector extraction} \\
| \quad [ \text{exp with exp}_1 = \text{exp}_2 ] \quad \text{vector functional update} \\
| \quad [ \text{exp with exp}_1 : \text{exp}_2 = \text{exp}_3 ] \quad \text{vector subrange update (with vector)} \\
| \quad \text{exp : exp}_2 \quad \text{vector concatenation}
\]

They can also be deconstructed by pattern matching:

\[
\text{pat} ::= \ldots \\
| \quad [ \text{pat}_1, \ldots, \text{pat}_n ] \quad \text{vector pattern} \\
| \quad [ \text{num}_1 = \text{pat}_1, \ldots, \text{num}_n = \text{pat}_n ] \quad \text{vector pattern (with explicit indices)} \\
| \quad \text{pat}_1 : \ldots : \text{pat}_n \quad \text{concatenated vector pattern}
\]

- tuples \( (\text{typ}_1, \ldots, \text{typ}_n) \): type of tuples of values, of types \( \text{typ}_1, \ldots, \text{typ}_n \). Tuple expressions are:

\[
\text{exp} ::= \ldots \\
| \quad ( \text{exp}_1, \ldots, \text{exp}_n ) \quad \text{tuple}
\]

---

1. If we were doing this again, we might prefer a uniform internal increasing-from-zero representation for values, using the additional start-index and direction information just in the user interface.

2. Vector functional update and subrange update are supported by type system, interpreter, and shallow embedding, but have not been used (or tested) so far.
and they can be deconstructed by pattern matching:

\[
\begin{align*}
\text{pat} & ::= \ldots & \text{Pattern} \\
& | (\pat_1, \ldots, \pat_n) & \text{tuple pattern}
\end{align*}
\]

- lists \text{list<typ>}: type of lists of elements of \text{typ}.\footnote{Lists are not used in our main specifications, but are used in the handwritten analysis of instruction footprints.} They can be constructed with:

\[
\begin{align*}
\text{exp} & ::= \ldots & \text{Expression} \\
& | [\mid \exp_1, \ldots, \exp_n \mid] & \text{list} \\
& | \exp_1 :: \exp_2 & \text{cons}
\end{align*}
\]

and deconstructed by pattern matching:

\[
\begin{align*}
\text{pat} & ::= \ldots & \text{Pattern} \\
& | [\mid \pat_1, \ldots, \pat_n \mid] & \text{list pattern}
\end{align*}
\]

- options \text{option<typ>}: type of values that are are either \text{Some(v)} for some value \text{v} of type \text{typ}, or \text{None}. This is often used for the return type of functions that may fail.

- strings \text{string}. Strings in Sail are typically only used in error-reporting. They are lexed using Lem’s string lexer and so support the same escape sequences as Lem.

- user-defined record types. In the context of a top-level type definition

\[
\begin{align*}
\text{typedef } \text{id} = \text{const struct} \{ \text{typ}_1 \text{id}_1; \ldots; \text{typ}_n \text{id}_n \}
\end{align*}
\]

the type identifier \text{id} represents a type of immutable structs (or records) with members \text{id}_1..\text{id}_n of types \text{typ}_1..\text{typ}_n. Member names can be reused in different struct types, but explicit type annotations may be needed if they are. Records can be constructed with expressions \{ \text{id}_1 = \exp_1; \ldots; \text{id}_n = \exp_n \}. Their members can be accessed with \text{exp}.\text{id}, and there is a functional update operation that constructs a new record with some members modified:\footnote{Functional update of structs are used in ASL.} \{ \text{exp with id}_1 = \exp_1; \ldots; \text{id}_k = \exp_k \}. They can be deconstructed by pattern matching:

\[
\begin{align*}
\text{pat} & ::= \ldots & \text{Pattern} \\
& | \{ f\pat_1; \ldots; f\pat_n \} & \text{struct pattern} \\
\text{fpat} & ::= \ldots & \text{field pattern} \\
& | \text{id} = \pat & \text{field pattern}
\end{align*}
\]

- user-defined tagged union types. In the context of a top-level type definition

\[
\begin{align*}
\text{typedef } \text{id} = \text{const union} \{ \text{typ}_1 \text{id}_1; \ldots; \text{typ}_n \text{id}_n \}
\end{align*}
\]

the type identifier \text{id} represents a type of immutable tagged unions, with members \text{id}_1..\text{id}_n of types \text{typ}_1..\text{typ}_n. Some of the \text{typ}_i can be omitted in the declaration, in which case the corresponding member is taken to be of type \text{unit}. Union values can be constructed by applying the member name to an expression of the appropriate type (which can be omitted if \text{unit}); they can be deconstructed with pattern matching.

\[
\begin{align*}
\text{pat} & ::= \ldots & \text{Pattern} \\
& | \text{id} (\pat_1, \ldots, \pat_n) & \text{union constructor pattern}
\end{align*}
\]
• user-defined enumeration types. In the context of a top-level type definition

\[
\text{typedef id = enumerate \{ id_1; \ldots; id_n \}}
\]

the type identifier id represents an enumeration type, with members id_1..id_n, corresponding to integers 0, 1, \ldots.

• mutable registers register<typ>. Registers are declared with a top-level

\[
\text{register typ id}
\]

after which id has type register<typ>. If id is mentioned in an expression, then it evaluates either to the register name (of that type) or the value contained in the register (of type typ), depending on the expected type provided by the context. If id appears on the left-hand-side of an assignment, e.g. in id := exp, the register name is used as an lvalue. More complex lvalues are also supported; see below (§5.3).

This is unusual because we want to combine (a) C-style implicit dereferencing and (b) use of register names as values, primarily in top-level non-function declarations, constructing aliases and vectors of registers. There are a few functions in the ARM (hand) model and (we think) in the ARMv8 (ASL) model to pass registers around, for abstraction, e.g. to pull out parts of the current exception level uniformly. For accessing register banks, we pass around an index, not a register name.\footnote{In hindsight, it might have been better to have special syntax to identify the register-name-as-value case — that would have simplified coercions and might have meant that we could have a symmetric type equality relation.}

The type typ should always be a bitvector type\footnote{This is assumed by our concurrency models; it may not be but should be enforced by Sail. It rules out declarations such as register (vector<0,32, inc, bit[64]>) GPR of a bank of registers in one go, though one could define register (vector<0,32+64,inc,bint>).}; and register declarations are only permitted at top-level, not within functions. All register types used in a specification should have the same order\footnote{The code assumes this, but it is not currently checked by the Sail type checker; it should be.}. Sail also supports a special type definition form to introduce bitvector types with named subvectors:

\[
type\_def ::= \ldots \quad \text{Type definition body}
| \quad \text{typedef id = register bits [ nexp : nexp' ] \{ index\_range_1 : id_1 ; \ldots ; index\_range_n : id_n \}}
\]

\[
index\_range ::= \quad | \quad \text{num}
| \quad \text{num}_1..\text{num}_2
| \quad \text{index\_range}_1, \text{index\_range}_2
\]

For example, the Power model contains

\[
\text{typedef cr = register bits [ 32 : 63 ] \{ }\n\quad 32..35 : \text{CR0};
\quad 32 : \text{LT}; 33 : \text{GT}; 34 : \text{EQ}; 35 : \text{SO};
\quad 36..39 : \text{CR1};
\quad 36 : \text{FX}; 37 : \text{FEX}; 38 : \text{VX}; 39 : \text{OX};
\quad 40..43 : \text{CR2};
\quad 44..47 : \text{CR3};
\quad 48..51 : \text{CR4};
\quad 52..55 : \text{CR5};
\quad 56..59 : \text{CR6};
\quad 60..63 : \text{CR7};
\quad \}\n\]

\[
\text{register (cr) CR}
\]

after which cr is a type abbreviation for a 32-bit bitvector type and CR is a register of that type, but one can refer to its fields with dot notation, e.g. CR.CR2 (both in an expression and in an lvalue).

One can also declare aliases for register fields, e.g.
register alias PSTATE_N = NZCV.N

which essentially declares a macro for NZCV.N, that can be used on left of := or in expressions, and for concatenations of registers, e.g.

register alias FPRp0 = FPR0 : FPR1

though the latter is not well tested.\textsuperscript{10}

The general forms of register and register alias declarations are:

\begin{align*}
\textit{alias\_spec} & ::= \quad \text{register alias expression forms} \\
& \mid \textit{reg\_id} . \textit{id} \\
& \mid \textit{reg\_id} [ \textit{exp} ] \\
& \mid \textit{reg\_id} [ \textit{exp} \ldots \textit{exp}' ] \\
& \mid \textit{reg\_id} : \textit{reg\_id}'
\end{align*}

\begin{align*}
\textit{dec\_spec} & ::= \quad \text{register declarations} \\
& \mid \textit{register typ \textit{id}} \\
& \mid \textit{register alias \textit{id} = alias\_spec} \\
& \mid \textit{register alias typ \textit{id} = alias\_spec}
\end{align*}

Each \textit{reg\_id} in an alias specification must refer to an unaliased register of a vector type.

- local mutable variables \texttt{reg<typ>}. ISA descriptions often use local mutable variables, which have these types in Sail. They are distinct from the above \texttt{register} types, but both allow imperative update and implicit dereferencing. To support common practice in the vendor documentation ISA descriptions, local variables can be introduced just with an assignment, e.g.

\begin{align*}
x & := \theta
\end{align*}

or with a type declaration at the same point, e.g.

\begin{align*}
(\texttt{int}) \ x & := \theta
\end{align*}

After either of these, \textit{x} has type \texttt{reg<int>}. Local variables cannot be declared without an initial expression.\textsuperscript{11} Permitting local variables to be introduced without a type does lead to potential confusion. Consider

\begin{align*}
(\texttt{register <\texttt{bit}[64]>}) a & = \texttt{GPR}[d] \quad \text{stores the register name \texttt{GPR}[d] in \textit{a}} \\
(\texttt{\texttt{bit}[64]}) \ b & = \texttt{GPR}[d] \quad \text{stores the contents of \texttt{GPR}[d] in \textit{b}} \\
c & = \texttt{GPR}[d]
\end{align*}

If \textit{c} has not been used before, the last line will store the register name in \textit{c}, like the first line rather than the second, as \textit{c} will have an inferred type of \texttt{register <\texttt{bit}[64]>}. If local variables had to have a declared type, this wouldn’t be a problem.

Inside functions, the first form seems to be rarely or never needed; at top-level it sometimes is, e.g. when setting up aliases.

Non-mutable local variables can be introduced with \texttt{let}:

\begin{align*}
\textit{exp} & ::= \ldots \quad \text{Expression} \\
& \mid \texttt{letbind in exp} \quad \text{let expression} \\
\textit{letbind} & ::= \quad \text{let binding} \\
& \mid \texttt{let} \ \texttt{typschm pat} = \textit{exp} \quad \text{let, explicit type (\textit{pat} must be total)} \\
& \mid \texttt{let} \ \textit{pat} = \textit{exp} \quad \text{let, implicit type (\textit{pat} must be total)}
\end{align*}

\textsuperscript{10}In hindsight, the latter perhaps should not be supported: it needs the alias definitions to be more than just a macro, which is confusing, e.g. when it comes to the atomicity of accesses to such aliases, and as far as we know ARM doesn’t need this.

\textsuperscript{11}This was probably an error, as it makes it hard to detect typos in repeated uses.
Function parameters, pattern-match variables, let-bound variables, and for-loop-introduced variables are not mutable.\footnote{The type checker should but doesn’t check that they can’t be of type \texttt{reg <typ>}.}

- functions \( \texttt{typ}_1 \rightarrow \texttt{typ}_2 \) \texttt{effect} \( \texttt{effect} \): the type of functions with parameters of type \( \texttt{typ}_1 \) and return-type \( \texttt{typ}_2 \), and whose bodies involve the effects \( \texttt{effect} \). We return below (§4.5) to what those can be. Only first-order functions are permitted, and only at top-level (in a \texttt{val} declaration for a function); function types cannot be nested within other types.

Functions can be introduced with top-level definitions:

\[
\texttt{fundef} ::= \begin{align*}
\text{function definition} & | \quad \text{function} \ 	exttt{rec} \ _\text{opt} \ 	exttt{tannot} \ _\text{opt} \ 	exttt{effect} \ _\text{opt} \ 	exttt{funcl} \ _1 \ \text{and} \ldots \text{and} \ 	exttt{funcl}_n
\end{align*}
\]

\[
\texttt{rec} \ _\text{opt} ::= \begin{align*}
\text{optional recursive annotation for functions} & | \quad \text{non-recursive} \\
& | \quad \text{recursive}
\end{align*}
\]

\[
\texttt{funcl} ::= \begin{align*}
\text{function clause} & | \quad \text{id pat} = \text{exp}
\end{align*}
\]

and functions are used in application expressions:

\[
\texttt{exp} ::= \ldots \text{Expression} \quad \text{function application}
\]

(if \( \texttt{exp} \) is a tuple, the extra parentheses can be omitted).

- another construct is syntactically similar to a type, but can only occur as the first element of a function argument type tuple: \texttt{implicit<typ>}; it indicates that that part of the argument is an implicit argument

- the type grammar includes a wildcard type \( _\)\texttt{arg} , but we believe this has not been used\footnote{The wildcard type should be removed.}.

\subsection{The type grammar}

\[
\texttt{typ} ::= \begin{align*}
_ & | \ id & | \ kid & | \ \texttt{typ}_1 \rightarrow \texttt{typ}_2 \ \texttt{effect} \ \texttt{effect} \\
( \texttt{typ}_1 , \ldots , \texttt{typ}_n ) & | \ id < \texttt{typ}_\texttt{arg}_1 , \ldots , \texttt{typ}_\texttt{arg}_n > \\
( \texttt{typ} ) & | \ [ \texttt{typ} ] & | \ [ \texttt{typ} : \texttt{typ} ] \\
& | \ [ \texttt{typ} \ : \texttt{typ}_1 \ : \ldots \ : \texttt{typ}_n ] & | \ [ \texttt{typ} \ : \texttt{typ} \ : \texttt{typ}_1 \ : \ldots \ : \texttt{typ}_n ] \\
& | \ [ \texttt{typ} \ : \texttt{typ} ] & | \ [ \texttt{typ} \ : \texttt{typ} ] \\
& | \ [ \texttt{typ} \ : \texttt{typ} \ : \texttt{typ} ] & | \ [ \texttt{typ} \ : \texttt{typ} \ : \texttt{typ} ] \\
& | \ [ \texttt{typ} \ : \texttt{typ} \ : \texttt{typ} ] & | \ [ \texttt{typ} \ : \texttt{typ} \ : \texttt{typ} ]
\end{align*}
\]

\[
\texttt{typ}_\texttt{arg} ::= \begin{align*}
\text{type constructor arguments of all kinds}
\end{align*}
\]

\begin{footnotesize}
\begin{enumerate}
\item Function (first-order only in user code)
\item Tuple
\item type constructor application
\item sugar for \texttt{range<0, nexp>}
\item sugar for \texttt{range<nexp, nexp'>}
\item sugar for \texttt{atom<nexp>=range<nexp, nexp>}
\item sugar for vector indexed by \texttt{[| nexp |]}
\item sugar for vector indexed by \texttt{[| nexp..nexp' |]}
\item sugar for increasing vector
\item sugar for decreasing vector
\item type constructor arguments of all kinds
\end{enumerate}
\end{footnotesize}
typ ::= order effect

kinded IDs: Type, Nat, Order, and Effect variables

The built-in initial kind environment of type constructors, id above (defined in type_internal.ml:initial_kind_env) is:

- unit Type
- bit Type
- string Type
- list Type -> Type
- reg Type -> Type
- register Type -> Type
- range Nat -> Nat -> Type
- vector Nat -> Nat -> Order -> Type -> Type
- atom Nat -> Type
- option Type -> Type
- bool Type (abbreviation for a range type)
- nat Type (abbreviation for a range type)
- int Type (abbreviation for a range type)
- uint8 Type (abbreviation for a range type)
- uint16 Type (abbreviation for a range type)
- uint32 Type (abbreviation for a range type)
- uint64 Type (abbreviation for a range type)

4.2 Dependent types for vector lengths, start indices, and integer ranges

4.2.1 Types with Nat constants

The combination of integer range types, and vector types that include their vector size and start-index, lets type checking of assignments detect range mismatches. For example:

- (bit[5]) x; (* declares x of type: vector of bits of length 5 *)
- ([| 31 |]) y; (* declares y of type: integer in 0..31 inclusive *)
- (int) z; (* declares z of type: unbounded integer *)

- x := y (* implicit coercion lets this type-check ok *)
- x := z (* ...while this is a type error *)

Vector accesses are also range-checked:

- ([| 5 |]) i; (* declares i of type: integer in 0..5 inclusive *)
- x[i]; (* this is ok *)
- x[z]; (* ...while this is a type error *)

4.2.2 Types with Nat variables and Nat-polymorphic functions

These sizes, start indices, and ranges are not limited to constants. For a start, types and functions can be polymorphic in type-level integer variables. For example, this CountLeadingSignBits function:

- function forall Nat 'N.
  - [|'N|] CountLeadingSignBits((bit['N]) x) = ...

for an arbitrary natural number 'N, takes an 'N-bit vector x, of type bit['N], and returns an integer between 0 and 'N inclusive, of type [|'N|].
4.2.3 Types with Nat arithmetic

Sizes, start indices, and ranges can also involve some arithmetic. For example, bitvector concatenation $exp_1 : exp_2$ takes an $exp_1$, of type $\text{bit}[M]$, and an $exp_2$, of type $\text{bit}[N]$, and produces their concatenation, of type $\text{bit}[M+N]$, for any $M$ and $N$.

The arithmetic expressions that are allowed in types are distinct from normal expressions: they must be from the following grammar:

\[
\text{nexp ::= numeric expression, of kind Nat} \\
| \text{id \quad abbreviation identifier} \\
| \text{kid \quad variable} \\
| \text{num \quad constant} \\
| \text{nexp_1 * nexp_2 \quad product} \\
| \text{nexp_1 + nexp_2 \quad sum} \\
| \text{nexp_1 - nexp_2 \quad subtraction} \\
| \text{2**nexp \quad exponential} \\
| (\text{nexp}) \quad S
\]

In the formal type system, they must have kind $\text{Nat}$, rather than the kind $\text{Type}$ that classifies the types of normal expressions. Note that term-level variables, function applications, and other term-level expressions of integer $\text{range}$ type are not permitted here: to keep type checking static, Sail enforces a clear distinction between $\text{Nat}$ expressions and term-level expressions, and variables of kind $\text{Nat}$ (or any other kind) are distinct from term-level variables: they must start with a ‘ whereas term-level variables must not. Term-level expressions might have a type $\text{range<exp_1, exp_2>}$, which itself has kind $\text{Type not Nat}$.

However, there is an expression-level construct that gives the value of a $\text{Nat}$ expression $nexp$ at runtime:\

\[
\text{exp ::= ...} \\
| \text{sizeof nexp \quad the value of nexp at run time}
\]

4.2.4 Types with constrained quantification of Nat variables

Quantified $\text{Nat}$-variables can also be subject to constraints. For example, the $\text{MEMw.EA}$ function of Fig. 2, used to announce the effective address and number of bytes of an upcoming memory write, has type:

\[
\text{val extern forall Nat 'n, 'n IN \{1,2,4,8,16\}.} \\
(\text{bit[64], [:'n:]}) -> \text{unit \quad effect \{eanmem\} \text{ MEMw.EA}}
\]

Here ‘$n$’ is constrained to be within the finite set \{1,2,4,8,16\}. Equality and comparisons are also permitted, in the grammar of constraints below:\

\[
\text{n_constraint ::= \quad constraint over kind Nat} \\
| \text{nexp = nexp'} \\
| \text{nexp >= nexp'} \\
| \text{nexp <= nexp'} \\
| \text{kid IN \{num_1, ... , num_n\}}
\]

Sail type inference involves solving constraints arising from types with $\text{Nat}$ arithmetic and from these expressions. In general this is undecidable, but those that actually occur in the specifications we have developed are solvable by an ad hoc solver built in to Sail.

4.2.5 Type definitions with quantified Nat variables

User definitions of record and tagged union types can also be parameterised on constrained $\text{Nat}$ variables. For example, our ARMv8 (hand) specification has an AST type $\text{ast<"R, 'D>}$ of decoded machine instructions

\footnote{\text{Nat} should be renamed $\text{Num}$, as constraints can also include negative integers and positive and negative infinities.} \footnote{$\text{sizeof}$ is a confusing name for this, as it just gives the value of the $nexp$; we should rename it \footnote{There are more constraint forms internally, some of which we might want to expose to the user, such as $<$ and $>$. Exposing $\langle>$ and the predicate constraints to the user is another question, which isn’t clear.}}
that is parametric on two Nat variables, for the register size and data size of instructions, each of which is constrained to a particular finite set of values:

```plaintext
typedef ast = const union forall Nat 'R, 'R IN {32, 64}, /* register size */
               Nat 'D, 'D IN {8,16,32,64}. /* data size */
{
  (reg_index,reg_index,AccType,MemOp,boolean,boolean,boolean,bit[64],[:'R:],[:'D:]) LoadImmediate;
  ...
}
```

One of the decode functions, for the LoadImmediate instructions, is:

```plaintext
function forall Nat 'R, 'R IN {32,64}, Nat 'D, 'D IN {8,16,32,64}.
  option<ast<'R,'D>> sharedDecodeLoadImmediate((bit[2]) opc,(bit[2]) size,Rn,Rt,wback,
                                            postindex,(uinteger) scale,offset,acctype,(bool) prefetchAllowed) =
{
  ...
  ([:'D:]) datasize := lsl(8, scale);
  ...
}
```

Most of the ARM memory access instructions have flavours for accessing 1, 2, 4 and 8 bytes (i.e. 8, 16, 32 and 64 bits). This information is recorded in the AST using the Nat variable 'D. In the function above 'D is used in the singleton range type of datasize. This allows us, later, to define bit vectors with suitable size (e.g. bit['D]) and also to use datasize as rvalue.

In a similar way, 'R is used to record the register size an instruction is accessing (in AArch64 each of the 31 general purpose registers can be accessed as a 32 bits register or a 64 bits register).

### 4.2.6 Implicit Nat arguments

Consider a function declaration and definition, e.g.

```plaintext
val forall Nat 'n. (implicit<'n>,bit) -> bit['n] replicate

function forall Nat 'n bit['n] replicate ((bit)i) = ...
```

and usage

```plaintext
...(bit[64]) replicate(0)
```

Where the return type of the function application is dependent on the context in which it’s called – i.e., when it includes a Nat variable that is not present in its argument types and not completely constrained by its constraints, then an implicit parameter is necessary. It must be the first parameter of the function (multiple such parameters are not supported). The definition and call site do not (and cannot) mention this parameter, but it is there at runtime. Our models all rely on this.

### 4.3 Order polymorphism

Sail functions can also be polymorphic in the indexing order used for vectors. For example, the built-in `~` bitwise-not function can be used on any bit-vector type, irrespective of its start index ('n), its size ('m), and its indexing order ('o), as in this type:

```plaintext
(bitwise_not) forall Nat 'n, Nat 'm, Order 'o.
  vector<'n, 'm, 'o, bit> -> vector<'n, 'm, 'o, bit>
```

Here 'o is a type-level variable of kind Order, ranging over inc and dec. This is typically used only in the types of built-in functions, as any particular ISA specification usually uses only one indexing direction for bitvectors.
4.4 Parametric type polymorphism

Sail functions can also be polymorphic over types, broadly as in ML polymorphism, Java generics, and suchlike. For example, the built-in vector `length` function:

\[
\text{length} : \forall \text{Type} ~ \forall \text{Nat} \ 'n, \text{Nat} \ 'm, \text{Order} \ '\text{ord}. \\
\text{vector}<'n, 'm, '\text{ord}, 'a> \to \text{atom}<'m>
\]

takes a value of an arbitrary vector type \text{vector}<'n, 'm, 'ord, 'a>, where 'n and 'm are arbitrary \text{Nat} start index and length, 'ord is an arbitrary order, and 'a is a type-level variable of kind \text{Type}, which can be instantiated to an arbitrary type. It returns an integer in the singleton range \text{atom}<'m>. Parametric type polymorphism is also typically used only in the built-in function types.

4.5 Effects

Sail has a simple effect system to keep track of the side-effects of functions, so that (for example) pure functions can be easily identified as such. There are a number of built-in base effects, listed below, to record the presence of register and memory accesses, memory barriers, dynamic footprint recalculation, etc.

\[
\text{base\_effect} ::= \text{effect} \\
| \text{rreg} \quad \text{read register} \\
| \text{wreg} \quad \text{write register} \\
| \text{rmem} \quad \text{read memory} \\
| \text{wmem} \quad \text{write memory} \\
| \text{wmea} \quad \text{signal effective address for writing memory} \\
| \text{wmv} \quad \text{write memory, sending only value} \\
| \text{barr} \quad \text{memory barrier} \\
| \text{depend} \quad \text{dynamic footprint} \\
| \text{undef} \quad \text{undefined-instruction exception} \\
| \text{unspec} \quad \text{unspecified values} \\
| \text{nondet} \quad \text{nondeterminism, from non\text{det}} \\
| \text{escape} \quad \text{potential call of exit} \\
| \text{lset} \quad \text{local mutation; not user-writable} \\
| \text{lret} \quad \text{local return; not user-writable}
\]

- \text{rreg} and \text{wreg} are introduced by expressions that access things of register type
- \text{rmem} through to \text{depend} are principally introduced by external function definitions that have those effects.
- \text{undef} arises from usage of the \text{undefined} expression (a literal in the grammar)
- \text{unspec} is not currently supported
- \text{nondet} arises from \text{nondet} blocks (none of our examples use those at present)
- \text{escape} arises from use of the \text{exit} expression
- \text{lset} is not in the user syntax. It arises from local mutation, for use by the monad translations
- \text{lret} is not in the user syntax. It’s for future use by the monad translations, to track whether expressions contain \text{return}

Function types \text{typ}_1 \to \text{typ}_2 \quad \text{effect} \quad \text{effect} include an effect expression, typically a finite set of base effects:

\[
\text{effect} ::= \text{effect set, of kind Effect} \\
| \text{kid}
\]
Top-level function definitions can have an optional effect annotation, if the user wants to express the effects allowed in the function body, or they can be left to be inferred by the type-checker.

\[\text{fundef} ::= \text{function definition} \]

\[| \quad \text{function rec}_\text{opt tannot}_\text{opt effect}_\text{opt funcl}_1 \text{ and } ... \text{ and funcl}_n\]

\[
\text{effect}_\text{opt} ::= \text{optional effect annotation for functions} \\
| \quad \text{sugar for empty effect set} \\
| \quad \text{effect effect}
\]

At present effect declarations must exactly match the possible effects of a function body, rather than allowing a proper inclusion. This has caught some forgotten-register-access bugs.

Sail also supports effect polymorphism: one can have type-level variables of kind \text{Effect} and quantify them in the same way as \text{Nat}, \text{Type}, and \text{Order} variables, but this seems not to be useful in practice.

### 4.6 Kinds, type constructors, and type applications

Putting this together, Sail type constructors, such as \text{range} and \text{vector}, are each classified by some \text{kind} of the following grammar:

\[\text{base}_\text{kind} ::= \text{base kind} \]

\[| \quad \text{Type} \quad \text{kind of types} \]

\[| \quad \text{Nat} \quad \text{kind of natural number size expressions} \]

\[| \quad \text{Order} \quad \text{kind of vector order specifications} \]

\[| \quad \text{Effect} \quad \text{kind of effect sets} \]

\[\text{kind} ::= \text{kinds} \]

\[| \quad \text{base}_\text{kind}_1 \rightarrow ... \rightarrow \text{base}_\text{kind}_n \]

For example:

\[\text{range} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Type} \]

\[\text{vector} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Order} \rightarrow \text{Type} \rightarrow \text{Type} \]

and a vector type such as \text{vector}<0,32, \text{inc}, \text{bit}[64]> is a type-constructor application of \text{vector} to the \text{Nat}-kinded terms 0 and 32, the \text{Order}-kinded term \text{inc}, and the \text{Type}-kinded term \text{bit}[64].

### 4.7 Type schemes

A Sail type scheme \text{typschm} is a type expression \text{typ} together an optional list of universally quantified kinded identifiers and \text{Nat} constraints:

\[\text{typschm} ::= \]

\[| \quad \text{typquant typ} \]

\[| \quad \text{forall quant}_\text{item}_1, ..., \text{quant}_\text{item}_n . \]

\[| \quad \text{kinded_id} \]

\[| \quad \text{n_constraint} \]

and a vector type such as \text{vector}<0,32, \text{inc}, \text{bit}[64]> is a type-constructor application of \text{vector} to the \text{Nat}-kinded terms 0 and 32, the \text{Order}-kinded term \text{inc}, and the \text{Type}-kinded term \text{bit}[64].
4.8 Flow-sensitive constraints

See §14.1.3.

4.9 Implicit coercions

To support the idioms we see in vendor specifications, Sail automatically infers and inserts coercions between various bit, bitvector, integer range, and enumeration types. It also inserts coercions under tuples, coercions for explicit type annotations, and coercions for implicit dereferencing of registers. Note that coercion insertion is not done in patterns. Coercion insertion is done by type_internal.ml:type_coerce_internal. It includes:

- recursive coercion insertion for components of tuple types
- coercion between two vector types to change the start index
- coercion from a bitvector to a corresponding integer range type
- coercion from an integer range type to a bitvector. Currently this allows lossy coercions if there is an explicit type annotation in the source, but that should probably be an error and we should require an explicit truncation.
- coercion for implicit dereferencing of a register, from register<typ> to typ, and then recursively perhaps also from typ to the expected type
- coercion from a bitvector of length 1 to bit and vice versa
- coercion from a bit to an integer range that includes 0..1
- coercion from a integer range 0..1 to bit
- coercion from a integer range to an enumeration type
- coercion from a bitvector to an enumeration type
- coercion from an enumeration type to an integer range

5 The Sail language: expressions

5.1 Pattern matching

Sail values can be deconstructed by pattern matching, in let expressions and in switch expressions. For example, the Fig. 1 ARMv8 (hand) specification includes:

```ml
let (result,nzcv) = AddWithCarry(operand1, operand2, carry_in) in {
    which invokes a Sail AddWithCarry function (defined elsewhere in the specification) that returns a pair, and binds the two components of that pair to variables result and nzvc in the subsequent code. The execute function for the Power stdu instruction in Fig. 1:
```

```ml
function clause execute (Stdu (RS, RA, DS)) =
    { EA := GPR[RA] + EXTS (DS : 0b00);
      MEMw(EA,8) := GPR[RS];
      GPR[RA] := EA }
```

is also a clause of a pattern match, matching the Stdu variant of the ast tagged-union type and binding the variables RS, RA, and DS to the three components of the triple of that variant.

Most of the pattern forms were introduced in §4, with patterns for tagged unions, records, vectors, tuples, and lists. Patterns can also include wildcards, type annotations, and as patterns (to bind a variable to whatever matches a subpattern):
Pattern matching is first-match (except for exhaustive use of the interpreter, where it continues also to look for other matches).\textsuperscript{17}

### 5.2 Control flow

Sail control flow is largely standard. Evaluation is left-to-right. Expressions of type \texttt{unit} can be composed into sequential blocks with ;. The syntax also includes nondeterministic blocks, but they are not yet used in our specifications.

\[
\begin{align*}
\text{exp} & ::= \ldots & \text{Expression} \\
| & \text{letbind in exp} & \text{let expression} \\
| & \text{switch exp \{ case pexp\_1 \ldots case pexp\_n \}} & \text{pattern matching} \\
\end{align*}
\]

\[
\begin{align*}
\text{letbind} & ::= \ldots & \text{let binding} \\
| & \text{let typschm pat = exp} & \text{let, explicit type (pat must be total)} \\
| & \text{let pat = exp} & \text{let, implicit type (pat must be total)} \\
\end{align*}
\]

\[
\begin{align*}
\text{pexp} & ::= \ldots & \text{pattern match} \\
| & \text{pat -> exp} & \text{pattern} \\
\end{align*}
\]

\[
\begin{align*}
\text{pat} & ::= \ldots & \text{literal constant pattern} \\
| & \text{lit} & \text{wildcard} \\
| & \_ & \text{named pattern} \\
| & (\text{pat as id}) & \text{typed pattern} \\
| & (\text{typ }\text{pat}) & \text{identifier} \\
| & \text{id} & \text{union constructor pattern} \\
| & \text{id (pat}_1, \ldots, \text{pat}_n) & \text{struct pattern} \\
| & \{\text{fpat}_1 ; \ldots ; \text{fpat}_n ; ?\} & \text{vector pattern} \\
| & \{\text{pat}_1 ; \ldots ; \text{pat}_n\} & \text{vector pattern (with explicit indices)} \\
| & \text{pat}_1 : \ldots : \text{pat}_n & \text{tuple pattern} \\
| & \text{[| pat}_1, \ldots, \text{pat}_n |\} & \text{list pattern} \\
| & (\text{pat}) & \text{field pattern} \\
\end{align*}
\]

\[
\begin{align*}
\text{fpat} & ::= \ldots & \text{field pattern} \\
| & \text{id = pat} & \text{S} \\
\end{align*}
\]

There is a general form of \texttt{for} loop, together with several sugared variations.\textsuperscript{18}

\[
\begin{align*}
\text{exp} & ::= \ldots & \text{Expression} \\
| & \text{foreach (id from exp}_1\text{ to exp}_2\text{ by exp}_3\text{ in order ) exp}_4 & \text{loop} \\
\end{align*}
\]

\textsuperscript{17}Sail does not attempt to check exhaustiveness of patterns. It would be desirable to provide warnings of missing and redundant cases.

\textsuperscript{18}It might be useful to have another sugared loop form, for each index of a vector or range.
As discussed under Pattern Matching above (§5.1), `switch` expressions let one case split, e.g. on the value of a tagged union:

\[
\begin{align*}
\text{foreach} & \quad \text{(id from exp}_1 \to \text{exp}_2 \text{ by exp}_3) \ \text{exp}_4 \quad S \\
\text{foreach} & \quad \text{(id from exp}_1 \text{ to exp}_2) \ \text{exp}_3 \quad S \\
\text{foreach} & \quad \text{(id from exp}_1 \text{ downto exp}_2 \text{ by exp}_3) \ \text{exp}_4 \quad S \\
\text{foreach} & \quad \text{(id from exp}_1 \text{ downto exp}_2) \ \text{exp}_3 \quad S
\end{align*}
\]

Finally, `return` permits early return from a function body, e.g. from inside a loop, and `exit` and `assert` permit termination of the running Sail code:

\[
\begin{align*}
\text{exp} & \quad ::= \ldots \quad \text{Expression} \\
\text{switch} \text{ exp} & \quad \{ \text{ case pexp}_1 \ldots \text{ case pexp}_n \} \quad \text{pattern matching} \\
\text{return} & \quad \text{exp} \quad \text{return exp from current function} \\
\text{exit} & \quad \text{exp} \quad \text{halt all current execution} \\
\text{assert} & \quad (\text{exp} , \text{exp}') \quad \text{halt with error exp' when not exp}
\end{align*}
\]

The exit

\[
\text{exit}( f(3) )
\]

looks syntactically like a function application of `exit`, but its semantics is to first discard the current context and then evaluate the expression `f(3)`. The `f` might be a function that signals an exception. In MIPS and CHERI, this is used for exceptional termination of the current instruction. In ARMv8 (hand), it’s used to signal points where the ISA model does not implement something. In ARMv8 (ASL), we currently use `exit` where the ASL has an instruction-level exception.\(^\text{19}\)

The `return` is semantically a standard early return.\(^\text{20}\)

The `assert` triggers a stop-the-world exception in the interpreter. The ocaml shallow embedding removes `asserts`, the lem shallow embedding maps them onto lem assertions.

### 5.3 Assignment and l-values

It is common in ISA specifications to assign to complex l-values, e.g. to a subvector or named field of a bitvector register, or to an l-value computed with some auxiliary function, e.g. to select the appropriate register for the current execution model. Accordingly, Sail supports a rich language of l-values.\(^\text{21}\)

\[
\begin{align*}
\text{exp} & \quad ::= \ldots \quad \text{Expression} \\
\text{lexp} & \quad ::= \quad \text{lvalue expression} \\
& \quad | \quad \text{id} \quad \text{identifier} \\
& \quad | \quad \text{id } ( \text{exp}_1 \ldots \text{exp}_n ) \quad \text{memory or register write via function call} \\
& \quad | \quad \text{id exp} \quad \text{sugared form of above for explicit tuple exp} \\
& \quad | \quad ( \text{typ}) \text{id} \quad \text{cast} \\
& \quad | \quad ( \text{lexp}_0 \ldots \text{lexp}_n ) \quad \text{multiple (non-memory) assignment} \\
& \quad | \quad \text{lexp } [ \text{exp} ] \quad \text{vector element} \\
& \quad | \quad \text{lexp } [ \text{exp}_1 \ldots \text{exp}_2 ] \quad \text{subvector} \\
& \quad | \quad \text{lexp } . \text{id} \quad \text{struct field}
\end{align*}
\]

For the memory or register write via function call, the function has to have a `wmem`, `wreg`, or `wmv` effect.

\(^{\text{19}}\)For some of these usages, we discussed introducing an “exceptional path” block.

\(^{\text{20}}\)This is not yet supported by the shallow embedding; here early returns should be implemented as a return effect for which the lem code generator inserts effect handlers at the call sites of functions with return effect.

This would have been useful in the ARMv8 (hand) model but was not available then, so we worked around it. It is used in the table-walking part of MIPS and CHERI. It is used extensively in ARMv8 (ASL).

\(^{\text{21}}\)The lvalue cast `(typ)id` should really be replaced by requiring local variables to be declared explicitly with their types.
5.4 Undefined values

The vendor Power, ARM, and MIPS documents use similar notions of undefined value to express loose specifications on particular register values after an instruction. The Power [3] *undefined value* is a value that:

> “May vary between implementations, and between different executions on the same implementation, and similarly for register contents, storage contents, etc., that are specified as being undefined.”

The MIPS64 specification [6, 7] uses “undefined” in broadly similar way for specific undefined values, e.g. for the arithmetic result value of $\text{DDIVU}$. The ARM [4] *UNKNOWN* is specified with:

> “An expression like $\text{bits}(32) \text{ UNKNOWN}$ indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not: (a) Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not \text{UNPREDICTABLE} or \text{CONSTRAINED UNPREDICTABLE} and do not return \text{UNKNOWN} values, (b) Be promoted as providing any useful information to software.”

Sail provides an *undefined* literal at any type:

$$
lit ::= \ldots \text{Literal constant}
\mid \text{undefined} \quad \text{constant representing undefined values}
$$

For example, it is used in our Power model for the $\text{mulhw}$ *Multiply High Word* instruction to set the high-order bits of the RT result register to undefined, just as the IBM pseudocode does:

```
function clause execute (Mulhw (RT, RA, RB, Rc)) =
{ ...
  (GPR[RT])[32..63] := prod[0 .. 31];
  (GPR[RT])[0..31] := undefined;
  ...
}
```

There are several possible semantic interpretations of *undefined*:

1. Require a processor-implementation-specific choice of how those are calculated, sidestepping the question.
2. Make a runtime nondeterministic choice in a relational semantics
3. Make a runtime nondeterministic choice, either (a) from a bitstream oracle, or (b) from some bits fed in dynamically from outside
4. Work with (a) register values containing lifted bits, or (b) register and memory values containing lifted bits, and halt if they are used in conditional branches or I/O.
5. Work with symbolic register and memory values throughout
6. Map onto the HOL4 or Isabelle/HOL ARB or to default values for targets where that is not available.

One may want different interpretations for different purposes, e.g.:

- for fast emulation of random traces, 3 ia appropriate, but it would make exhaustive enumeration of the possible behaviours impractical
- for checking RTL simulation, 3(b) is appropriate, if one can instrument the simulation to announce the choices it makes.
- for verifying well-behaved software, perhaps 2, 4, or 6.
The Sail interpreter is intended to treat all operations as strict in undefinedness (i.e., if any argument is undefined, the result is)\textsuperscript{22}.

In the shallow embedding, at present undefined is only present in the representation type for bits\textsuperscript{23}.

Neither of these attempt to capture the ARM prose requiring \texttt{UNKNOWN} to not return \textquote{information that cannot be accessed at the current or a lower level of privilege}. Making a precise statement that captures that intent is nontrivial, as it is a (possibly statistical) information-flow property.

5.5 Undefined behaviour

Each architecture document also uses one or more notions of undefined behaviour: Boundedly undefined (Power), \texttt{UNPREDICTABLE} (ARM), and \texttt{UNPREDICTABLE / UNDEFINED / UNSTABLE} behaviour (MIPS). At present we model these with \texttt{exit}, e.g. when an ARM load instruction tries to write-back the address and (Power),

5.6 The full expression grammar

We now collect together the full Sail expression grammar, as introduced above.

\[
\text{expression} ::= \text{sequential block} \\
\text{nondeterministic block} \\
\text{identifier} \\
\text{literal constant} \\
\text{cast} \\
\text{function application} \\
\text{function application to tuple} \\
\text{infix function application} \\
\text{tuple} \\
\text{conditional} \\
\text{loop} \\
\text{vector (indexed from 0)} \\
\text{vector (indexed consecutively)} \\
\text{vector access} \\
\text{subvector extraction} \\
\text{vector functional update} \\
\text{vector subrange update, with vector} \\
\text{vector concatenation} \\
\text{list} \\
\text{cons} \\
\text{struct} \\
\text{functional update of struct} \\
\text{field projection from struct} \\
\text{pattern matching} \\
\text{let expression} \\
\text{imperative assignment} \\
\text{the value of } nexp \text{ at run time}
\]

\textsuperscript{22}Some library functions might have missing cases.

\textsuperscript{23}This works well for our ARMv8 (hand) and POWER specifications. For MIPS / CHERI-MIPS, we wanted undefined numbers. For ARMv8 (ASL), at present local variables are all initially set to \texttt{undefined}, which needs more thought; the intended semantics may be sometimes to collapse the declaration and the first write, sometimes to use an arbitrary default value, and only sometimes real \texttt{undefined}."
| return exp | return exp from current function |
| exit exp | halt all current execution |
| assert (exp , exp') | halt with error exp' when not exp |

\[
\text{lit ::= literal constant} \quad \\
| () | ():unit |
| bitzero | bitzero : bit |
| bitone | bitone : bit |
| true | true : bool |
| false | false : bool |
| num | natural number constant |
| hex | bit vector constant, C-style |
| bin | bit vector constant, C-style |
| string | string constant |
| undefined | undefined-value constant |

\[
\text{fexp ::= field expression} \quad \\
| id = exp |
\]

\[
\text{fexps ::= field expression list} \quad \\
| fexp1 ; ... ; fexpn ; ? |
\]

6 The Sail language: files and top-level definitions

A Sail definition can be composed of multiple files, each of which is a sequence of top-level definitions:

\[
\text{defs ::= definition sequence} \quad \\
| def1 .. defn |
\]

\[
\text{def ::= top-level definition} \quad \\
| kind_def | definition of named kind identifiers |
| type_def | type definition |
| fundef | function definition |
| letbind | value definition |
| val_spec | top-level type constraint |
| default_spec | default kind and type assumptions |
| scattered_def | scattered function and type definition |
| dec_spec | register declaration |

6.1 Nat abbreviations

One can define abbreviations for Nat quantities\(^\text{24}\text{25}\):

\[
\text{kind_def ::= Definition body for elements of kind} \quad \\
| Def kind id name scm_opt = nexp | Nat-expression abbreviation |
\]

6.2 Type definitions

Type definitions, introduced in §4.1, include type abbreviations, the definitions of struct, tagged union, and enumeration types, and the definition of register types with named fields of bitvectors:

\(^\text{24}\text{The parser is limited here: it may be that only a constant or constant plus identifier are supported.}\)
\(^\text{25}\text{The syntax also includes other Def definition forms for kinded identifiers, but they seem to add complexity for no purpose and should be removed from the syntax; they are labelled D (deprecated) in the Ott source. The kind_def nonterminal root should be renamed. The name scm_opt here should also be removed.}\)
\[ type\_def \ ::= \text{type definition body} \\
\mid \text{typedef} \ id \ name\_scm\_opt = \text{typschm} \\
\quad \text{type abbreviation} \\
\mid \text{typedef} \ id \ name\_scm\_opt = \text{const struct typquant} \{ \text{typ}_1 \ id_1 \ ; \ \ldots \ ; \text{typ}_n \ id_n ; ? \} \\
\quad \text{struct type definition} \]
typedef id name\_scm\_opt = const union typquant \{ type\_union1 ; ... ; type\_union\_n ; ? \}
  tagged union type definition
typedef id name\_scm\_opt = enumerate \{ id1 ; ... ; id\_n ; ? \}
  enumeration type definition
typedef id = register bits \{ nexp : nexp' \} \{ index\_range1 : id1 ; ... ; index\_range\_n : id\_n \}
  register mutable bitfield type definition

The name\_scm\_opt here is an optional regular-expression constraint on the syntactic form of identifiers of the defined type:\textsuperscript{26}

\[
\text{name\_scm\_opt ::= optional variable naming-scheme constraint}
\]
\[
\quad | \quad [ \text{name = regexp} ]
\]

6.3 Function definitions

A function definition can have an optional rec (permitting recursive calls), an optional type annotation, and an optional effect annotation. Recursive functions must also have a preceding val specification. The body of the definition consists of a sequence of one or more function clauses, allowing definition by top-level pattern matching:

\[
\text{fundef ::= function definition}
\]
\[
\quad | \quad function rec\_opt tannot\_opt effect\_opt \text{funcl1 and ... and funcl\_n}
\]

\[
\text{rec\_opt ::= optional recursive annotation for functions}
\]
\[
\quad | \quad \text{rec non-recursive}
\]

\[
\text{tannot\_opt ::= optional type annotation for functions}
\]
\[
\quad | \quad \text{rec recursive}
\]

\[
\text{effect\_opt ::= optional effect annotation for functions}
\]
\[
\quad | \quad \text{effect effect sugar for empty effect set}
\]

\[
\text{funcl ::= function clause}
\]
\[
\quad | \quad \text{id pat = exp}
\]

6.4 Value definitions

A top-level let binds the identifiers of its pattern to the result of evaluating the expression, which must be pure:\textsuperscript{27}

\[
\text{letbind ::= let binding}
\]
\[
\quad | \quad \text{let typescm pat = exp let, explicit type (pat must be total)}
\]
\[
\quad | \quad \text{let pat = exp let, implicit type (pat must be total)}
\]

6.5 Type constraints

One can provide explicit type constraints for upcoming definitions, e.g. for documentation that will be checked by the Sail type-checker. Usually this is not required. val specifications are also used to introduce the identifiers and types of external functions, and of external functions from the Lem library:\textsuperscript{28} In the last case, the string should be an explicit path to the required function but will not be checked by Sail.

\textsuperscript{26}This is supported only by the parser and has not been used in our specifications; it should either be removed from the language or properly implemented.

\textsuperscript{27}For uniformity, perhaps the grammar should be refactored to use a tannot\_opt here, and that should have a typescm instead of a separate typquant and typ.

\textsuperscript{28}This may not have been used in our current specifications; perhaps it could be removed.
val_spec ::= value type specification
   \mid \texttt{val \ typeschm id} specify the type of an upcoming definition
   \mid \texttt{val extern \ typeschm id} specify the type of an external function
6.6 Default kind and type assumptions

One can give a default vector indexing order, which applies to all the Sail code in the file. One can also
give default kinds and types for identifiers, which applies to the rest of the file. The intended semantics of
these is that if an id in binding position does not have a kind or type annotation, then we look through the
defaults (in order from the beginning) and pick the first, otherwise it is left to type inference.

\[
default\_spec ::= \begin{align*} 
& \text{default kinding or typing assumption} \\
& | \text{defaultOrder order} \\
& | \text{default base\_kind kid} \\
& | \text{default typeschm id}
\end{align*}
\]

6.7 Scattered definitions

In a Sail specification, sometimes it is desirable to collect together the definitions relating to each machine
instruction (or group thereof), e.g. grouping the clauses of an AST type with the associated clauses of decode
and execute functions, as in Fig. 2. Sail permits this with syntactic sugar for “scattered” definitions:

\[
\text{scattered\_def ::= scattered function and union type definitions} \\
& | \text{scattered function rec\_opt tannot\_opt effect\_opt id} \\
& \quad \text{scattered function definition header} \\
& | \text{function clause funcl} \\
& \quad \text{scattered function definition clause} \\
& | \text{scattered typedef id name\_scm\_opt = const union typquant} \\
& \quad \text{scattered union definition header} \\
& | \text{union id member type\_union} \\
& \quad \text{scattered union definition member} \\
& | \text{end id} \\
& \quad \text{scattered definition end}
\]

Semantically, scattered definitions of types appear at the start of their definition, and scattered definitions
of functions appear at the end. A scattered function definition can be recursive, but mutually recursive
scattered function definitions may not work.\textsuperscript{30,31}

6.8 Register declarations

Register declarations are as introduced in §4.1:

\[
\text{dec\_spec ::= register declarations} \\
& | \text{register typ id} \\
& | \text{register alias id = alias\_spec} \\
& | \text{register alias typ id = alias\_spec}
\]

7 Sail initial type environment

Below we list the built-in Sail type environment: a library of operators on bitvectors, integers, etc. Many
operators are overloaded, for example bitwise or, written |, can be used for single bits or on two bitvectors,
of the same, but arbitrary, length, initial index, and indexing direction. The Sail overloading resolution rewrites

\textsuperscript{29}This may not have been used – perhaps for Power.
\textsuperscript{30}We should check that that case is caught.
\textsuperscript{31}The scattered function definition syntax doesn’t have a way to specify the argument type except with a separate \textbf{val}
specification, which makes (e.g.) the definition of \textbf{execute} in Fig. 2 look strange. This is an unintended consequence of the C-like
syntax.
occurrences of | to one of two functions, with those two types, that have internal names bitwise_or_bit and bitwise_or respectively.\(^{32}\) The built-in functions are all pure; we omit the effect {} annotations on the function types below.

\[\texttt{not} : \text{(bitwise\_not\_bit)} \]
\[\text{bit} \rightarrow \text{bit}\]

\[\sim:\]
\[(\text{bitwise\_not}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{o}.
\text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}> \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}>\]
\[(\text{bitwise\_not\_bit}) \text{\_} \text{bit} \rightarrow \text{bit}\]

\[\mid:\]
\[(\text{bitwise\_or}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{o}.
\text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}> > \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}>\]
\[(\text{bitwise\_or\_bit}) \text{\_} \text{bit}, \text{bit} \rightarrow \text{bit}\]

\[\wedge:\]
\[(\text{bitwise\_xor}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{o}.
\text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}> > \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}>\]
\[(\text{bitwise\_xor\_bit}) \text{\_} \text{bit}, \text{bit} \rightarrow \text{bit}\]

\[\&:\]
\[(\text{bitwise\_and}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{o}.
\text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}> > \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{o}, \text{bit}>\]
\[(\text{bitwise\_and\_bit}) \text{\_} \text{bit}, \text{bit} \rightarrow \text{bit}\]

\[\text{(* bitwise shifts and rotates *)}\]
\[\ll<:\] \[(\text{bitwise\_leftshift}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{ord}.
\text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>, \text{range }<0, '\text{m}> \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>\]
\[\gg > : \] \[(\text{bitwise\_rightshift}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{ord}.
\text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>, \text{range }<0, '\text{m}> \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>\]
\[\ll< = : \] \[(\text{bitwise\_rotate}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Order }'\text{ord}.
\text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>, \text{range }<0, '\text{m}> \rightarrow \text{vector }<'\text{n}, '\text{m}, '\text{ord}, \text{bit}>\]

\[\text{(* bitvector duplicate, extension, and MSB *)}\]
\[\ll<:\]
\[(\text{duplicate}) \text{\_} \forall \text{Nat }'\text{n}.
\text{bit}, \text{atom }<'\text{n}> \rightarrow \text{vector }<0, '\text{n}, \text{inc}, \text{bit}>\]
\[(\text{duplicate\_bits}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Nat }'\text{o}, \text{Order }'\text{ord}.
\text{vector }<'\text{o}, '\text{m}, '\text{ord}, \text{bit}>, \text{atom }<'\text{n}> \rightarrow \text{vector }<'\text{o}, '\text{m}*'\text{n}, '\text{ord}, \text{bit}>\]
\[\text{EXTZ} : \] \[(\text{extz}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Nat }'\text{o}, \text{Nat }'\text{p}, \text{Order }'\text{ord}.
\text{vector }<'\text{o}, '\text{n}, '\text{ord}, \text{bit}> \rightarrow \text{vector }<'\text{p}, '\text{m}, '\text{ord}, \text{bit}>\]
\[\text{EXTS} : \] \[(\text{exts}) \text{\_} \forall \text{Nat }'\text{n}, \text{Nat }'\text{m}, \text{Nat }'\text{o}, \text{Nat }'\text{p}, \text{Order }'\text{ord}.
\text{vector }<'\text{o}, '\text{n}, '\text{ord}, \text{bit}> \rightarrow \text{vector }<'\text{p}, '\text{m}, '\text{ord}, \text{bit}>\]

\(^{32}\)The optimal balance between overloading of built-in operators and inference of coercions between types is debatable. Basically once a function is overloaded then coercion-inference is problematic because there’s too much scope for which parameter or the return to coerce to what, so Sail doesn’t really try. We decided we needed both overloading and coercions to separate pure mathematics from bit manipulations.
most_significant : forall Nat 'n, Nat 'm, Order 'ord.
    vector<':n, 'm, 'ord, bit> -> bit

(* arithmetic *)

+ :
  (add) forall Nat 'n, Nat 'm.
    (atom<':n, atom<':m>) -> atom<':n'+':m>
  (add.vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> vector<':o, 'n, 'ord, bit>
  (add.vec.range) forall Nat 'n, Nat 'o, Nat 'p, Nat 'q, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> range<':q, 2**':n>
  (add.range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (vector<':n, 'm, 'ord, bit>, atom<':o>) -> vector<':n, 'm, 'ord, bit>
  (add.overflow.vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> (vector<':o, 'n, 'ord, bit>, bit, bit)
  (add.vec.range.range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> vector<':n, 'm, 'ord, bit>
  (add.range.vec) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> vector<':n, 'm, 'ord, bit>
  (add.range.vec.range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> range<':o, 'o+2**':m-1>
  (add.vec.bit) forall Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'p, 'ord, bit), bit) -> vector<':o, 'p, 'ord, bit>
  (add.bit.vec) forall Nat 'o, Nat 'p, Order 'ord.
    (bit, vector<':o, 'p, 'ord, bit>) -> vector<':o, 'p, 'ord, bit>

+s : 
  (add.signed) forall Nat 'n, Nat 'm.
    (atom<':n, atom<':m>) -> atom<':n'+':m>
  (add.vec.signed) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> vector<':o, 'n, 'ord, bit>
  (add.vec.range.signed) forall Nat 'n, Nat 'o, Nat 'p, Nat 'q, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> range<':q, 2**':n>
  (add.range.signed) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (vector<':n, 'm, 'ord, bit>, atom<':o>) -> vector<':n, 'm, 'ord, bit>
  (add.overflow.vec.signed) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> (vector<':o, 'n, 'ord, bit>, bit, bit)
  (add.vec.range.range.signed) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> vector<':n, 'm, 'ord, bit>
  (add.range.vec.signed) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> range<':o, 'o+2**':m-1>
  (add.range.vec.range.signed) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o <= 2**':m-1.
    (atom<':o, vector<':n, 'm, 'ord, bit>) -> range<':o, 'o+2**':m>
  (add.vec.bit.signed) forall Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'p, 'ord, bit), bit) -> vector<':o, 'p, 'ord, bit>
  (add.overflow.vec.bit.signed) forall Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'p, 'ord, bit), bit) -> (vector<':o, 'p, 'ord, bit>, bit, bit)
  (add.bit.vec.signed) forall Nat 'o, Nat 'p, Order 'ord.
    (bit, vector<':o, 'p, 'ord, bit>) -> vector<':o, 'p, 'ord, bit>

- :
  (minus) forall Nat 'n, Nat 'm.
    (atom<':n, atom<':m>) -> atom<':n'-':m>
  (minus.vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> vector<':o, 'n, 'ord, bit>
  (minus.vec.range) forall Nat 'm, Nat 'n, Nat 'o, Nat 'p, Order 'ord.
    (vector<':o, 'n, 'ord, bit>, vector<':p, 'n, 'ord, bit>) -> atom<':m>
  (minus.vec.range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
    (vector<':n, 'm, 'ord, bit>, atom<':o>) -> vector<':n, 'm, 'ord, bit>
forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> range\( 'o \), \( 'o+2^{+'}m \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(atom\( 'o \), vector\( 'n \), 'm, 'ord, bit\)) -> vector\( 'n \), 'm, 'ord, bit\)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(atom\( 'o \), vector\( 'n \), 'm, 'ord, bit\)) -> range\( 'o \), \( 'o+2^{+'}m \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> (vector\( 'o \), 'n, 'ord, bit\), bit, bit)

forall Nat \( 'o \), Nat \( 'p \), Order \( 'ord \).
(vector\( 'o \), 'p, 'ord, bit\), bit) -> (vector\( 'o \), 'p, 'ord, bit\), bit, bit)

\( -\_s : \)

forall Nat \( 'n \), Nat \( 'm \).
(atom\( 'n \), atom\( 'm \)) -> atom\( 'n-'m \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> vector\( 'o \), 'n, 'ord, bit\)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> vector\( 'n \), 'm, 'ord, bit\)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> range\( 'o \), \( 'o+2^{+'}m \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> range\( 'o \), \( 'o+2^{+'}m \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> range\( 'o \), \( 'o+2^{+'}m \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> (vector\( 'o \), 'n, 'ord, bit\), bit, bit)

forall Nat \( 'o \), Nat \( 'p \), Order \( 'ord \).
(vector\( 'o \), 'p, 'ord, bit\), bit) -> (vector\( 'o \), 'p, 'ord, bit\), bit, bit)

\( * : \)

forall Nat \( 'n \), Nat \( 'm \).
(atom\( 'n \), atom\( 'm \)) -> atom\( 'n* 'm \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Nat \( 'q \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> vector\( 'q \), 'n* 'm \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> vector\( 'q \), 'm* 'm \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> vector\( 'q \), 'm* 'm \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Nat \( 'q \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> (vector\( 'o \), 'n, 'ord, bit\), bit, bit)

\( *\_s : \)

forall Nat \( 'n \), Nat \( 'm \).
(atom\( 'n \), atom\( 'm \)) -> atom\( 'n* 'm \)

forall Nat \( 'n \), Nat \( 'o \), Nat \( 'p \), Nat \( 'm \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> vector\( 'o \), 'n* 'm \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> vector\( 'p \), 'm* 'm \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), atom\( 'o \)) -> vector\( 'p \), 'm* 'm \)

forall Nat \( 'n \), Nat \( 'p \), Nat \( 'm \), Order \( 'ord \).
(vector\( 'o \), 'n, 'ord, bit\), vector\( 'p \), 'n, 'ord, bit\)) -> (vector\( 'o \), 'n* 'm \)

\( mod : \)

forall Nat \( 'n \), Nat \( 'o \), \( 'o \geq 1 \).
(atom\( 'n \), atom\( 'o \)) -> range<0, \( 'o+1 \)

forall Nat \( 'n \), Nat \( 'm \), Nat \( 'o \), Order \( 'ord \), \( 'o \geq 1 \).
(vector\( 'n \), 'm, 'ord, bit\), range<1, 'o>) -> vector\( 'n \), 'm, 'ord, bit\)

forall Nat \( 'n \), Nat \( 'm \), Order \( 'ord \).
(vector\( 'n \), 'm, 'ord, bit\), vector\( 'n \), 'm, 'ord, bit\)) -> vector\( 'n \), 'm, 'ord, bit\)

mod : \( \)
mod_s : 
(mod_signed) forall Nat 'n, Nat 'o, 'o >= 1. 
(atom<('n, atom<('o-> range<0, 'o-1)
(mod_signed_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o >= 1. 
(vector<('n, 'm, 'ord, bit>, range<1, 'o>) -> vector<('n, 'm, 'ord, bit>
(mod_signed_vec) forall Nat 'n, Nat 'm, Order 'ord. 
(vector<('n, 'm, 'ord, bit>, vector<('n, 'm, 'ord, bit>) -> vector<('n, 'm, 'ord, bit>

div : (quot) forall Nat 'n, Nat 'm, Nat 'o, 'o^'o <= 'm. 
(atom<('n, atom<('m-> atom<('o>

quot : 
(quot) forall Nat 'n, Nat 'm, Nat 'o, 'o^'o <= 'm. 
(quot<('n, atom<('m) -> atom<('o>
(quot_vec) forall Nat 'n, Nat 'm, Nat 'p, Nat 'q, Order 'ord, 'm >= 'q. 
(vector<('n, 'm, 'ord, bit>, vector<('p, 'q, 'ord, bit>) -> vector<('n, 'm, 'ord, bit>
(quot_overflow_vec) forall Nat 'n, Nat 'm, Nat 'p, Nat 'q, Order 'ord, 'm >= 'q. 
(vector<('n, 'm, 'ord, bit>, vector<('p, 'q, 'ord, bit>) -> (vector<('n, 'm, 'ord, bit>, bit, bit)

(quot_signed) forall Nat 'n, Nat 'm, Nat 'o, Nat 'p, Nat 'q, Nat 'r, 'p*'r <= 'm. 
(range<('n, range<('m, range<('o, 'p) -> range<('q, 'r>
(quot_vec_signed) forall Nat 'n, Nat 'm, Nat 'p, Nat 'q, Order 'ord, 'm >= 'q. 
(vector<('n, 'm, 'ord, bit>, vector<('p, 'q, 'ord, bit>) -> vector<('n, 'm, 'ord, bit>
(quot_overflow_vec_signed) forall Nat 'n, Nat 'm, Nat 'p, Nat 'q, Order 'ord, 'm >= 'q. 
(vector<('n, 'm, 'ord, bit>, vector<('p, 'q, 'ord, bit>) -> (vector<('n, 'm, 'ord, bit>, bit, bit)

(* additional arithmetic on singleton ranges; vector length *)
** : (power) forall Nat 'o. 
(atom<2>, atom<('o-> atom<2**('o>

abs : (abs) forall Nat 'n, Nat 'm. 
(atom<('n) -> range<0, ('m>

max : (max) forall Nat 'n, Nat 'm, Nat 'o. 
(atom<('n, atom<('m) -> atom<('o>

min : (min) forall Nat 'n, Nat 'm, Nat 'o. 
(atom<('n, atom<('m) -> atom<('o>

length : (length) forall Type 'a, Nat 'n, Nat 'm, Order 'ord. 
(vector<('n, 'm, 'ord, 'a) -> atom<('m>

(* comparisons *)
== : 
(eq_vec) forall Nat 'n, Nat 'm, Nat 'o, Type 'a, Order 'ord. 
(vector<('n, 'm, 'ord, 'a>, vector<('o, 'm, 'ord, 'a>) -> bit
(eq_range_vec) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord. 
(atom<('o, vector<('n, 'm, 'ord, bit>) -> bit
(eq_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord. 
(vector<('n, 'm, 'ord, bit>, atom<('o>) -> bit
(eq.range) forall Nat 'n, Nat 'm, flow_constraints('n = 'm, 'n != 'm). 
(atom<('n, atom<('m) -> bit
(eq.bit) 
(bit, bit) -> bit
(eq) forall Type 'a. 
(('a, 'a) -> bit

!= :
(neq_vec) forall Nat 'n, Nat 'm, Nat 'o, Type 'a, Order 'ord.
    (vector<'n, 'm, 'ord, 'a>, vector<'o, 'm, 'ord, 'a>) -> bit
(neq_range_vec) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
    (atom<o>, vector<'n, 'm, 'ord, bit>) -> bit
(neq_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
    (vector<'n, 'm, 'ord, bit>, atom<o>) -> bit
(neq_range) forall Nat 'n, Nat 'm, flow_constraints('n = 'm, 'n != 'm).
    (atom<o>, atom<>'m) -> bit
(neq_bit) (bit, bit) -> bit
(neq) forall Type 'a.
    ('a, 'a) -> bit

< :
    (lt) forall Nat 'n, Nat 'm, flow_constraints('n < 'm, 'n >= 'm).
        (atom<>'n, atom<>'m) -> bit
(lt_vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
(lt_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
        (vector<'n, 'm, 'ord, bit>, atom<o>) -> bit
<u :
    (lt_unsigned) forall Nat 'n, Nat 'm, flow_constraints('n < 'm, 'n >= 'm).
        (atom<>'n, atom<>'m) -> bit
(lt_vec_unsigned) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
<s : (lt_signed) forall Nat 'n, Nat 'm, flow_constraints('n < 'm, 'n >= 'm).
        (atom<>'n, atom<>'m) -> bit
(lt_vec_signed) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
>
    (gt) forall Nat 'n, Nat 'm, flow_constraints('n > 'm, 'n <= 'm).
        (atom<>'n, atom<>'m) -> bit
(gt_vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
(gt_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
        (vector<'n, 'm, 'ord, bit>, atom<o>) -> bit
>_u :
    (gt_unsigned) forall Nat 'n, Nat 'm, flow_constraints('n > 'm, 'n <= 'm).
        (atom<>'n, atom<>'m) -> bit
(gt_vec_unsigned) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
>_s :
    (gt_signed) forall Nat 'n, Nat 'm, flow_constraints('n > 'm, 'm <= 'm).
        (atom<>'n, atom<>'m) -> bit
(gt_vec_signed) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
        (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit

<= :
    (lteq) forall Nat 'n, Nat 'm, flow_constraints('n <= 'm, 'n > 'm).
        (atom<>'n, atom<>'m) -> bit
(lteq_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
        (vector<'n, 'm, 'ord, bit>, atom<o>) -> bit
(lteq_range_vec) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.

34
(atom<'o>, vector<'n, 'm, 'ord, bit>) -> bit
(lteq_vec forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
  (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit

<=_s :
  (lteq_signed) forall Nat 'n, Nat 'm, 'n <= 'o, 'm <= 'p.
  (atom<'m>, atom<'n>) -> bit
(lteq_vec_signed) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
  (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit

>= :
  (gteq) forall Nat 'n, Nat 'm, Nat 'o, Nat 'p, 'n >= 'o, 'm >= 'p.
  (range<'n, 'm, range<o, 'p>) -> bit
(gteq_vec) forall Nat 'n, Nat 'o, Nat 'p, Order 'ord.
  (vector<'o, 'n, 'ord, bit>, vector<'p, 'n, 'ord, bit>) -> bit
(gteq_vec_range) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
  (vector<'n, 'm, 'ord, bit>, atom<o>) -> bit
(gteq_range_vec) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord.
  (atom<o>, vector<'n, 'm, 'ord, bit>) -> bit

(* oddments *)
is_one : (is_one)
  bit -> bit
signed : (signed) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o >= -2**'m, 'o <= 2**'m-1.
  vector<o, 'n, 'ord, bit> -> atom<o>
unsigned : (unsigned) forall Nat 'n, Nat 'm, Nat 'o, Order 'ord, 'o >= 0, 'o <= 2**'m-1.
  vector<o, 'n, 'ord, bit> -> atom<o>
ignore : forall Type 'a.
  'a -> unit
mask : (mask) forall Type 'a, Nat 'n, Nat 'm, Nat 'o, Nat 'p, Order 'ord, 'm >= 'o.
  vector<o, 'n, 'ord, 'a> -> vector<'p, 'o, 'ord, 'a>

to_vec_inc : forall Type 'a.
  nat -> 'a

to_vec_dec : forall Type 'a.
  nat -> 'a

(* option type constructors *)
Some : forall Type 'a.
  'a -> option<'a>
None : forall Type 'a.
  unit -> option<'a>

(* list operations *)
append : forall Type 'a.
  (list<'a>, list<'a>) -> list<'a>

35
8  Tips for writing Sail specifications

This section offers advice for writing Sail specifications that will work well with the Sail interpreter and shallow embeddings.

- Declare memory access functions as one read, one write-address-announce, and one write-value for each kind of access.

For basic user-mode instructions, there should only be a need for one memory read and two memory write function. These should each be declared using `val extern` and should have effects `rmem`, `eamem`, and `wmv`.

A memory read function will typically take as parameters a `size` (a number of bytes, e.g. less than or equal to 32) and an address and return a bit vector with length $8 \times size$. The sequential and concurrent interpreters both only read and write memory in vectors of bytes.

- Declare a default vector order. Vectors can be either decreasing or increasing, i.e. if we have a vector $a$ with elements [1,2,3] then in an increasing specification the 1 is accessed with $a[0]$ but with $a[2]$ in a decreasing system. It is useful to have `default Order inc` or `default Order dec` early in a specification. The default default is increasing.

- Vectors don’t necessarily begin indexing at 0 or ‘n-1. Without any additional specification, a vector will begin indexing at $0$ in an increasing spec and ‘n-1 (where ‘n is its length) in a decreasing specification. A type declaration can reset this first position to any number.

Importantly, taking a slice of a vector does not reset the indexes. So if $a = [10,20,30,40]$ in an increasing system, the slice $a[2..3]$ generates the vector [30,40] and the 30 is indexed at position 2 in either vector.

- Be precise in numeric types. While Sail includes very wide types like `int` and `nat`, for bounds checking, numeric operations, and general clarity, if you know that a number in the specification will range only between (say) 0 and 31, it is better to use a specific range type such as `[|31|]`. If you don’t know the range precisely, it may also be best to leave Sail’s type resolution to infer the bounds in a particular use.

- Use bit vectors for registers. Statically, the Sail language will allow a register to store a value of any type, but the Sail interpreter expects that it is simulating a machine where all registers are bit vectors. Given a bitvector of length one, such as $a$, one can read the element of $a$ either with $a$ or $a[0]$ (by inferred coercions).

- Have functions named `decode` and `execute` to evaluate instructions. The Sail interpreter is hard-wired to expect functions with these names.

- Type annotations are necessary to read the contents of a register into a local variable. The code $x := \text{GPR}[4]$, where GPR is a vector of general purpose registers, will store a local reference to the fourth general purpose register, not the contents of that register, i.e. this will not read the register. To read the register contents into a local variable, the type is required explicitly so `(bit[64]) x := \text{GPR}[4]$ reads the register contents into $x$. The type annotation may be on either side of the assignment.

9  Dynamic semantics: the Sail interpreter

A Sail definition can be executed by running the Sail interpreter (which is essentially a small-step operational semantics, written in Lem) over the Lem deep embedding of the definition. Typically both of these would be translated (by Lem) into OCaml code, which can then be compiled.

The Sail interpreter is defined in the `.lem` files in `sail/src/lem_interp`. It has the interface defined in `interp_interface.lem`, in which the main functions are:
(* decode opcode to instruction *)
val decode_to_instruction : context -> maybe (list (reg_name * register_value)) -> opcode
  -> instruction_or_decode_error

(* construct initial interpreter state from instruction *)
val instruction_to_istate : context -> instruction -> instruction_state

(* execute interpreter to the next outcome *)
val interp : interp_mode -> instruction_state -> outcome

Here outcome is a type giving the next observable outcome together with the next instruction state or
instruction-state continuation. The semantics of an instruction in isolation is effectively a labelled transition
system with these labels.

type outcome =
  (* Request to read N bytes at address *)
  (* The register list, used when mode.track_values, is those that the address depended on *)
  | Read_mem of read_kind * address_lifted * nat * maybe (list reg_name) * (memory_value -> instruction_state)

  (* Request to write memory *)
  | Write_mem of write_kind * address_lifted * nat * maybe (list reg_name)
  * memory_value * maybe (list reg_name) * (bool -> instruction_state)

  (* Tell the system a write is imminent, at address lifted tainted by register list, of size nat *)
  | Write_ea of write_kind * address_lifted * nat * maybe (list reg_name) * instruction_state

  (* Request to write memory at last signaled address. Memory value should be 8* the size given in Write_ea *)
  | Write_memv of maybe address_lifted * memory_value * maybe (list reg_name) * (bool -> instruction_state)

  (* Request a memory barrier *)
  | Barrier of barrier_kind * instruction_state

  (* Tell the system to dynamically recalculate dependency footprint *)
  | Footprint of instruction_state

  (* Request to read register, will track dependency when mode.track_values *)
  | Read_reg of reg_name * (register_value -> instruction_state)

  (* Request to write register *)
  | Write_reg of reg_name * register_value * instruction_state

  (* List of instruction states to be run in parallel, any order*)
  | Nondet_choice of list instruction_state * instruction_state

  (* Escape the current instruction, for traps, some sys calls, interrupts, etc. Can optionally
  provide a handler. The non-optional instruction_state is what we would be doing if we’re
  not escaping. This is for exhaustive interp *)
  | Escape of maybe instruction_state * instruction_state

  (*Result of a failed assert with possible error message to report*)
  | Fail of maybe string

  (* Stop for incremental stepping, function can be used to display function call data *)
  | Internal of maybe string * maybe (unit -> string) * instruction_state

  (* Analysis can lead to non_deterministic evaluation, represented with this outcome *)
  (*Note: this should not be externally visible *)
  | Analysis.non_det of list instruction_state * instruction_state

  (*Completed interpreter*)

37
Additionally, the interpreter can be run in an exhaustive mode, tracking register dependencies, to analyse
the pending register and memory footprint:

(* Run the interpreter without external interaction, feeding in Unknown
on all reads except for those register values provided *)

val interp_exhaustive : maybe (list (reg_name * register_value)) -> instruction_state -> list event

10 Dynamic semantics: the shallow embeddings

The shallow embeddings translate a Sail definition to one which is directly executable.

10.1 Sail-to-Lem translation

In order to map Sail definitions to Lem definitions expressions using Sail features not natively supported by
Lem are rewritten in Sail-to-Sail transformations that avoid these features. There are three main Sail-to-Sail
transformations:

1. elimination of vector-concatenation patterns,

2. separating pure expressions from effectful expressions, and

3. eliminating of imperative local variable updates.

These work roughly as follows:

1. Vectors are implemented essentially as lists of bit in the Lem embedding. Since Lem has no list
concatenation patterns Sail vector concatenation patterns are transformed into standard list patterns.
The high-level steps implemented by this transformation (rewrite_exp_remove_vector_concat_pat) are
these:

   • for convenience remove P_typ type annotations from the patterns
   • introduce fresh names for all P_vector_concat patterns using P_as unless they are already named
   • introduce fresh names for all arguments to P_vector_concat patterns (the vectors patterns that
     are being concatenated) unless they are already named (P_as or P_id)
   • remove all P_as names from the pattern expression and bind the name of a P_vector_concat
     “child” node to an expression selecting the subvector of the P_vector_concat root (using the
     vector length information): ((0b00 as a) : ((0b01 as b1) : ([x] as b2) as b)) as v becomes
     0b00 : (0b01 : [x]) together with the let-bindings

     let a = v[0..1] in
     let b = v[2..length a - 1] in
     let b1 = b[0..1] in
     let b2 = b[2..length b - 1] in
     < body >,

     forgetting the names that no other let bindings or the body refer to.
   • flatten nested vector concatenation patterns: 0b00 : 0b01 : [x]
   • expand the vector-concatenation pattern into a list pattern\(^{33}\), putting wild cards where the pattern
     is unconstrained 0::0::0::1::x

\(^{33}\) this is done by misusing the P_vector_concat to contain elements that are consed onto each other to map onto list cons patterns
2. Imperative Sail expressions will be mapped to monadic Lem expressions. In order to do that, (globally and locally) effectful expressions have to be separated from pure expressions in places where monadic expressions are not allowed, for example in the arguments to standard function applications. This transformation “pulls” effectful terms into separate let-bindings that will then be mapped onto monad bindings: e.g. \( f(GPR[21]) \) will be mapped to \( \text{plet } w\_1 = GPR[21] \text{ in } f(w\_1) \) for some function \( f \) where \( \text{plet} \) is using the \( \text{E\textunderscore internal\textunderscore plet} \) AST node that will be translated to the monad bind.

The continuation-passing style code of this transformation (\texttt{rewrite\_defs\_letbind\_effects}) is very similar to code used for \textit{A-normalisation}: \url{http://matt.might.net/articles/a-normalization}. In the original code a predicate, here called \texttt{value}, determines when expressions are atomic. Here this is adapted so almost all pure expressions are considered to be values to try stay as close to the original definitions as possible.

3. Since all Lem expressions are pure, local imperative variable updates cannot be mapped directly to Lem. Here there are two choices:

- implement local state as part of a state monad (L3 does this)
- implement local state by (repeatedly) let-binding local variable names.

The shallow embedding uses the latter, \( x := v; e \) becomes \texttt{let } x = v \text{ in } e. \) The function \texttt{rewrite\_var\_updates} implements this transformation. After the previous transformation effectful programs consist of sequences of let-bindings \texttt{let } w\_1 = e' \text{ in } e for effectful terms \( e' \) and “control flow” expressions: \( \text{E\textunderscore if}, \text{E\textunderscore case}, \text{ and E\textunderscore for}. \) The former is translated to \texttt{let } x = v \text{ in case } e' is an update of \( x \) to \( v \), for the latter the transformation looks up which local mutable variables are updated in the if-branches/the different cases/the fo-loop body and makes these “control flow” expressions return the updated values of those variables.

After this transformation the let-expressions implementing local variable updates are translated to Lem let-expressions; if-expressions and case-expressions are mapped onto their Lem counterparts and for-loops are translated to function applications of a pre-defined \texttt{foreach} function that takes two arguments: the loop bounds and the loop body as a function from the loop variable.

10.2 Generated code

After these transformations the Sail code only uses features that are directly supported by Lem. Where the interpreter uses a universal value type the shallow embedding has

- a three-valued bit type: zero, one, and undefined
- a polymorphic vector type consisting of a list of elements, a start index, and the index direction
- mathematical integers and natural numbers (almost all numbers are currently mapped to integers)
- a type for registers consisting of a (string) name, length, start index, index direction, and a list of its register fields.

The definition of these types and the library functions working over them is in the file \texttt{sail\_values.lem}.

The shallow embedding of Sail into Lem is parametric in the monad; there are two different interpretations of the primitive monad functions and the monad return and bind operators:

- a non-deterministic state and exception monad where register and memory accesses are mapped to actions on a register state and flat memory
- an instance of the free monad using the type “outcome”:

\begin{verbatim}
  type outcome 'a =
  (* Request to read memory at a given location and size *)
  Read_mem of (read_kind * address_lifted * nat) * (memory_value -> outcome 'a)
  (* Request to write memory at some location and for a particular size *)
\end{verbatim}
Write_ea of (write_kind * address_lifted * nat) * outcome 'a
(* Request to write memory at the last signalled address *)
Write_memv of memory_value * (bool -> outcome 'a)
...
(* Proper value of type 'a *)
| Done of 'a

The return operator wraps values into Done, the monad return composes the continuations in the last argument of the different outcomes.

(The full definitions can be found in the State and Prompt modules.) For a sequential model the state monad interpretation is standard, the monad using the outcome type is more general: it can be used with any interpreter for the outcomes/effects, for ppcmem it is the interface between concurrency model and instruction semantics.

Each Sail ISA document declares memory access functions and barrier functions as external, giving only the type specification. Therefore, as with the deep embedding each architecture must give an implementation of the memory access and barrier functions written in Lem, for the shallow embedding in terms of the primitive monad functions.

11 Additional potential changes, with hindsight

Additional things we would change about Sail in hindsight:

- Require all variables to be declared, probably with their type. Unless those would be too unfamiliar for engineers, for size-polymorphic things – maybe make syntax more C++ template/Java Generic-like? The forall's force variables to be declared with their kinds, which helps.
- Syntactically separate the register-as-value and implicit-dereferencing cases.
- The coercions add a lot of complexity, but whenever we tried to remove one, it made one or more of the specifications ugly and users complained.
- All vectors have lengths and starting indices, but often you don’t care about the indexing.
  When do you get a non-canonical (zero for inc, length-1 for dec) start index from an operation?
    - slicing results in the original indexing but sliced
    - explicit type annotation
  We could add a don’t-care for the start index, to avoid having to have boring noisy type variables.
- The parser was annoying to develop, principally because it’s C-like. We have to know what identifiers are introduced as type identifiers to properly parse, so we need to be two-phase. And there’s a very strong similarity between nexp's and expressions. Tuple types used to use * and so did Nat multiplication; that was very awkward. Now tuple types use commas (,), so this might be easier. Using parentheses to introduce types of identifiers is awkward. The parser implementation is two lexers and two parsers, both using ocamllex/ocamlyacc. We used menhir for debugging, but (though historical accident) didn’t for the main implementation.

12 Repository history

Development of Sail started in June 2013 in the bitbucket l2 repository, based on earlier notes in rsem/idl. Later we renamed the l2 repo to sail, in 2016/6/3. A few files came from Lem: util.ml and reporting_basic.ml.
13 Conversion from ASL

13.1 Definition patching

In some cases we need to patch generated Sail. The basic flow is:

sail source
--parser-->
parse_ast
initialcheck-->
NoTyp-annotated ast
--type_check-->
type-annotated ast
--rewriter (for the various backends)-->

Here initialcheck checks kinds, Nat vs Type, etc. (the parseAst has one node for all kinds), checks and collapses scattered definitions.

In asl_parser, asl_to_sail generates NoTyp-annotated AST, and sail_to_sail works over that, then we use initial_check_full to check kinds etc. (producing NoType-annotated AST again).

Patches are in arm_replace.sail. They are processed in the asl_parser main.ml. The patcher calls the parser and initial check, and feeds that back into sail_to_sail, operating over the NoTyp-annotated AST. That’s fine for a full function, but scattered definitions have already been collapsed. This is stopping us insert implementation-defined instructions for thread-create etc.

The initial_check_full code is a mutated copy of the initial_check code. De-scattering could be pulled back into common.

13.2 Missing constraints and constraint inference

In scattered clauses, we need to support some path-specific constraints that we don’t currently have.

The ARMv8-A (hand) model uses ‘regsize IN {32,64} and ‘datasize IN {8,16,32,64,128}, consistently.

In the ASL, there is a regsize, datasize, and destsize. In some instructions, regsize is {32,64}, while others have {32,64,128}. In some, datasize is {8,16,32,64,128}, while others have {32,64}. In some, destsize = datasize, while in others they differ.

Really we want, in the execute header:

    scattered function forall Nat ‘datasize, Nat ‘destsize, Nat ‘regsize . unit execute
to put set constraints on each, and then in some of the execute scattered clauses, to be able to tighten those constraints, e.g. for

    execute (integer_arithmetic.addsub.immediate

to say ‘datasize IN {32,64}. Or have the ast completely generic and set contraints on each clause – that may be easier, as we don’t have mechanism to intersect two set constraints.

In the ARMv8 (hand), the ast and execute clause for each instruction are tied together with singleton range types. For ARMv8 (ASL), we’re currently not doing that; we need to mine more information from the asserts in the ASL, e.g.

    function forall Nat ‘width . bit[*‘width] read_X ([31]) n =
    {
        (*assert(n >= 0 & n <= 31)*) ();
        (*assert(width == 8 | width == 16 | width == 32 | width == 64)*) ();
        if n != 31 then (read_R(n))[sizeof ‘width - 1 .. 0] else Zeros_with_N(sizeof ‘width)
    }

In the ASL, read_R reads the full 64-bit value from a register, while read_X calls read_R and then slices out a subvector, based only on the expected type at the read_X callsite.
function clause execute (integer_arithmetic_addsub_immediate
   (([[31]]) d, ([:'datasize:]) datasize, (bit[:'datasize]) imm,
   (([[31]]) n, (bool) setflags, (bool) sub_op)) =
   {
      (bit[:'datasize]) result := undefined;
      (bit[:'datasize]) operand1 := if n == 31 then read_SP() else read_X(n);
   }

In the ASL, the sizes at which things are actually used can in principle be determined, eg from the
integer_arithmetic_addsub_immediate_decode
   ([:'datasize:]) datasize := if sf == 0b1 then 64 else 32;

Sail will currently infer that 'datasize is in the range 32..64, not in the set \(\{32, 64\}\). It’s not clear how to
choose locally between sets and ranges.

14 Internals: the Sail formal type system and typechecker

The Sail type inference rules are defined as a collection of inductive relations in the Ott source files, which
correspond closely to the implemented type inference code. point to typeset version when that’s cleaned up

The type rules model what the code does – the top-level definition does inference and rewrites the
expression, making it fully type-annotated, fully coercion-inserted, and with overloading resolved. It involves:

congs_to: roughly checking whether the dependent-type-erasure of two types are the same (it does
look at vector lengths iff they are constants). This is the closest thing to a type equality relation. It’s not
symmetric: reg<t> conforms to t but not vice versa. And register<t> conforms to t but not vice versa.
Primarily used for overloading resolution and for typing conditionals. This is conformance up to possible
coections, though it does not insert coecions.

consistent_with: this is roughly a subtyping relation, eg [:3:] (aka atom<3>) is consistent with [|2 : 5 |].
It produces constraints. consistent_with is not up to possible coecions It does not permit instantiation of
Type variables.

correctes_to: given an expression with its expected type and its “natural” type, this finds the possible
coections between those types, inserts one (there should be at most one), and calls consistent_with for the
new expression and expected type. It should always succeed but might generate some constraints (which
might not end up satisfiable). If there are no coecions between the types, it just calls consistent_with.
This is used for every expression.

For most expression forms, the use of the above is straightforward: we just call correctes_to on the natural
and expected type.

For function calls, if the function is not overloaded (only the built-in functions in type_internal.ml:initial_typ_env
can be overloaded), it’s simple: we find the stated “most general” type from initial_typ_env. We use that
to see if the natural types of the parameters conform. Then we go through the variants.

• Case: no overloading on return type. Find all the variants that consor_to the parameter types. If
   there is one, do correctes_to for each parameter and the function argument/result type. If there is more
   than one or zero, fail.

• Case otherwise: Do the above, but if there were more than one, see how many of the return types
   correctes_to our expected return type. If one, we are done (and do correctes_to), otherwise fail.

The formal rules are syntax-directed. If one did inductive search over them, that *should* do the same
as the implementation.

Typechecking expressions just builds up a set of constraints. When we have finished checking a function or
top-level let, we try to resolve them. Constraint solving is in type_internal.ml:resolve_constraints. The
formal type rules don’t say how constraint resolution is done (they appeal to a resolve-constraints function,
not defined in the rules), or even record where path-aware constraints get generated. We did not try using
an external solver: (a) we’re ok leaving constraints somewhat loose, which off-the-shelf solvers seemed not to
not do, and (b) we didn’t see a way to represent path-aware constraints (BranchCons and CondCons).

If Nat, Effect, and Order annotations were erased, the type inference system would be similar to standard
Hindley-Milner except with extensions for overloading and type coercions. To handle those,expression terms
are rewritten during type checking to include coercions and to specify the resolved overloaded functions. Expression terms are also annotated with their inferred types for type checking after inference. However, we use a top-down inference resolution so that val specifications and type annotations can be used to identify types during inference and improve the locality of error messages. We treat registers as a special value within Sail, which can be stored locally, or read or written to externally.

Overloading is only permitted within the standard library. Each overloaded function has a general type specification and can be overloaded in either the input type, return type, or both; however, the variants must all have the same number of parameters. When an overloaded function is called, the parameters are first checked against the most general types for the function, which may fail. If they succeed, then the possible overloading are checked against the parameter types of the actual arguments using the conforms_to relation, up to widening but not coercion. Widening relates to the size of numeric values or the starting position of vectors; coercions are discussed below. If only one function relates via the conforms_to relation on the parameters, then this is the function option used to annotate the expression and check the return type. When multiple function options match, and the overloading does not permit overloading on the return type, then an error occurs; otherwise the expected type is checked against the conforms_to relation of the declared return types. Again, if none match, then there is no overloaded function applicable.

Coercion is not permitted within overloaded function selection, to clarify which function should be selected. For instance, if a function \(+\) were defined over vectors, numbers, and a combination of vectors and numbers, it may be unclear to the programmer which function would be used if vectors and numbers were also being coerced to each other, and the specificity of using a number instead of a vector or vice versa could be lost.

The most common coercion in Sail specifications is that from vectors to numbers, particularly used in using an opcode fragment to access a register, or a portion of a register via index. This idiom of programming led us to include coercions, to unclutter the specifications. Similarly we coerce between single bit vectors and bits and vice-versa, vectors and numbers, numbers and vectors with a specific type annotation (we do not coerce these generally due to a potential loss of information within the number), from registers to the contents of the register (which adds a register read effect), from registers to the contents of registers, and to and from enumeration constants and numbers.

Because coercions are not symmetric, we do not have true type equality within the Sail system. Instead we have three type relations, conforms_to, which does not consider Nats in all circumstances and considers whether one type could be coerced to the other in the proper direction, consistent_to, which considers two types consistent if they would conform to each other and can generate consistent constraints, to be discussed below, and coerces_to, which relates two types and two expressions when an inserted coercion would make the two types consistent; when two types are already consistent, the coercion is the identity coercion (not actually applied).

The conforms_to and consistent_to relations are used within the type inference in locations where we will not be adding coercions, such as within pattern matching, when resolving overloading, etc. These can then be flagged to consider conformance without allowing for coercions. Additionally, loose conformance will not look at numbers whereas strict conformance will use a strict equality on numeric kinds.

Special constraints are also generated to indicate that a function may rely on context for a Nat parameter of a return type, be it a vector or register length. The type system adds an explicit parameter for these implicit ones at all call sites, relying on Nat unification and constraint resolution to place a proper value into the call based on information at the context site. These can nest, so that one implicit parameter feeds into the next and is resolved to a final value by a later call site. This places a slight strain on Nat resolution, as Nat variables that are used within an implicit declaration must not be resolved into equations; it is an error to resolve one to a constant except under case analysis.

14.1 Nat constraints

Vector accesses natively put constraints on the vectors and numbers involved in the access; function calls also put size constraints on variables and expressions. Other constraints arise from these constraints and whether two types are consistent relation may require that certain additional constraints be met for the program to be well typed. Programmers may declare constraints on functions and data types. All constraints are deemed to be anded together.

The set of possible constraints includes equality, greater than, less than, combinations of these, and
set inclusion (where a variable may be stated to be within a set of constants). Internally there are more
constraints including inequality, path-aware constraints, to be discussed below, and predicate constraints,
which are placed only on functions with a bit return that should be considered as a predicate, where one
set of constraints holds on positive paths and one on negative paths. Additionally constraints may be
requirements or guarantees on numeric behaviour, also discussed below.

The constraint language is not in general decidable. However, undecidability has not been an issue
in practice for the ISA specifications presently represented in Sail. Instruction set specifications tend to
use relatively straightforward idioms for bit vector sizes and indexing, even though the operations that
the instructions are representing may themselves be quite complex (i.e. division or cryptographic instructions).

Constraints are resolved after inferring a type for a function, which uses coercion and consistency checking
to derive a set of constraints that must hold for the function to be correctly typed. This resolution considers
simple constraints, set-bounded constraints, path-aware constraints, and guarantee-require constraints. We
treat each of these resolutions separately below.

14.1.1 Constraint solving in the simple case

Although complex constraints do arise in the specifications, such as \( y \times z < 2^x - 1 \), these do not occur often.
The vast majority of the constraints have simple forms, e.g. \( 0 = 0 \), \( 64 = 64 \), \( x = 31 \land x < 32 \). We begin
resolution by first putting all \texttt{nexp}s into a normal form, collapsing constant arithmetic, setting all variable
multiplications to be \texttt{constant \ast variable}, changing subtractions to multiplications by \texttt{-1}, and ordering the
term into a sum of factors sorted by variable creation point, where terms with a power of \( 2 \) appear first in
the terms, and are sorted by their power unless that is a constant. In these a variable appears once in a base
term as a factor, unless two or more variables are multiplied together. We then process these normal forms
with a relatively simple rewriting system that first finds all variables that are equal, which are all unification
variables either introduced by the consistency checking or standing in for programmer declared variables,
and creating equality chains for these variables (e.g. \( a = b = c = d \)).

Then we undertake a rewriting pass over all of the constraints, removing constraints that involve only
constants which hold, and adding constants set to variables in equality chains (so now \( a = b = c = d = 5 \)).
We then iterate to a fixed point of constraints. This process removes more than 90% of the constraints in
our specifications. We further consider simple inequalities to resolve, e.g. \( 2 \times x \geq x \) if we also know that \( x \)
is positive.

14.1.2 Set bounded constraints

When a constraint refines a variables to a set of constants, it is important that the above rewriting not set
an equality chain equal to a value during checking, as this may only be one of the set of constraints and then
an incorrect equality may arise between two or more members of the allowed set. Therefore when a variable
is specified as within a set, we do not terminate an equality chain with a constant or non-constant equation
(i.e. \( 2 + x \)). Instead, we check the constraints using that variable and any in the equality chain with each of
the values in the set, unless under a path constraint described below. If any of these indicate inconsistency,
then there is not a solution for the set.

14.1.3 Path-aware constraints

Let’s start with an example illustrating why one might want path-aware constraints in this domain. make
this example real

\[
\texttt{forall Nat }\ 'n', \ 'n' \texttt{ IN \{8,16,32,64\}.}
\{ \texttt{bit['n] v := 0;}
\texttt{...}
\texttt{switch sizeof 'n}
\texttt{ case 8 -> ... v ... (* v: bit[8] *)}
\texttt{ case 16 -> ... v ... (* v: bit[16] *)}
\texttt{...}
\]

44
if sizeof 'n = 8
then ... v ...  (* v: bit[8]*)
else ... v ...  (* v: bit['k], 'k IN {16,32,64} *)
...

We also use path-aware constraints across if-branches, particularly when the condition used includes a predicate constraint specification.

The sorts of constraints added for path aware constraints, which are internal only, are conditional constraints and branch constraints. A conditional constraint consists of a set of pattern or predicate derived constraints and expression derived constraints. A branch constraint consists of a list of conditional constraints. A branch constraint with one branch resolves into the contained constraint.

To resolve a branch constraint generally, we first generate new unification variables for each path and create a mapping between the original unification variable and the generated ones; we do this only for Nat unification variables which occur in the pattern or predicate portion of conditional constraints along the branch. We then perform the standard constraint resolution for one iteration. At the end of each iteration, we consolidate information that has been accumulated over the branches; this way if a unification variable from outside the branch is resolved into a constant or a computation, then this information is pushed along the branches following the maps.

At the end of the resolution, we perform one additional pass over the variables, pushing each variable that is set entirely to other variables along all paths into one chain. This is necessary to resolve implicit length constraints.

14.2 Require/guarantee constraints

Again let’s start with an example that explains why we need this and what we mean by require vs guarantee. A good one is an int index into a vector maybe? This references contract work. Variables in the inputs of a function are requires, others are guarantees, most are guarantees. Vector accesses create requirements, most library functions create guarantees. On equality constraints, there’s no such thing; equality is equality.

For resolving inequality constraints, we create per-variable constraints, i.e., constraints which all have a variable isolated on one side as much as possible. So with a constraint such as $5 \times x + y > 0$ we get the constraints $5 \times x > -y$ and $y > -5 \times x$. We don’t remove the 5 on the $5 \times x > -y$ formula because we don’t support rational numbers within the nat expressions and thus leave it alone.

We then consider all of the constraints which contain a particular variable and consolidate all of the greater than and less than constraints into maximum and minimum guaranteed vs required ranges. Then we check for consistency across the constraints for one variable. Provide example here of the infinity vs 31 check in the ARM specs. When a constraint contains multiple variables where one can be sufficiently constrained, we can check whether the variables ranges when combined produce consistent results. However, this impacts less than a percent of the constraints we’ve seen in the specs to date.

14.3 Putting it all together

Constraints unresolved by the previous methods are saved for each function. These are checked for consistency against any specified constraints for the function definition; equality here is insufficient because it would be reasonable for a function to declare that a result would be between 0 and 100 when the function actually provides a result between 5 and 20. If none are declared then the function will carry any unresolved constraints for each use, where the unification variables are turned into standard variables.

14.4 Checking vs Inference

The above describes the type inference process. Post inference we can in principle type check the annotated terms, and there is incomplete code to do that. This is fairly straightforward, with the primary task being to recall the Nat variables and assign them to dynamic values from the call sites, which uses the specified and generated constraints to identify the role of the variables. Think, would connections between variables and
nat vars still be useful? Yes, so we need to specify how we generate them. Then the constraint checking is a
simple variant of the previous one for inference. I think.

Constraint resolution here is a simplified form of the one above, as we have dynamic values instead of
potential future values in more cases, so we can look at more constraints of the form \(2 \leq 31 \) and \(0 \leq 2\) rather
than the same constraints on \(x\).

This is mostly but not completely implemented, as a typechecker written in Lem that operates over fully
type-annotated expressions (bitbucket/sail/src/lem_interp/type_check.lem, not to be confused with the
other type_check.ml file). It considers the natural type, declared type, and expected type of an expression,
checking they are pairwise consistent with in that order. It doesn’t need a version of coerces_to.

In the implemented part, consistent_with doesn’t generate new constraints; it checks the existing ones.
Two things are missing:

- something to provide the type environment of the whole definition; and
- the rest of the expression cases.

This should be usable for typechecking the result of the existing type inference, and for checking the result
after each step of the sequential interpreter.

The interpreter has disjoint expression and value types. To make a typecheckable expression from an
interpreter state, we need to map values back into expressions. Mostly that’s fine, but not where there are
data constructors whose types are parameterised type constructors. (Power doesn’t have those, as its ast
nodes don’t have parameters, so we might be closer to testing this for Power. ARM has regsize and destsize
parameterisation).

A  Sail syntax

The Sail syntax is defined in Ott [24, 25], which also generates the OCaml AST types used in the implementa-
tion. The implementation parser is a hand-written menhir parser. The full grammar is below, with internal
and deprecated nodes (tagged I and D in the Ott source) elided. Productions tagged S are syntactic sugar.

A.1 Metavariables

| n, m, i, j          | Index variables for meta-lists |
| num, numZero, numOne| Numeric literals               |
| hex                 | Bit vector literal, specified by C-style hex number |
| bin                 | Bit vector literal, specified by C-style binary number |
| string              | String literals                |
| regexp              | Regular expressions, as a string literal |
| x, y, z             | identifier                     |
| ix                  | infix identifier               |

A.2 Grammar

| i                          | source location |
|                           |                 |
| annot                     |                 |
| id                       | identifier |
|                           | x                      |
|                           | bool M               | built in type identifiers |
|                           | bit M                |
|                           | unit M               |

\[^{34}\]The id grammar includes a deinfix construct, which has not been used and should probably be removed.
nat M
string M
range M
atom M
vector M
list M
reg M
to_num M
to_vec M
msb M

built-in function identifiers

kind ::= 'x

kind ::=
  Type kind of types
  Nat kind of natural number size expressions
  Order kind of vector order specifications
  Effect kind of effect sets

base_kind ::= base_kind1 -> ... -> base_kindn

kind ::=
  base_kind1 -\rightarrow ... -\rightarrow base_kindn

nexp ::= id abbreviation identifier
  kid variable
  num constant
  nexp1 * nexp2 product
  nexp1 + nexp2 sum
  nexp1 - nexp2 subtraction
  2** nexp exponential
  ( nexp ) S

order ::= kid variable
  inc increasing
  dec decreasing
  ( order ) S

base_effect ::= rreg read register
  wreg write register
  rmem read memory
  wmem write memory
  wmea signal effective address for writing memory
  wmv write memory, sending only value
  barr memory barrier
  depend dynamic footprint
  undef undefined-instruction exception
  unspec unspecified values
  nondet nondeterminism, from nondet
  escape potential call of exit

47
local mutation; not user-writable
local return; not user-writable

\[ \text{effect} ::= \]
\[ \text{kind} \]
\[ \{ \text{base\_effect}_1, \ldots, \text{base\_effect}_n \} \]
\[ \text{pure} \]
\[ \text{effect}_1 \uplus \ldots \uplus \text{effect}_n \]

\[ \text{effect set, of kind Effect} \]

\[ \text{effect set} \]
\[ \text{M} \]
\[ \text{sugar for empty effect set} \]
\[ \text{M} \]
\[ \text{union of sets of effects} \]

\[ \text{typ} ::= \]
\[ \_ \]
\[ \text{id} \]
\[ \text{kind} \]
\[ \text{typ}_1 \rightarrow \text{typ}_2 \text{effect effect} \]
\[ (\text{typ}_1, \ldots, \text{typ}_n) \]
\[ \text{id} < \text{typ\_arg}_1, \ldots, \text{typ\_arg}_n > \]
\[ (\text{typ}) \]
\[ [\text{nexp}] \]
\[ [\text{nexp} : \text{nexp}'] \]
\[ [: \text{nexp}] \]
\[ \text{typ}[\text{nexp}] \]
\[ \text{typ} [\text{nexp} : \text{nexp}'] \]
\[ \text{typ} [\text{nexp} <: \text{nexp}'] \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]
\[ \text{S} \]

\[ \text{typ\_arg} ::= \]
\[ \text{nexp} \]
\[ \text{typ} \]
\[ \text{order} \]
\[ \text{effect} \]

\[ \text{typ\_arg} \]
\[ \text{type constructor arguments of all kinds} \]

\[ \text{n\_constraint} ::= \]
\[ \text{nexp} = \text{nexp}' \]
\[ \text{nexp} \geq \text{nexp}' \]
\[ \text{nexp} \leq \text{nexp}' \]
\[ \text{kind IN} \{\text{num}_1, \ldots, \text{num}_n\} \]

\[ \text{n\_constraint} \]
\[ \text{constraint over kind Nat} \]

\[ \text{kinded\_id} ::= \]
\[ \text{kind} \]
\[ \text{kind kid} \]

\[ \text{kinded\_id} \]
\[ \text{optionally kind-annotated identifier} \]
\[ \text{id} \]
\[ \text{kind-annotated variable} \]

\[ \text{quant\_item} ::= \]
\[ \text{kinded\_id} \]
\[ \text{n\_constraint} \]

\[ \text{quant\_item} \]
\[ \text{kinded identifier or Nat constraint} \]
\[ \text{optionally kinded identifier} \]
\[ \text{Nat constraint} \]

\[ \text{typquant} ::= \]
\[ \text{forall quant\_item}_1, \ldots, \text{quant\_item}_n . \]

\[ \text{typquant} \]
\[ \text{type quantifiers and constraints} \]
\[ \text{empty} \]

\[ \text{typschm} ::= \]
\[ \text{typquant typ} \]

\[ \text{typschm} \]
\[ \text{type scheme} \]
### Name Scm Opt

- **name**
- **name** ::=
- [ name = regexp ]

### Type Def

- **type def** ::= type definition body
- **typedef** id name opt = typschm
- **typedef** id name opt = const struct typquant { typ1 id1 ; ... ; typn idn }?
- **typedef** id name opt = const union typquant { type_union1 ; ... ; type_unionn }?
- **typedef** id name opt = enumerate { id1 ; ... ; idn }?
- **typedef** id = register bits [ nexp : nexp ] { index_range1 ; id1 ; ... ; index_range_n ; idn }

### Kind Def

- **kind def** ::= Definition body for elements of kind
- **Def** kind id name opt = nexp

### Type Union

- **type union** ::= type union constructors
- **id**
- **typ id**

### Index Range

- **index range** ::= index specification, for bitfields in register types
- **num**
- **num1 .. num2**
- **index_range1 , index_range2**

### Lit

- **lit** ::= literal constant
- ()
- bitzero
- bitone
- true
- false
- num
- hex
- bin
- string
- undefined

### ;?

- **;?** ::= optional semi-colon

### Pat

- **pat** ::= pattern
- **lit** literal constant pattern
- **_**
- ( pat as id ) named pattern
- ( typ ) pat typed pattern
- **id** identifier
- **id ( pat1 , .. , patn )** union constructor pattern
\[ \begin{align*}
\text{lexp} & ::= \\
\text{fpat} & ::= \\
\text{exp} & ::= \\
\text{switch} & \text{ exp } \{ \text{ case pexp}_1 \ldots \text{ case pexp}_n \} \\
\text{letbind in} & \text{ exp } \\
\text{lexp} & ::= \text{ exp } \\
\text{sizeof} & \text{ nexp } \\
\text{return} & \text{ exp } \\
\text{exit} & \text{ exp } \\
\text{assert} & \{ \text{ exp } , \text{ exp}' \} \\
\text{ ( exp )} \\
\end{align*} \]

\[
\text{fpat} ::= \text{id = pat} \\
\text{exp} ::= \{ \text{ exp}_1 , \ldots , \text{ exp}_n \} \\
\text{nondet} \{ \text{ exp}_1 , \ldots , \text{ exp}_n \} \\
\text{id} \\
\text{lit} \\
\text{ ( typ ) exp} \\
\text{id \{ \text{ exp}_1 , \ldots , \text{ exp}_n \} } \\
\text{id exp} \\
\text{exp_1 \ id exp_2} \\
\text{ ( exp_1 , \ldots , \text{ exp}_n )} \\
\text{if \ exp_1 \ then \ exp_2 \ else \ exp_3} \\
\text{if \ exp_1 \ then \ exp_2} \\
\text{foreach} \ ( \text{id from exp_1 to exp_2 by exp_3 in order }) \text{ exp_4} \\
\text{foreach} \ ( \text{id from exp_1 to exp_2 by exp_3}) \text{ exp_4} \\
\text{foreach} \ ( \text{id from exp_1 downto exp_2 by exp_3}) \text{ exp_4} \\
\text{foreach} \ ( \text{id from exp_1 downto exp_2}) \text{ exp_3} \\
\text{id exp} \\
\text{exp_1 , \ldots , \text{ exp}_n } \\
\text{ [ num_1 = exp_1 , \ldots , num_n = exp_n , opt_. default ]} \\
\text{exp [ exp’ ]} \\
\text{exp [ exp_1 , \ldots , exp_2]} \\
\text{ [ exp with exp_1 = exp_2 ]} \\
\text{ [ exp with exp_1 : exp_2 = exp_3 ]} \\
\text{exp : exp_2} \\
\text{ [ [ exp_1 , \ldots , exp_2 ] ]} \\
\text{exp_1 :: exp_2} \\
\text{ ( fexp ) \{ \text{ exp with fexp } } \\
\text{exp . id} \\
\text{switch \ exp } \{ \text{ case pexp}_1 \ldots \text{ case pexp}_n \} \\
\text{letbind in} \text{ exp } \\
\text{lexp ::= exp} \\
\text{sizeof \ nexp} \\
\text{return \ exp} \\
\text{exit \ exp} \\
\text{assert} \{ \text{ exp , exp’ } \} \\
\text{ ( exp )} \\
\end{align*} \]

\[
\text{expression} ::= \\
\text{sequential block} \\
\text{nondeterministic block} \\
\text{identifier} \\
\text{literal constant} \\
\text{cast} \\
\text{function application} \\
\text{funtion application to tuple} \\
\text{infix function application} \\
\text{tuple} \\
\text{conditional} \\
\text{loop} \\
\text{vector (indexed from 0)} \\
\text{vector (indexed consecutively)} \\
\text{vector access} \\
\text{subvector extraction} \\
\text{vector functional update} \\
\text{vector subrange update, with vector} \\
\text{vector concatenation} \\
\text{list} \\
\text{cons} \\
\text{struct} \\
\text{functional update of struct} \\
\text{field projection from struct} \\
\text{pattern matching} \\
\text{let expression} \\
\text{imperative assignment} \\
\text{the value of \text{nexp} at run time} \\
\text{return \text{ exp} from current function} \\
\text{halt all current execution} \\
\text{halt with error \text{ exp’} when not \text{exp}} \\
\text{lvale expression} \\
\text{identifier} \\
\text{memory or register write via function call} \]
| id exp |
| ( typ ) id |
| ( lexp0 , .. , lexpn ) |
| lexp [ exp ] |
| lexp [ exp1 .. exp2 ] |
| lexp . id |

S sugared form of above for explicit tuple exp
cast
multiple (non-memory) assignment
vector element
subvector
struct field

ferp ::= id = exp

field expression

fexp ::= id = exp

field expression list

fexps ::= fexp ; ... ; fexpn ;

opt default ::= ; default = exp

optional default value for indexed vector expressions

pexp ::= pat -> exp

pattern match

tannot_opt ::= typquant typ

optional type annotation for functions

rec_opt ::= rec

optional recursive annotation for functions
non-recursive
recursive

effect_opt ::= effect effect

optional effect annotation for functions
sugar for empty effect set

funcl ::= id pat = exp

function clause

fundef ::= function rec_opt tannot_opt effect_opt opt funcl and ... and funcln

function definition

letbind ::= let typeschm pat = exp

let binding
let, explicit type (pat must be total)

let pat = exp
let, implicit type (pat must be total)

val_spec ::= val typeschm id

value type specification
specify the type of an upcoming definition

val externally typeschm id
specify the type of an external function

val externally typeschm id = string
specify the type of a function from Lem

default_spec ::= default Order order

default kinding or typing assumption

default base_kind kid
scattered_def ::= scattered function rec_opt tannot_opt effect_opt id
              | function clause funcl
              | scattered typedef id name scm_opt = const union typquant
              | union id member type_union
              | end id

def_id ::= default typedefm id

reg_id ::= id

alias_spec ::= reg_id . id
             | reg_id [ exp ]
             | reg_id [ exp .. exp' ]
             | reg_id : reg_id'

def_spec ::= register typ id
             | register alias id = alias_spec
             | register alias typ id = alias_spec

def ::= kind_def
      | type_def
      | fundef
      | letbind
      | val_spec
      | default_spec
      | scattered_spec
      | dec_spec

defs ::= def1 .. defn

References


