The Sail instruction-set semantics specification language

Kathryn E. Gray Peter Sewell Christopher Pulte Shaked Flur
Robert Norton-Wright Alasdair Armstrong Thomas Bauereiss

May 11, 2018

Contents

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 2007–2018 we have developed mathematically rigorous models for the allowed architectural envelope of concurrent code, for x86, IBM Power, RISC-V, 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 specification 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 specification, 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).
- A compiled version of the specification directly into OCaml.
- A efficient executable version of the specification, compiled into C code.

Sail does not currently support description of the assembly syntax of instructions, or the mapping between that and instruction AST or binary descriptions, although this is something we plan to add.

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

ARMv8 (hand)	hand-written
ARMv8 (ASL)	generated from ARM's v8.3 public ASL spec
IBM Power	extracted/patched from IBM Framemaker XML
MIPS	hand-written
CHERI	hand-written
RISC-V	hand-written

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 ARM's machine-readable public v8.3 ASL specification[1]. It includes everything in ARM's specification.

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.

[1] ARM v8-A Architecture Specification: https://github.com/meriac/archex

2 A tutorial RISC-V example

We introduce the basic features of Sail via a small example from our RISC-V model that includes just two instructions: add immediate and load double.

We start with some basic type synonyms. We create a type `xlen_t` for bitvectors of length 64, then we define a type `regno`, which is a type synonym for the builtin type `atom`. The type `atom('n)` is a number which is exactly equal to the type variable `atom('n)`. Type variables are syntactically marked with single quotes, as in ML. A *constraint* can be attached to this type synonym—ensuring that it is only used where we can guarantee that its value will be between 0 and 31. Sail supports a rich variety of numeric types, including range types, which are statically checked. We then define a synonym `regbits` for `bits(5)`. We don't want to manually convert between `regbits` and `regno` all the time, so we define a function that maps between them and declare it as a *cast*, which allows the type-checker to insert it where needed. By default, we do not do any automatic casting (except between basic numeric types when safe), but to allow idioms in ISA vendor description documents we support flexible user defined casts. To ensure that the constraint on the `regno` type synonym is satisfied, we return an existentially quantified type `{'n, 0 <= 'n < 32. regno('n)}`.

```
type xlen_t = bits(64)

type regno ('n : Int), 0 <= 'n < 32 = atom('n)

type regbits = bits(5)

val cast regbits_to_regno : bits(5) -> {'n, 0 <= 'n < 32. regno('n)}

function regbits_to_regno b = let r as atom(_) = unsigned(b) in r
```

We now set up some basic architectural state. First creating a register of type `xlen_t` for both the program counter `PC`, and the next program counter, `nextPC`. We define the general purpose registers as a vector of 32 `xlen_t` bitvectors. The `dec` keyword isn't important in this example, but Sail supports two different numbering schemes for (bit)vectors `inc`, for most significant bit is zero, and `dec` for least significant bit is zero. We then define a getter and setter for the registers, which ensure that the zero register is treated specially (in RISC-V register 0 is always hardcoded to be 0). Finally we overload both the read (`rX`) and write (`wX`) functions as simply `X`. This allows us to write registers as `X(r)= value` and read registers as `value = X(r)`. Sail supports flexible ad-hoc overloading, and has an expressive l-value language in assignments, with the aim of allowing pseudo-code like definitions.

```
register PC : xlen_t
register nextPC : xlen_t

register Xs : vector(32, dec, xlen_t)

val rX : forall 'n, 0 <= 'n < 32. regno('n) -> xlen_t effect {rreg}

function rX 0 = 0x0000000000000000

and rX (r if r > 0) = Xs[r]

val wX : forall 'n, 0 <= 'n < 32. (regno('n), xlen_t) -> unit effect {wreg}

function wX (r, v) =
  if (r != 0) then {
    Xs[r] = v;
  }

overload X = {rX, wX}
```

We also give a function `MEMr` for reading memory, this function just points at a builtin we have defined elsewhere. Note that functions in sail are annotated with effects. This effect system is quite basic, but indicates whether or not functions read or write registers (rreg and wreg), read and write memory (rmem and wmem), as well as a host of other concurrency model related effects. They also indicate whether a function throws exceptions or has other non-local control flow (the escape effect).

```
val MEMr : forall 'n. (xlen_t, atom('n)) -> bits(8 * 'n) effect {rmem}

function MEMr (addr, width) =
  match __RISCV_read(addr, width) { Some(v) => v, None() => zeros(8 * width) }
```

It's common when defining architecture specications to break instruction semantics down into separate functions that handle decoding (possibly even in several stages) into custom intermediate datatypes and executing the decoded instructions. However it's often desirable to group the relevant parts of these functions and datatypes together in one place, as they would usually be found in an architecture reference manual. To support this sail supports *scattered* definitions. First we give types for the execute and decode functions, and declare them as scattered functions, as well as the `ast` union.

```
enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI, RISCV_ORI, RISCV_ANDI} /*
    ↪ immediate ops */

scattered union ast

val decode : bits(32) -> option(ast) effect pure
scattered function decode

val execute : ast -> unit effect {rmem, rreg, wreg}
scattered function execute
```

Now we provide the clauses for the add immediate `ast` type, as well as it's execute and decode clauses. We can define the decode function by directly pattern matching on the bitvector representing the instruction. Sail supports vector concatentation patterns (`@` is the vector concatenation operator), and uses the types provided (e.g. `bits(12)` and `regbits`) to destructure the vector in the correct way.

```
union clause ast = ITYPE : (bits(12), regbits, regbits, iop)

function clause decode imm : bits(12) @ rs1 : regbits @ 0b000 @ rd : regbits @ 0b0010011
  = Some(ITYPE(imm, rs1, rd, RISCV_ADDI))

function clause execute (ITYPE (imm, rs1, rd, RISCV_ADDI)) =
  let rs1_val = X(rs1) in
  let imm_ext : xlen_t = EXTS(imm) in
  let result = rs1_val + imm_ext in
  X(rd) = result
```

Now we do the same thing for the load duble instruction:

```
union clause ast = LOAD : (bits(12), regbits, regbits)

function clause decode imm : bits(12) @ rs1 : regbits @ 0b011 @ rd : regbits @ 0b0000011
  = Some(LOAD(imm, rs1, rd))

function clause execute(LOAD(imm, rs1, rd)) =
    let addr : xlen_t = X(rs1) + EXTS(imm) in
    let result : xlen_t = MEMr(addr, 8) in
    X(rd) = result
```

Finally we define the fallthrough case for the decode function, and end all our scattered definitions. Note that the clauses in a scattered function will be executed in the order they appear in the file.

```
function clause decode _ = None()

end ast
end decode
end execute
```

The actual code for this example, as well as our more complete RISC-V specification can be found on our github at https://github.com/rems-project/sail/blob/sail2/riscv/riscv_duopod.sail.

3 Using Sail

In its most basic use-case Sail is a command-line tool, analogous to a compiler: one gives it a list of input Sail files; it type-checks them and provides translated output.

To simply typecheck Sail files, one can pass them on the command line with no other options, so for our RISC-V spec:

```
sail prelude.sail riscv_types.sail riscv_sys.sail riscv.sail
```

The sail files passed on the command line are simply treated as if they are one large file concatentated together, although the parser will keep track of locations on a per-file basis for error-reporting. As can be seen, this specification is split into several logical components. `prelude.sail` defines the initial type environment and builtins, `riscv_types.sail` gives type definitions used in the rest of the specification and then `riscv_sys.sail` and `riscv.sail` implement most of the specification.

For more complex projects, once can use `$include` statments in Sail source, for example:

```
$include <library.sail>
$include "file.sail"
```

Here, Sail will look for `library.sail` in the `$SAIL_DIR/lib`, where `$SAIL_DIR` is usually the root of the sail repository. It will search for `file.sail` relative to the location of the file containing the `$include`. The space after the include is mandatory. Sail also supports `$define`, `$ifdef`, and `$ifndef`. These are things that are understood by Sail itself, not a separate preprocessor, and are handled after the AST is parsed [2].

3.1 OCaml compilation

To compile a Sail specification into OCaml, one calls Sail as

```
sail -ocaml FILES
```

This will produce a version of the specification translated into Ocaml, which is placed into a directory called `_sbuild`, similar to ocamlbuild's `_build` directory. The generated OCaml is intended to be fairly close to the original Sail source, and currently we do not attempt to do much optimisation on this output.

The contents of the `_sbuild` directory are set up as an ocamlbuild project, so one can simply switch into that directory and run

```
ocamlbuild -use-ocamlfind out.cmx
```

to compile the generated model. Currently the OCaml compilation requires that lem, linksem, and zarith are available as ocamlfind findable libraries, and also that the environment variable `$SAIL_DIR` is set to the root of the Sail repository.

If the Sail specification contains a `main` function with type `unit -> unit` that implements a fetch/decode/execute loop then the OCaml backend can produce a working executable, by running

```
sail -o out -ocaml FILES
```

Then one can run

```
./out ELF_FILE
```

to simulate an ELF file on the specification. One can do `$include <elf.sail>` to gain access to some useful functions for accessing information about the loaded ELF file from within the Sail specification. In particular `elf.sail` defines a function `elf_entry : unit -> int` which can be used to set the PC to the correct location. ELF loading is done by the linksem library[3].

There is also an `-ocaml_trace` option which is the same as `-ocaml` except it instruments the generated OCaml code with tracing information.

[2] This can affect precedence declarations for custom user defined operators—the precedence must be redeclared in the file you are including the operator into.

[3] https://github.com/rems-project/linksem

3.2 Lem and Isabelle

We have a separate document detailing how to generate Isabelle theories from Sail models, and how to work with those models in Isabelle, see:

https://github.com/rems-project/sail/raw/sail2/snapshots/isabelle/Manual.pdf

Currently there are generated Isabelle snapshots for some of our models in `snapshots/isabelle` in the Sail repository. These snapshots are provided for convenience, and are not guaranteed to be up-to-date.

In order to open a theory of one of the specifications in Isabelle, use the -l Sail command-line flag to load the session containing the Sail library. Snapshots of the Sail and Lem libraries are in the `lib/sail` and `lib/lem` directories, respectively. You can tell Isabelle where to find them using the -d flag, as in

```
isabelle jedit -l Sail -d lib/lem -d lib/sail riscv/Riscv.thy
```

When run from the `snapshots/isabelle` directory this will open the RISC-V specification.

3.3 Interactive mode

Compiling Sail with

```
make isail
```

builds it with a GHCi-style interactive interpreter. This can be used by starting Sail with `sail -i`. If Sail is not compiled with interactive support the `-i` flag does nothing. Sail will still handle any other command line arguments as per usual, including compiling to OCaml or Lem. One can also pass a list of commands to the interpreter by using the `-is` flag, as

```
sail -is FILE
```

where `FILE` contains a list of commands. Once inside the interactive mode, a list of commands can be accessed by typing `:commands`, while `:help` can be used to provide some documentation for each command.

3.4 Other options

Here we summarize most of the other options available for Sail. Debugging options (usually for debugging Sail itself) are indicated by starting with the letter d.

- `-v` Print the Sail version.
- `-help` Print a list of options.
- `-no_warn` Turn off warnings.
- `-enum_casts` Allow elements of enumerations to be automatically casted to numbers.
- `-memo_z3` Memoize calls to the Z3 solver. This can greatly improve typechecking times if you are repeatedly typechecking the same specification while developing it.
- `-no_lexp_bounds_check` Turn off bounds checking in the left hand side of assignments.
- `-no_effects` Turn off effect checking. May break some backends that assume effects are properly checked.
- `-undefined_gen` Generate functions that create undefined values of user-defined types. Every type `T` will get a `undefined_T` function created for it. This flag is set automatically by some backends that want to re-write `undefined`.
- `-just_check` Force Sail to terminate immediately after typechecking.
- `-dno_cast` Force Sail to never perform type coercions under any circumstances.

- `-dtc_verbose <verbosity>` Make the typechecker print a trace of typing judgements. If the verbosity level is 1, then this should only include fairly readable judgements about checking and inference rules. If verbosity is 2 then it will include a large amount of debugging information. This option can be useful to diagnose tricky type-errors, especially if the error message isn't very good.
- `-ddump_tc_ast` Write the typechecked AST to stdout after typechecking
- `-ddump_rewrite_ast <prefix>` Write the AST out after each re-writing pass. The output from each pass is placed in a file starting with `prefix`.
- `-dsanity` Perform extra sanity checks on the AST.
- `-dmagic_hash` Allow the # symbol in identifiers. It's currently used as a magic symbol to separate generated identifiers from those the user can write, so this option allows for the output of the various other debugging options to be fed back into Sail.

4 Sail Language

4.1 Functions

In Sail, functions are declared in two parts. First we write the type signature for the function using the `val` keyword, then define the body of the function via the `function` keyword. In this Subsection, we will write our own version of the `replicate_bits` function from the Sail library. This function takes a number n and a bitvector, and copies that bitvector n times.

```
val my_replicate_bits : forall 'n 'm, 'm >= 1 & 'n >= 1. (int('n), bits('m)) -> bits('n * 'm)
```

The general syntax for a function type in Sail is as follows:

`val` *name* `:` `forall` *type variables* `,` *constraint* `.` `(` $type_1$ `,` `...` `,` $type_2$ `)` `->` *return type*

An implementation for the `replicate_bits` function would be follows

```
function my_replicate_bits(n, xs) = {
  ys = zeros(n * length(xs));
  foreach (i from 1 to n) {
    ys = ys << length(xs);
    ys = ys | zero_extend(xs, length(ys))
  };
  ys
}
```

The general syntax for a function declaration is:

`function` *name* `(` $argument_1$ `,` `...` `,` $argument_n$ `)` `=` *expression*

Code for this example can be found in the Sail repository in `doc/examples/my_replicate_bits.sail`. To test it, we can invoke Sail interactively using the `-i` option, passing the above file on the command line. Typing `replicate_bits(3, 0xA)` will step through the execution of the above function, and eventually return the result `0xAAA`. Typing `:run` in the interactive interpreter will cause the expression to be evaluated fully, rather than stepping through the execution.

Sail allows type variables to be used directly within expressions, so the above could be re-written as

```
function my_replicate_bits_2(n, xs) = {
  ys = zeros('n * 'm);
  foreach (i from 1 to n) {
    ys = (ys << 'm) | zero_extend(xs, 'n * 'm)
  };
  ys
}
```

This notation can be succinct, but should be used with caution. What happens is that Sail will try to re-write the type variables using expressions of the appropriate size that are available within the surrounding scope, in this case `n` and `length(xs)`. If no suitable expressions are found to trivially rewrite these type variables, then additional function parameters will be automatically added to pass around this information at run-time. This feature is however very useful for implementing functions with implicit parameters, e.g. we can implement a zero extension function that implicitly picks up its result length from the calling context as follows:

```
val cast extz : forall 'n 'm, 'm >= 'n. bits('n) -> bits('m)

function extz(xs) = zero_extend(xs, 'm)
```

Notice that we annotated the `val` declaration as a `cast`—this means that the type checker is allowed to automatically insert it where needed in order to make our code type check. This is another feature that must be used carefully, because too many implicit casts can quickly result in unreadable code. Sail does not make any distinction between expressions and statements, so since there is only a single line of code within the foreach block, we can drop it and simply write:

```
function my_replicate_bits_3(n, xs) = {
  ys = zeros('n * 'm);
  foreach (i from 1 to n) ys = ys << 'm | xs;
  ys
}
```

4.2 External Bindings

Rather than defining functions within Sail itself, they can be mapped onto functions definition within the various backend targets supported by Sail. This is primarily used to setup the primitive functions defined in the Sail library. These declarations work like much FFI bindings in many programming languages—in Sail we provide the `val` declaration, except rather than giving a function body we supply a string used to identify the external operator in each backend. For example, we could link the `>>` operator with the `shiftl` primitive as

```
val operator << = "shiftl" : forall 'm. (bits('m), int) -> bits('m)
```

If the external function has the same name as the sail function, such as `shiftl = "shiftl"`, then we can use the shorthand syntax

```
val "shiftl" : forall 'm. (bits('m), int) -> bits('m)
```

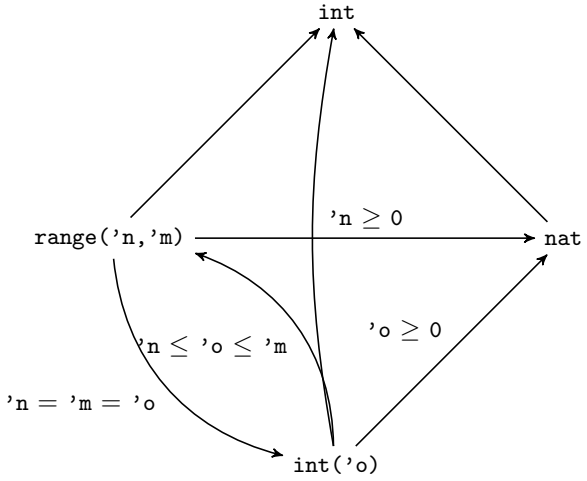
We can map onto differently-named operations for different targets by using a JSON-like `key: "value"` notation, for example:

```
val operator >> = {
  ocaml: "shiftr_ocaml",
  c: "shiftr_c",
  lem: "shiftr_lem",
  _: "shiftr"
} : forall 'm. (bits('m), int) -> bits('m)
```

We can also omit backends—in which case those targets will expect a definition written in Sail. Note that no checking is done to ensure that the definition in the target matches the type of the Sail function. Finally, the `_` key used above is a wildcard that will be used for any backend not otherwise included. If we use the wildcard key, then we cannot omit specific backends to force them to use a definition in Sail.

4.3 Numeric Types

Sail has three basic numeric types, `int`, `nat`, and `range`. The type `int` is an arbitrary precision mathematical integer, and likewise `nat` is an arbitrary precision natural number. The type `range('n,'m)` is an inclusive range between the `Int`-kinded type variables `'n` and `'m`. The type `int('o)` is an integer exactly equal to the `Int`-kinded type variable `'n`, i.e. `int('o) = range('o,'o)`. These types can be used interchangeably provided the rules summarised in the below diagram are satisfied (via constraint solving).



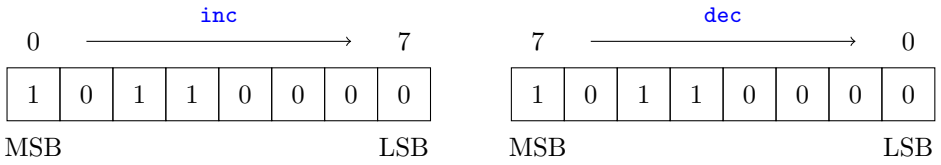
Note that `bit` isn't a numeric type (i.e. it's not `range(0,1)`. This is intentional, as otherwise it would be possible to write expressions like `(1 : bit)+ 5` which would end up being equal to `6 : range(5, 6)`. This kind of implicit casting from bits to other numeric types would be highly undesirable. The `bit` type itself is a two-element type with members `bitzero` and `bitone`.

4.4 Vector Type

Sail has the built-in type vector, which is a polymorphic type for fixed-length vectors. For example, we could define a vector `v` of three integers as follows:

```
let v : vector(3, dec, int) = [1, 2, 3]
```

The first argument of the vector type is a numeric expression representing the length of the vector, and the last is the type of the vector's elements. But what is the second argument? Sail allows two different types of vector orderings—increasing (`inc`) and decreasing (`dec`). These two orderings are shown for the bitvector 0b10110000 below.



For increasing (bit)vectors, the 0 index is the most significant bit and the indexing increases towards the least significant bit. Whereas for decreasing (bit)vectors the least significant bit is 0 indexed, and the indexing decreases from the most significant to the least significant bit. For this reason, increasing indexing is sometimes called 'most significant bit is zero' or MSB0, while decreasing indexing is sometimes called 'least significant bit is zero' or LSB0. While this vector ordering makes most sense for bitvectors (it is usually called bit-ordering), in Sail it applies to all vectors. A default ordering can be set using

```
default Order dec
```

and this should usually be done right at the beginning of a specification. Most architectures stick to either convention or the other, but Sail also allows functions which are polymorphic over vector order, like so:

```
val foo : forall ('a : Order). vector(8, 'a, bit) -> vector(8, 'a, bit)
```

Bitvector Literals Bitvector literals in Sail are written as either `0x`*hex string* or `0b`*binary string*, for example `0x12FE` or `0b1010100`. The length of a hex literal is always four times the number of digits, and the length of binary string is always the exact number of digits, so `0x12FE` has length 16, while `0b1010100` has length 7.

Accessing and Updating Vectors A vector can be indexed by using the *vector*`[`*index*`]` notation. So, in the following code:

```
let v : vector(4, dec, int) = [1, 2, 3, 4]
let a = v[0]
let b = v[3]
```

`a` will be `4`, and `b` will be `1` (not that `v` is `dec`). By default, Sail will statically check for out of bounds errors, and will raise a type error if it cannot prove that all such vector accesses are valid.

Vectors can be sliced using the *vector*`[`$index_{msb}$ `..` $index_{lsb}$`]` notation. The indexes are always supplied with the index closest to the MSB being given first, so we would take the bottom 32-bits of a decreasing bitvector `v` as `v[31 .. 0]`, and the upper 32-bits of an increasing bitvector as `v[0 .. 31]`, i.e. the indexing order for decreasing vectors decreases, and the indexing order for increasing vectors increases.

A vector index can be updated using `[`*vector* `with` *index* `=` *expression*`]` notation. Similarly, a subrange of a vector can be updated using `[`*vector* `with` $index_{msb}$ `..` $index_{lsb}$ `=` *expression*`]`, where the order of the indexes is the same as described above for increasing and decreasing vectors.

These expressions are actually just syntactic sugar for several built-in functions, namely `vector_access` ↪ , `vector_subrange`, `vector_update`, and `vector_update_subrange`.

4.5 List Type

In addition to vectors, Sail also has `list` as a built-in type. For example:

```
let l : list(int) = [|1, 2, 3|]
```

The cons operator is `::`, so we could equally write:

```
let l : list(int) = 1 :: 2 :: 3 :: [||]
```

Pattern matching can be used to destructure lists, see Section 4.7

4.6 Other Types

Sail also has a `string` type, and a `real` type. The `real` type is used to model arbitrary real numbers, so floating point instructions could be specified by saying that they are equivalent to mapping the floating point inputs to real numbers, performing the arithmetic operation on the real numbers, and then mapping back to a floating point value of the appropriate precision.

4.7 Pattern Matching

Like most functional languages, Sail supports pattern matching via the `match` keyword. For example:

```
let n : int = f();
match n {
  1 => print("1"),
  2 => print("2"),
  3 => print("3"),
  _ => print("wildcard")
}
```

The `match` keyword takes an expression and then branches using a pattern based on its value. Each case in the match expression takes the form *`pattern => expression`*, separated by commas. The cases are checked sequentially from top to bottom, and when the first pattern matches its expression will be evaluated.

`match` in Sail is not currently check for exhaustiveness—after all we could have arbitrary constraints on a numeric variable being matched upon, which would restrict the possible cases in ways that we could not determine with just a simple syntactic check. However, there is a simple exhaustiveness checker which runs and gives warnings (not errors) if it cannot tell that the pattern match is exhaustive, but this check can give false positives. It can be turned off with the `-no_warn` flag.

When we match on numeric literals, the type of the variable we are matching on will be adjusted. In the above example in the `print("1")` case, n will have the type `int('e)`, where `'e` is some fresh type variable, and there will be a constraint that `'e` is equal to one.

We can also have guards on patterns, for example we could modify the above code to have an additional guarded case like so:

```
let n : int = f();
match n {
  1 => print("1"),
  2 => print("2"),
  3 => print("3"),
  m if m <= 10 => print("n␣is␣less␣than␣or␣equal␣to␣10"),
  _ => print("wildcard")
}
```

The variable pattern m will match against anything, and the guard can refer to variables bound by the pattern.

Matching on enums Match can be used to match on possible values of an enum, like so:

```
enum E = A | B | C

match x {
  A => print("A"),
  B => print("B"),
  C => print("C")
}
```

Note that because Sail places no restrictions on the lexical structure of enumeration elements to differentiate them from ordinary identifiers, pattern matches on variables and enum elements can be somewhat ambiguous. This is the primary reason why we have the basic, but incomplete, pattern exhaustiveness check mentioned above—it can warn you if removing an enum constructor breaks a pattern match.

Matching on tuples We use match to destructure tuple types, for example:

```
let x : (int, int) = (2, 3) in
match x {
  (y, z) => print("y␣=␣2␣and␣z␣=␣3")
}
```

Matching on unions Match can also be used to destructure union constructors, for example using the option type from Section 4.10.3:

```
match option {
  Some(x) => foo(x),
  None() => print("matched␣None()")
}
```

Note that like how calling a function with a unit argument can be done as `f()` rather than `f(())`, matching on a constructor `C` with a unit type can be achieved by using `C()` rather than `C(())`.

Matching on bit vectors Sail allows numerous ways to match on bitvectors, for example:

```
match v {
  0xFF => print("hex␣match"),
  0x0000_0001 => print("binary␣match"),
  0xF @ v : bits(4) => print("vector␣concatenation␣pattern"),
  0xF @ [bitone, _, b1, b0] => print("vector␣pattern"),
  _ : bits(4) @ v : bits(4) => print("annotated␣wildcard␣pattern")
}
```

We can match on bitvector literals in either hex or binary forms. We also have vector concatenation patterns, of the form *pattern* `@` ... `@` *pattern*. We must be able to infer the length of all the sub-patterns in a vector concatenation pattern, hence why in the example above all the wildcard and variable patterns beneath vector concatenation patterns have type annotations. In the context of a pattern the `:` operator binds tighter than the `@` operator (as it does elsewhere).

We also have vector patterns, which for bitvectors match on individual bits. In the above example, `b0` and `b1` will have type `bit`. The pattern `bitone` is a bit literal, with `bitzero` being the other bit literal pattern.

Note that because vectors in sail are type-polymorphic, we can also use both vector concatenation patterns and vector patterns to match against non-bit vectors.

Matching on lists Sail allows lists to be destructured using patterns. There are two types of patterns for lists, cons patterns and list literal patterns.

```
match ys {
  x :: xs => print("cons␣pattern"),
  [||] => print("empty␣list")
}
```

```
match ys {
  [|1, 2, 3|] => print("list␣pattern"),
  _ => print("wildcard")
}
```

As patterns Like OCaml, Sail also supports naming parts of patterns using the `as` keyword. For example, in the above list pattern we could bind the entire list as ys as follows:

```
match ys {
  x :: xs as zs => print("cons␣with␣as␣pattern"),
  [||] => print("empty␣list")
}
```

The as pattern has lower precedence than any other keyword or operator in a pattern, so in this example zs will refer to `x :: xs`.

4.8 Mutable and Immutable Variables

Local immutable bindings can be introduced via the `let` keyword, which has the following form

`let` *pattern* `=` *expression* `in` *expression*

The pattern is matched against the first expression, binding any identifiers in that pattern. The pattern can have any form, as in the branches of a match statement, but it should be complete (i.e. it should not fail to match)[4].

When used in a block, we allow a variant of the let statement, where it can be terminated by a semicolon rather than the in keyword.

```
{
  let pattern = expression_0;
  expression_1;
  ⋮
  expression_n
}
```

This is equivalent to the following

```
{
  let pattern = expression_0 in {
    expression_1;
    ⋮
    expression_n
  }
}
```

If we were to write

```
{
  let pattern = expression_0 in
  expression_1;
  ⋮
  expression_n // pattern not visible
}
```

instead, then *pattern* would only be bound within $expression_1$ and not any further expressions. In general the block-form of let statements terminated with a semicolon should be preferred within blocks.

Variables bound within function arguments, match statement, and let-bindings are always immutable, but Sail also allows mutable variables. Mutable variables are bound implicitly by using the assignment operator within a block.

[4] although this is not checked right now

```
{
  x : int = 3 // Create a new mutable variable x initialised to 3
  x = 2 // Rebind it to the value 2
}
```

The assignment operator is the equality symbol, as in C and other programming languages. Sail supports a rich language of *l-value* forms, which can appear on the left of an assignment. These will be described in Subsection 4.8.1. Note that we could have written

```
{
  x = 3;
  x = 2
}
```

but it would not have type-checked. The reason for this is if a mutable variable is declared without a type, Sail will try to infer the most specific type from the left hand side of the expression. However, in this case Sail will infer the type as `int(3)` and will therefore complain when we try to reassign it to `2`, as the type `int(2)` is not a subtype of `int(3)`. We therefore declare it as an `int` which as mentioned in Section 4.3 is a supertype of all numeric types. Sail will not allow us to change the type of a variable once it has been created with a specific type. We could have a more specific type for the variable `x`, so

```
{
  x : {|2, 3|} = 3;
  x = 2
}
```

would allow `x` to be either 2 or 3, but not any other value. The `{|2, 3|}` syntax is equivalent to `{'n, 'n in {2, 3}. int('n)}`.

4.8.1 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.

We have l-values that allow us to write to individual elements of a vector:

```
{
  v : bits(8) = 0xFF;
  v[0] = bitzero;
  assert(v == 0xFE)
}
```

as well as sub ranges of a vector:

```
{
  v : bits(8) = 0xFF;
  v[4 .. 0] = 0x0; // assume default Order dec
  assert(v == 0xF0)
}
```

We also have vector concatenation l-values, which work much like vector concatenation patterns

```
{
  v1 : bits(4) = 0xF;
  v2 : bits(4) = 0xF;
  v1 @ v2 = 0xAB;
  assert(v1 == 0xA & v2 == 0xB)
}
```

For structs we can write to an individual struct field as

```
{
  s : S = struct { field = 0xFF }
  s.field = 0x00
}
```

assume we have a struct type `S`, with a field simply called `field`. We can do multiple assignment using tuples, e.g.

```
{
  (x, y) = (2, 3);
  assert(x == 2 & x == 3)
}
```

Finally, we allow functions to appear in l-values. This is a very simple way to declare 'setter' functions that look like custom l-values, for example:

```
{
  memory(addr) = 0x0F
}
```

This works because `f(x)= y` is sugar for `f(x, y)`. This feature is commonly used when setting registers or memory that has additional semantics for when they are read or written. We commonly use the overloading feature to declare what appear to be getter/setter pairs, so the above example we could implement a `read_memory` function and a `write_memory` function and overload them both as `memory` to allow us to write memory using `memory(addr)= data` and read memory as `data = memory(addr)`, as so:

```
val read_memory : bits(64) -> bits(8)
val write_memory : (bits(64), bits(8)) -> unit

overload memory = {read_memory, write_memory}
```

For more details on operator and function overloading see Section 4.12.

4.9 Registers

Registers can be declared with a top level

`register` *name* : *type*

declaration. Registers are essentially top-level global variables and can be set with the previously discussed l-expression forms. There is currently no restriction on the type of a register in Sail.[5]

Registers differ from ordinary mutable variables as we can pass around references to them by name. A reference to a register `R` is created as `ref R`. If the register `R` has the type `A`, then the type of `ref R` will be `register(A)`. There is a dereferencing l-value operator `*` for assigning to a register reference. One use for register references is to create a list of general purpose registers, so they can be indexed using numeric variables. For example:

```
default Order dec
$include <prelude.sail>

register X0 : bits(8)
register X1 : bits(8)
register X2 : bits(8)

let X : vector(3, dec, register(bits(8))) = [ref X2, ref X1, ref X0]

function main() : unit -> unit = {
  X0 = 0xFF;
  assert(X0 == 0xFF);
  (*X[0]) = 0x11;
  assert(X0 == 0x11);
  (*ref X0) = 0x00;
  assert(X0 == 0x00)
}
```

We can dereference register references using the `"reg_deref"` builtin (see Section 4.16), which is set up like so:

[5] We may at some point want to enforce that they can be mapped to bitvectors.

```
val "reg_deref" : forall ('a : Type). register('a) -> 'a effect {rreg}
```

Currently there is no built-in syntactic sugar for dereferencing registers in expressions.

Unlike previous versions of Sail, referencing and de-referencing registers is done explicitly, although we can use an automatic cast to implictly dereference registers if that semantics is desired for a specific specification that makes heavy use of register references, like so:

```
val cast auto_reg_deref = "reg_deref" : forall ('a : Type). register('a) -> a effect {rreg}
```

4.10 Type declarations

4.10.1 Enumerations

Enumerations can be defined in either a Haskell-like syntax (useful for smaller enums) or a more traditional C-like syntax, which is often more readable for enumerations with more members. There are no lexical constraints on the identifiers that can be part of an enumeration. There are also no restrictions on the name of a enumeration type, other than it must be a valid identifier. For example, the following shows two ways to define the enumeration `Foo` with three members, `Bar`, `Baz`, and `quux`:

```
enum Foo = Bar | Baz | quux

enum Foo = {
  Bar,
  Baz,
  quux
}
```

For every enumeration type E sail generates a `num_of_`E function and a E`_of_num` function, which for `Foo` above will have the following definitions[6]:

```
val Foo_of_num : forall 'e, 0 <= 'e <= 2. int('e) -> Foo
function Foo_of_num(arg) = match arg {
  0 => Bar,
  1 => Baz,
  _ => quux
}

val num_of_Foo : Foo -> {'e, 0 <= 'e <= 2. int('e)}
function num_of_Foo(arg) = match arg {
  Bar => 0,
  Baz => 1,
  quux => 2
}
```

Note that these functions are not automatically made into implicit casts.

4.10.2 Structs

Structs are defined using the struct keyword like so:

```
struct Foo = {
  bar : vector(4, dec, bit),
  baz : int,
  quux : range(0, 9)
}
```

If we have a struct `foo : Foo`, its fields can be accessed by `foo.bar`, and set as `foo.bar = 0xF`. It can also be updated in a purely functional fashion using the construct `{foo with bar = 0xF}`. There is no lexical restriction on the name of a struct or the names of its fields.

[6] It will ensure that the generated function name `arg` does not clash with any enumeration constructor.

4.10.3 Unions

As an example, the `maybe` type á la Haskell could be defined in Sail as follows:

```
union maybe ('a : Type) = {
  Just : 'a,
  None : unit
}
```

Constructors, such as `Just` are called like functions, as in `Just(3): maybe(int)`. The `None` constructor is also called in this way, as `None()`. Notice that unlike in other languages, every constructor must be associated with a type—there are no nullary constructors. As with structs there are no lexical restrictions on the names of either the constructors nor the type itself, other than they must be valid identifiers.

4.10.4 Bitfields

The following example creates a bitfield type called `cr` and a register `CR` of that type.

```
bitfield cr : vector(8, dec, bit) = {
  CR0 : 7 .. 4,
  LT : 7,
  GT : 6,
  CR1 : 3 .. 2,
  CR3 : 1 .. 0
}
register CR : cr
```

A bitfield definition creates a wrapper around a bit vector type, and generates getters and setters for the fields. For the setters, it is assumed that they are being used to set registers with the bitfield type[7]. If the bitvector is decreasing then indexes for the fields must also be in decreasing order, and vice-versa for an increasing vector. For the above example, the bitfield wrapper type will be the following:

```
union cr = { Mk_cr(vector(8, dec, bit)) }
```

The complete vector can be accessed as `CR.bits()`, and a register of type `cr` can be set like `CR->bits()` ↪ `= 0xFF`. Getting and setting individual fields can be done similarly, as `CR.CR0()` and `CR->CR0()= 0xF`. Internally, the bitfield definition will generate a `_get_`F and `_set_`F function for each field F, and then overload them as `_mod_`F for the accessor syntax. The setter takes the bitfield as a reference to a register, hence why we use the `->` notation. For pure updates of values of type `cr` a function `update_`F is also defined. For more details on getters and setters, see Section **??**. A singleton bit in a bitfield definition, such as `LT : 7` will be defined as a bitvector of length one, and not as a value of type `bit`, which mirrors the behaviour of ARM's ASL language.

4.11 Operators

Valid operators in Sail are sequences of the following non alpha-numeric characters: `!%&*+-./:<>=@^|`. Additionally, any such sequence may be suffixed by an underscore followed by any valid identifier, so `<=_u` or even `<=_unsigned` are valid operator names. Operators may be left, right, or non-associative, and there are 10 different precedence levels, ranging from 0 to 9, with 9 binding the tightest. To declare the precedence of an operator, we use a fixity declaration like:

```
infix <=_u 4
```

For left or right associative operators, we'd use the keywords `infixl` or `infixr` respectively. An operator can be used anywhere a normal identifier could be used via the `operator` keyword. As such, the `<=_u` operator can be defined as:

```
val operator <=_u : forall 'n. (bits('n), bits('n)) -> bool
function operator <=_u(x, y) = unsigned(x) <= unsigned(y)
```

[7] This functionality was originally called *register types* for this reason, but this was confusing because types of registers are not always register types.

Builtin precedences The precedence of several common operators are built into Sail. These include all the operators that are used in type-level numeric expressions, as well as several common operations such as equality, division, and modulus. The precedences for these operators are summarised in Table 1.

Precedence	Left associative	Non-associative	Right associative
9			
8			`^`
7	`*`, `/`, `%`		
6	`+`, `-`		
5			
4		`<`, `<=`, `>`, `>=`, `!=`, `=`, `==`	
3			`&`
2			`|`
1			
0			

Table 1: Default Sail operator precedences

Type operators Sail allows operators to be used at the type level. For example, we could define a synonym for the built-in `range` type as:

```
infix 3 ...

type operator ... ('n : Int) ('m : Int) = range('n, 'm)

let x : 3 ... 5 = 4
```

Note that we can't use `..` as an operator name, because that is reserved syntax for vector slicing. Operators used in types always share precedence with identically named operators at the expression level.

4.12 Ad-hoc Overloading

Sail has a flexible overloading mechanism using the `overload` keyword

$$\texttt{overload}\ name\ \texttt{=}\ \{\ name_1\ \texttt{,}\ \ldots\ \texttt{,}\ name_n\ \}$$

This takes an identifier name, and a list of other identifier names to overload that name with. When the overloaded name is seen in a Sail definition, the type-checker will try each of the overloads in order from left to right (i.e. from $name_1$ to $name_n$). until it finds one that causes the resulting expression to type-check correctly.

Multiple `overload` declarations are permitted for the same identifier, with each overload declaration after the first adding it's list of identifier names to the right of the overload list (so earlier overload declarations take precedence over later ones). As such, we could split every identifier from above syntax example into it's own line like so:

$$\begin{array}{c}\texttt{overload}\ name\ \texttt{=}\ \{\ name_1\ \}\\ \vdots\\ \texttt{overload}\ name\ \texttt{=}\ \{\ name_n\ \}\end{array}$$

As an example for how overloaded functions can be used, consider the following example, where we define a function `print_int` and a function `print_string` for printing integers and strings respectively. We overload `print` as either `print_int` or `print_string`, so we can print either number such as 4, or strings like `"Hello, World!"` in the following `main` function definition.

```
val print_int : int -> unit

val print_string : string -> unit
```

```
overload print = {print_int, print_string}

function main() : unit -> unit = {
  print("Hello, World!");
  print(4)
}
```

We can see that the overloading has had the desired effect by dumping the type-checked AST to stdout using the following command `sail -ddump_tc_ast examples/overload.sail`. This will print the following, which shows how the overloading has been resolved

```
function main () : unit = {
  print_string("Hello, World!");
  print_int(4)
}
```

This option can be quite useful for testing how overloading has been resolved. Since the overloadings are done in the order they are listed in the source file, it can be important to ensure that this order is correct. A common idiom in the standard library is to have versions of functions that guarantee more constraints about there output be overloaded with functions that accept more inputs but guarantee less about their results. For example, we might have two division functions:

```
val div1 : forall 'm, 'n >= 0 & 'm > 0. (int('n), int('m)) -> {'o, 'o >= 0. int('o)}

val div2 : (int, int) -> option(int)
```

The first guarantees that if the first argument is greater than or equal to zero, and the second argument is greater than zero, then the result will be greater than or not equal to zero. If we overload these definitions as

```
overload operator / = {div1, div2}
```

Then the first will be applied when the constraints on it's inputs can be resolved, and therefore the guarantees on it's output can be guaranteed, but the second will be used when this is not the case, and indeed, we will need to manually check for the division by zero case due to the option type. Note that the return type can be very different between different cases in the overloaded.

The amount of arguments overloaded functions can have can also vary, so we can use this to define functions with optional arguments, e.g.

```
val zero_extend_1 : forall 'm 'n, 'm <= 'n. bits('m) -> bits('n)

val zero_extend_2 : forall 'm 'n, 'm <= 'n. (bits('m), int('n)) -> bits('n)

overload zero_extend = {zero_extend_1, zero_extend_2}
```

In this example, we can call `zero_extend` and the return length is implicit (likely using `sizeof`, see Section 4.13) or we can provide it ourselves as an explicit argument.

4.13 Sizeof and Constraint

As already mention in Section 4.1, Sail allows for arbitrary type variables to be included within expressions. However, we can go slightly further than this, and include both arbitrary (type-level) numeric expressions in code, as well as type constraints. For example, if we have a function that takes two bitvectors as arguments, then there are several ways we could compute the sum of their lengths.

```
val f : forall 'n 'm. (bits('n), bits('m)) -> unit

function f(xs, ys) = {
  let len = length(xs) + length(ys);
  let len = 'n + 'm;
  let len = sizeof('n + 'm);
  ()
}
```

Note that the second line is equivalent to

```
let len = sizeof('n) + sizeof('n)
```

There is also the `constraint` keyword, which takes a type-level constraint and allows it to be used as a boolean expression, so we could write:

```
function f(xs, ys) = {
  if constraint('n <= 'm) {
    // Do something
  }
}
```

Rather than the equivlent test `length(xs)<= length(ys)`. This way of writing expressions can be succint, and can also make it very explicit what constraints will be generated during flow typing. However, all the constraint and sizeof definitions must be re-written to produce executable code, which can result in the generated theorem prover output diverging (in appearance) somewhat from the source input. In general, it is probably best to use `sizeof` and `constraint` sparingly.

However, as previously mentioned both `sizeof` and `constraint` can refer to type variables that only appear in the output or are otherwise not accessible at runtime, and so can be used to implement implicit arguments, as was seen for `replicate_bits` in Section 4.1.

4.14 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 Section 2. Sail permits this with syntactic sugar for 'scattered' definitions. Either functions or union types can be scattered.

One begins a scattered definition by declaring the name and kind (either function or union) of the scattered definition, e.g.

```
scattered function foo

scattered union bar
```

This is then followed by a list of clauses for either the union or the function, which can be freely interleaved with other definitions (such as `E` in the below code)

```
union clause bar : Baz(int, int)

function clause foo(Baz(x, y)) = ...

enum E = A | B | C

union clause bar : Quux(string)

function clause foo(Quux(str)) = print(str)
```

Finally the scattered definition is ended with the `end` keyword, like so:

```
end foo

end bar
```

Semantically, scattered definitions of union 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 should be avoided.

4.15 Exceptions

Perhaps suprisingly for a specification language, Sail has exception support. This is because exceptions as a language feature do sometimes appear in vendor ISA pseudocode, and such code would be very

difficult to translate into Sail if Sail did not itself support exceptions. We already translate Sail to monadic theorem prover code, so working with a monad that supports exceptions there is fairly natural.

For exceptions we have two language features: `throw` statements and `try-catch` blocks. The throw keyword takes a value of type `exception` as an argument, which can be any user defined type with that name. There is no builtin exception type, so to use exceptions one must be set up on a per-project basis. Usually the exception type will be a union, often a scattered union, which allows for the exceptions to be declared throughout the specification as they would be in OCaml, for example:

```
val print = {ocaml: "print_endline"} : string -> unit

scattered union exception

union clause exception = Epair : (range(0, 255), range(0, 255))

union clause exception = Eunknown : string

function main() : unit -> unit = {
  try {
    throw(Eunknown("foo"))
  } catch {
    Eunknown(msg) => print(msg),
    _ => exit()
  }
}

union clause exception = Eint : int

end exception
```

Note how the use of the scattered type allows additional exceptions to be declared even after they are used.

4.16 Preludes and Default Environment

By default Sail has almost no built-in types or functions, except for the primitive types described in this Chapter. This is because different vendor-pseudocode's have varying naming conventions and styles for even the most basic operators, so we aim to provide flexibility and avoid committing to any particular naming convention or set of built-ins. However, each Sail backend typically implements specific external names, so for a PowerPC ISA description one might have:

```
val EXTZ = "zero_extend" : ...
```

while for ARM, one would have

```
val ZeroExtend = "zero_extend" : ...
```

where each backend knows about the `"zero_extend"` external name, but the actual Sail functions are named appropriately for each vendor's pseudocode. As such each Sail ISA spec tends to have it's own prelude.

However, the `lib` directory in the Sail repository contains some files that can be included into any ISA specification for some basic operations. These are listed below:

flow.sail Contains basic definitions required for flow typing to work correctly.

arith.sail Contains simple arithmetic operations for integers.

vector_dec.sail Contains operations on decreasing (`dec`) indexed vectors, see Section 4.4.

vector_inc.sail Like `vector_dec.sail`, except for increasing (`inc`) indexed vectors.

option.sail Contains the definition of the option type, and some related utility functions.

prelude.sail Contains all the above files, and chooses between `vector_dec.sail` and `vector_inc.sail` based on the default order (which must be set before including this file).

smt.sail Defines operators allowing div, mod, and abs to be used in types by exposing them to the Z3 SMT solver.

exception_basic.sail Defines a trivial exception type, for situations where you don't want to declare your own (see Section 4.15).

5 Type System

(This section is still a work in progress)

5.1 Blocks

$$\frac{\Gamma \vdash E_0 \Leftarrow \texttt{bool} \qquad \Gamma \vdash M \Leftarrow \texttt{string} \qquad \text{FlowThen}(\Gamma, E_0) \vdash \texttt{\{}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}{\Gamma \vdash \texttt{\{assert(}E_0\texttt{,}M\texttt{);}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}$$

$$\frac{\text{BindAssignment}(\Gamma, L_0, E_0) \vdash \texttt{\{}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}{\Gamma \vdash \texttt{\{}L_0\texttt{=}E_0\texttt{;}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}$$

$$\frac{\Gamma \vdash E_0 \Leftarrow \texttt{unit} \qquad \Gamma \vdash \texttt{\{}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}{\Gamma \vdash \texttt{\{}E_0\texttt{;}E_1\texttt{;}\ldots\texttt{;}E_n\texttt{\}} \Leftarrow A}$$

$$\frac{\Gamma \vdash E \Leftarrow A}{\Gamma \vdash \texttt{\{}E\texttt{\}} \Leftarrow A}$$

5.2 Let bindings

Note that `{let x = y; `E_0`; ...; `E_n`}` is equivalent to `{let x = y in {`E_0`; ...; `E_n`}}`, which is why there are no special cases for let bindings in Subsection 5.1.

$$\frac{\Gamma \vdash E_0 \Leftarrow B \qquad \text{BindPattern}(\Gamma, P, B) \vdash E_1 \Leftarrow A}{\Gamma \vdash \texttt{let}\ P\ \texttt{:}\ B\ \texttt{=}\ E_0\ \texttt{in}\ E_1 \Leftarrow A}$$

$$\frac{\Gamma \vdash E_0 \Rightarrow B \qquad \text{BindPattern}(\Gamma, P, B) \vdash E_1 \Leftarrow A}{\Gamma \vdash \texttt{let}\ P\ \texttt{=}\ E_0\ \texttt{in}\ E_1 \Leftarrow A}$$

Pattern bindings The BindPattern and BindPattern′ functions are used to bind patterns into an environment. The first few cases are simple, if we bind an identifier x against a type T, where x is either immutable or unbound, then $x : T$ is added to the environment. If we bind a type against a wildcard pattern, then the environment is returned unchanged. An `as` pattern binds its variable with the appropriate type then recursively binds the rest of the pattern. When binding patterns we always bind against the base type, and bring existentials into scope, which is why BindPattern does this and then calls the BindPattern′ function which implements all the cases.

$$\begin{aligned}
\text{BindPattern}(\Gamma, P, T) &= \text{BindPattern}'(\Gamma \lhd T, P, \text{Base}(T)) \\
\text{BindPattern}'(\Gamma, x, T) &= \Gamma \oplus x : T, && (x \text{ is unbound or immutable}) \\
\text{BindPattern}'(\Gamma, _, T) &= \Gamma, \\
\text{BindPattern}'(\Gamma, P\ \texttt{as}\ x, T) &= \text{BindPattern}(\Gamma \oplus x : T, P, T). && (x \text{ is unbound or immutable})
\end{aligned}$$

If we try to bind a numeric literal n against a type `int(`N`)` then we add a constraint to the environment that the nexp N is equal to n.

$$\text{BindPattern}'(\Gamma, n, \texttt{int(}N\texttt{)}) = \Gamma \oplus (N = n).$$

We also have some rules for typechecking lists, as well as user defined constructors in unions (omitted here)

$$\begin{aligned}
\text{BindPattern}'(\Gamma, [], \texttt{list(}A\texttt{)}) &= \Gamma, \\
\text{BindPattern}'(\Gamma, P_{hd}\ \texttt{::}\ P_{tl}, \texttt{list(}A\texttt{)}) &= \text{BindPattern}(\text{BindPattern}(\Gamma, P_{hd}, A), P_{tl}, \texttt{list(}A\texttt{)}).
\end{aligned}$$

The pattern binding code follows a similar structure to the bi-directional nature of the typechecking rules—the BindPattern function acts like a checking rule where we provide the type, and there is also an InferPattern function which acts like bind pattern but infers the types from the patterns. There is therefore a final case $\text{BindPattern}(\Gamma, P, T) = \Gamma'$ where $(\Gamma', T') = \text{InferPattern}(\Gamma, P)$ and $T \subseteq T'$.

The InferPattern function is defined by the following cases

$$\begin{aligned}
\text{InferPattern}(\Gamma, x) &= (\Gamma, T_{enum}), && (x \text{ is an element of enumeration } T_{enum}) \\
\text{InferPattern}(\Gamma, L) &= (\Gamma, \text{InferLiteral}(L)), && (L \text{ is a literal}) \\
\text{InferPattern}(\Gamma, P : T) &= (\text{BindPattern}(\Gamma, P, T), T).
\end{aligned}$$

Type patterns There is one additional case for BindPattern′ which we haven't discussed. TODO: type patterns

5.3 If statements

$$\frac{\Gamma \vdash E_{if} \Leftarrow \texttt{bool} \qquad \text{FlowThen}(\Gamma, E_{if}) \vdash E_{then} \Leftarrow A \qquad \text{FlowElse}(\Gamma, E_{if}) \vdash E_{else} \Leftarrow A}{\Gamma \vdash \texttt{if}\ E_{if}\ \texttt{then}\ E_{then}\ \texttt{else}\ E_{else} \Leftarrow A}$$

5.4 Return

When checking the body of a function, the expected return type of the function is placed into the context Γ.

$$\frac{\Gamma \vdash E \Leftarrow \text{Return}(\Gamma)}{\Gamma \vdash \texttt{return(}E\texttt{)} \Leftarrow A}$$

5.5 Functions

Depending on the context, functions can be either checked or inferred—although the only difference between the two cases is that in the checking case we can use the expected return type to resolve some of the function quantifiers, whereas in the inferring case we cannot.

$$\frac{f : \forall Q, C.(B_0, \ldots, B_n) \to R \in \Gamma \quad \textsc{InferFun}(\Gamma, Q, C, (B_0, \ldots, B_n), R, (x_0, \ldots, x_n)) = R'}{\Gamma \vdash f(x_0, \ldots, x_n) \Rightarrow R'}$$

$$\frac{f : \forall Q, C.(B_0, \ldots, B_n) \to R \in \Gamma \quad \textsc{CheckFun}(\Gamma, Q, C, (B_0, \ldots, B_n), R, (x_0, \ldots, x_n), R')}{\Gamma \vdash f(x_0, \ldots, x_n) \Leftarrow R'}$$

The rules for checking or inferring functions are rather more complicated than the other typing rules and are hard to describe in purely logical terms, so they are instead presented as an algorithm in Figure 5.5. Roughly the inference algorithm works as follows:

1. InferFun takes as input the typing context Γ, the list of quantifiers Q (a list of type variable/kind pairs), a constraint C, the function argument types $B_0 \ldots B_n$, the function return type R, and finally this list of argument expressions the function is applied to $x_0 \ldots x_n$.

2. We create an empty list of unsolved typing goals (expression/type pairs) called *unsolved*, a list of constraints *Constraints*, and a set of existential variables *Existentials*.

3. We iterate over each argument expression and type x_m and B_m, if x_m contains free type variables in Q we infer the type of x_n and attempt to unify that inferred type with B_m. If this unification step fails we add (x_m, B_m) to the list of unsolved goals. This unification step may generate new existential variables and constraints which are added to *Existentials* and *Constraints* as needed. The results of this unification step are used to resolve the univarsally-quantified type variables in Q. If x_m does not contain free type variables in Q, then we simply check it against B_m.

4. After this loop has finished we expect all the type variables in Q to have been resolved. If not, we throw a type error.

5. We now try to prove the function's constraint C using the resolved type variables, and check any remaining function arguments in *unsolved*.

6. Finally, we add any new existentials and constraints to the function's return type R, simplifying if at all possible (using SimplifyExist), before returning this type as the inferred type of the function.

The CheckFun calls the InferFun function, but it takes an additional X argument which the the required return type in the context where the function being checked is called. It additionally unifies the function's declared return type with the expected return type, and uses this to resolve any quantifiers in Q, provided that the return type is not existentially quantified. It may also be required to coerce R into X.

```
1: function InferFun(Γ, Q, C, (B_0, ..., B_n), R, (x_0, ..., x_n))
2:     unsolved ← []; Constraints ← []; Existentials ← ∅
3:     for all m ∈ 0, ..., n do
4:         if B_m contains type variables in Q then
5:             Γ ⊢ x_m ⇒ E                                   ▷ Infer the type of x_m as E
6:             unifiers, existentials, constraint ← CoerceAndUnify(Γ, E, B)
7:             if CoerceAndUnify failed with UnificationError then
8:                 unsolved ← (x_m, B_m) : unsolved
9:                 continue                                  ▷ Skip to next iteration of loop
10:            else if existentials is not empty then
11:                Add type variables existentials to Γ
12:                Add constraint constraint to Γ
13:                Constraints ← constraint : Constraints
14:                Existentials ← existentials ∪ Existentials
15:            end if
16:            for all (nvar, nexp) ∈ unifiers do
17:                B_0, ..., B_n ← B_0[nvar := nexp], ..., B_n[nvar := nexp]
18:                R ← R[nvar := nexp]; C ← C[nvar := nexp]
19:                Remove nvar from Q
20:            end for
21:        else if B_m does not contain type variables in Q then
22:            Γ ⊢ x_m ⇐ B_m                                 ▷ Check type of x_m against B_m
23:        end if
24:    end for
25:    if Q is not empty then
26:        raise TypeError                                   ▷ Unresolved universal quantifers
27:    end if
28:    Prove(Γ, C)
29:    for all (x_m, B_m) ∈ unsolved do Γ ⊢ x_m ⇐ B_m
30:    end for
31:    return SimplifyExist(exist Existentials, Constraints. R)
32: end function
33:
34: function CheckFun(Γ, Q, C, (B_0, ..., B_n), R, (x_0, ..., x_n), X)
35:    if X and R are not existentially quantified then
36:        unifiers, _, _ ← Unify(Γ, R, X)
37:        if Unify failed with UnificationError then skip
38:        else
39:            for all (nvar, nexp) ∈ unifiers do
40:                B_0, ..., B_n ← B_0[nvar := nexp], ..., B_n[nvar := nexp]
41:                R ← R[nvar := nexp]; C ← C[nvar := nexp]
42:                Remove nvar from Q
43:            end for
44:        end if
45:    end if
46:    R' ← InferFun(Γ, Q, C, (B_0, ..., B_n), R, (x_0, ..., x_n))
47:    return Coerce(R', X)
48: end function
```

Figure 1: Inference and checking algorithms for function calls

References

[1] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm, December 2016. 325462-061US.

[2] AMD. AMD64 architecture programmer's manual volume 1: Application programming. http://support.amd.com/TechDocs/24592.pdf, October 2013. Revision 3.21.

[3] *Power ISA Version 2.06B*. IBM, 2010. https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf (accessed 2015/07/22).

[4] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*, 2015. ARM DDI 0487A.h (ID092915).

[5] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Proc. FMCAD*, 2016.

[6] MIPS Technologies, Inc. MIPS64 Architecture For Programmers. Volume II: The MIPS64 Instruction Set, July 2005. Revision 2.50. Document Number: MD00087.

[7] MIPS Technologies, Inc. MIPS64 Architecture For Programmers. Volume III: The MIPS64 Privileged Resource Architecture, July 2005. Revision 2.50. Document Number: MD00091.

[8] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of POPL 2009: the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 379–391, January 2009.

[9] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *Proceedings of TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009.

[10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).

[11] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proceedings of TACAS 2011: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 175–186, 2011.

[14] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI 2012, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, pages 311–322, 2012.

[15] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2015.

[16] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL 2017: The 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, January 2017.

[18] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.

[19] Lem implementation. https://bitbucket.org/Peter_Sewell/lem/, 2017.

[20] Robert N. M. Watson, Jonathan Woodruff, David Chisnall, Brooks Davis, Wojciech Koszek, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Robert Norton, and Michael Roe. Bluespec Extensible RISC Implementation: BERI Hardware reference. Technical Report UCAM-CL-TR-868, University of Cambridge, Computer Laboratory, April 2015.

[21] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual (Second Edition)*. MIPS Technologies, Inc., 1994.

[22] MIPS Technologies, Inc. MIPS32 Architecture For Programmers. Volume I: Introduction to the MIPS32 Architecture, March 2001. Revision 0.95. Document Number MD00082.

[23] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, June 2016.