The Sail instruction-set semantics specification language

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

July 14, 2021

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.

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 [8, 9], 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 (ASL)	generated from ARM's v8.3 public ASL spec
MIPS	hand-written
CHERI	hand-written
RISC-V	hand-written

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 [10], which in turn drew on MIPS4000 and MIPS32 [11, 12]. The CHERI model is based on that and the CHERI ISA reference manual version 5 [13]. 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 by including two files from the main sail-riscv development:

```
$include "prelude.sail"
$include "riscv_xlen64.sail"
```

The prelude sets up basic definitions we will use, it can vary on a per-architecture basis to account for stylistic differences in ISA specifications. `riscv_xlen64.sail` introduces some type synonyms. It creates a integer type xlen, which is 64. Sail supports definitions which are generic over both regular types, and integers (think const generics in C++, but more expressive). We also create a type `xlenbits` for bitvectors of length `xlen`.

```
type xlen : Int = 64

type xlenbits = bits(xlen)
```

For the purpose of this example, we also introduce a type synonym for bitvectors of length 5, which represent registers.

```
type regbits = bits(5)
```

We now set up some basic architectural state. First creating a register of type `xlenbits` for both the program counter `PC`, and the next program counter, `nextPC`. We define the general purpose registers as a vector of 32 `xlenbits` 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 : xlenbits
register nextPC : xlenbits

register Xs : vector(32, dec, xlenbits)

val rX : regbits -> xlenbits effect {rreg}

function rX(r) =
  match r {
    0b00000 => EXTZ(0x0),
    _ => Xs[unsigned(r)]
  }

val wX : (regbits, xlenbits) -> unit effect {wreg}

function wX(r, v) =
  if r != 0b00000 then {
     Xs[unsigned(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 = {lem: "MEMr", coq: "MEMr", _: "read_ram"}: forall ('n 'm : Int), 'n >= 0.
  (int('m), int('n), bits('m), bits('m)) -> bits(8 * 'n) effect {rmem}
```

It's common when defining architecture specifications 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, as well as the `ast` union.

```
enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI, RISCV_ORI, RISCV_ANDI}

scattered union ast

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

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

Now we provide the clauses for the add-immediate `ast` type, as well as its execute and decode clauses. We can define the decode function by directly pattern matching on the bitvector representing the instruction. Sail supports vector concatenation 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. We use the `EXTS` library function that sign-extends its argument.

```
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 : xlenbits = EXTS(imm) in
  let result = rs1_val + imm_ext in
  X(rd) = result
```

Now we do the same thing for the load-double 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 : xlenbits = X(rs1) + EXTS(imm) in
    let result : xlenbits = read_mem(addr, sizeof(xlen_bytes)) in
    X(rd) = result
```

Finally we define the fallthrough case for the decode function. Note that the clauses in a scattered function will be matched in the order they appear in the file. 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-riscv/blob/master/model/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_mem.sail riscv_sys.sail riscv_vmem.sail riscv.sail
```

The sail files passed on the command line are simply treated as if they are one large file concatenated 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, `riscv_mem.sail` and `riscv_vmem.sail` describe the physical and virtual memory interaction, and then `riscv_sys.sail` and `riscv.sail` implement most of the specification.

For more complex projects, one can use `$include` statements 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 C compilation

To compile Sail into C, the `-c` option is used, like so:

```
sail -c FILES 1> out.c
```

The translated C is by default printed to stdout, but one can also use the `-o` option to output to a file, so

```
sail -c FILES -o out
```

will generate a file called `out.c`. To produce an executable this needs to be compiled and linked with the C files in the `sail/lib` directory:

```
gcc out.c $SAIL_DIR/lib/*.c -lgmp -lz -I $SAIL_DIR/lib/ -o out
```

The C output requires the GMP library for arbitrary precision arithmetic, as well as zlib for working with compressed ELF binaries.

There are several Sail options that affect the C output:

- `-O` turns on optimisations. The generated C code will be quite slow unless this flag is set.
- `-Oconstant_fold` apply constant folding optimisations.
- `-c_include` Supply additional header files to be included in the generated C.
- `-c_no_main` Do not generate a `main()` function.
- `-static` Mark generated C functions as static where possible. This is useful for measuring code coverage.

The generated executable for the Sail specification (provided a main function is generated) supports several options for loading ELF files and binary data into the specification memory.

- `-e/--elf` Loads an ELF file into memory. Currently only AArch64 and RISC-V ELF files are supported.
- `-b/--binary` Loads raw binary data into the specification's memory. It is used like so:

  ```
  ./out --binary=0xADDRESS,FILE
  ./out -b 0xADDRESS,FILE
  ```

  The contents of the supplied file will be placed in memory starting at the given address, which must be given as a hexadecimal number.
- `-i/--image` For ELF files that are not loadable via the `--elf` flag, they can be pre-processed by Sail using linksem into a special image file that can be loaded via this flag. This is done like so:

  ```
  sail -elf ELF_FILE -o image.bin
  ./out --image=image.bin
  ```

  The advantage of this flag is that it uses Linksem to process the ELF file, so it can handle any ELF file that linksem is able to understand. This also guarantees that the contents of the ELF binary loaded into memory is exactly the same as for the OCaml backend and the interpreter as they both also use Linksem internally to load ELF files.
- `-n/--entry` sets a custom entry point returned by the `elf_entry` function. Must be a hexadecimal address prefixed by `0x`.
- `-l/--cyclelimit` run the simulation until a set number of cycles have been reached. The main loop of the specification must call the `cycle_count` function for this to work.

3.3 Lem, Isabelle & HOL4

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.

3.4 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.

3.5 LaTeX Generation

Sail can be used to generate latex for inclusion in documentation as:

```
sail -o DIRECTORY -latex FILES
```

The list of `FILES` is a list of Sail files to produce latex for, and `DIRECTORY` is the directory where the generated latex will be placed. The list of files must be a valid type-checkable series of Sail files. The intention behind this latex generation is for it to be included within existing ISA manuals written in Latex, as such the latex output generates a list of commands for each top-level Sail declaration in `DIRECTORY/commands.tex`. The rest of this section discusses the stable features of the latex generation process—there are additional features for including markdown doc-comments in Sail code and formatting them into latex for inclusion id documentation, among other things, but these features are not completely stable yet. This manual itself makes use of the Sail latex generation, so `doc/manual.tex`, and `doc/Makefile` can be used to see how the process is set up.

Requirements The generated latex uses the *listings* package for formatting source code, uses the macros in the *etoolbox* package for the generated commands, and relies on the *hyperref* package for cross-referencing. These packages are available in most TeX distributions, and are available as part of thetexlive packages for Ubuntu.

Usage Due to the oddities of latex verbatim environments each Sail declaration must be placed in it's own file then the command in `commands.tex` includes in with `\lstinputlisting`. To include the generated Sail in a document one would do something like:

```
\input{commands.tex}
\sailtype{my_type}
\sailval{my_function}
\sailfn{my_function}
```

which would include the type definition for `my_type`, and the declaration (`\sailval`) as well as the definition (`\sailfn`) for `my_function`. These can also be used with escaped version of the names, such as `my\_type`, which is useful when defining more complex macros.

It is sometimes useful to include multiple versions of the same Sail definitions in a latex document. In this case the `-latex_prefix` option can be used. For example if we used `-latex_prefix prefix` then the above example would become:

```
\input{commands.tex}
\prefixtype{my_type}
\prefixval{my_function}
\prefixfn{my_function}
```

The generated definitions are created wrapped in customisable macros that can be overridden to change the formatting of the Sail code. For `\sailfn` there is a macro `\saildocfn` that must be defined, and similarly for the other Sail toplevel types.

Cross-referencing For each macro `\sail`X`{id}` there is a macro `\sailref`X`{id}{text}` which creates a hyper-reference to the original definition. This requires the hyper-ref package.

3.6 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 cast 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.
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

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 `my_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 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 : Int), 'm >= 'n. (implicit('m), bits('n)) -> bits('m)

function extz(m, 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:

4.2 Mappings

Mappings are a feature of Sail that allow concise expression of bidirectional relationships between values that are common in ISA specifications: for example, bit-representations of an enum type or assembly-language string representations of an instruction AST.

They are defined similarly to functions, with a `val`-spec and a definition. Currently, they only work for monomorphic types.

`val` *name* `:` $type_1$ `<->` $type_2$

`mapping` *name* `=` `{` *pattern* `<->` *pattern* `,` *pattern* `<->` *pattern* `,` ... `}`

All the functionality of pattern matching, described below, is available, including guarded patterns: but note that guards apply only to one side. This sometimes leads to unavoidable duplicated code.

As a shorthand, you can also specify a mapping and its type simultaneously.

`mapping` *name* `:` $type_1$ `<->` $type_2$ `=` `{` *pattern* `<->` *pattern* `,` *pattern* `<->` *pattern* `,` ... `}`

A simple example from our RISC-V model:

```
mapping size_bits : word_width <-> bits(2) = {
  BYTE <-> 0b00,
  HALF <-> 0b01,
  WORD <-> 0b10,
  DOUBLE <-> 0b11
}
```

Mappings are used simply by calling them as if they were functions: type inference will determine in which direction the mapping runs. (This gives rise to the restriction that the types on either side of a mapping must be different.)

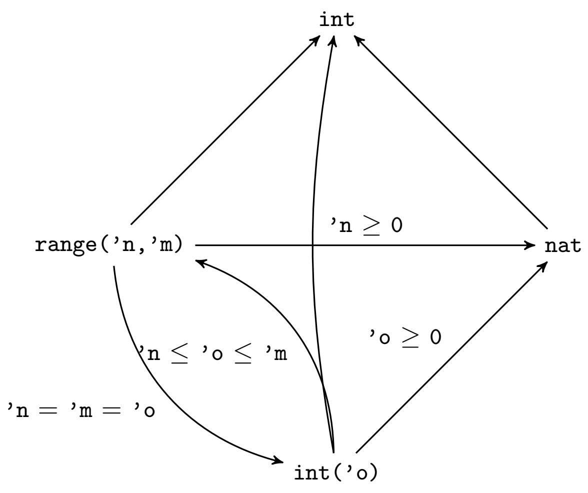
```
let width : word_width = size_bits(0b00);
let width : bits(2) = size_bits(BYTE);
```

Mappings are implemented by transforming them at compile time into a forwards and a backwards function, along with some auxiliary functions. Once a mapping is declared with a `val`-spec, it can be implemented by defining these functions manually instead of defining the mapping as above. These functions and their types are:

```
val name_forwards : type_1 -> type_2
val name_backwards : type_2 -> type_1
val name_forwards_matches : type_1 -> bool
val name_backwards_matches : type_2 -> bool
```

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).


$$\begin{tikzcd}
 & \texttt{int} & \\
\texttt{range('n,'m)} \arrow[ur] \arrow[rr, "\texttt{'n} \geq 0"] \arrow[dr, bend right, "\texttt{'n} = \texttt{'m} = \texttt{'o}"'] & & \texttt{nat} \arrow[ul] \\
 & \texttt{int('o)} \arrow[uu, bend left] \arrow[ul, bend right, "\texttt{'n} \leq \texttt{'o} \leq \texttt{'m}"] \arrow[ur, "\texttt{'o} \geq 0"'] &
\end{tikzcd}$$

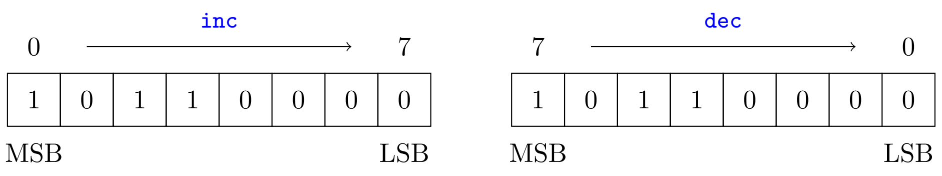
Note that `bit` isn't a numeric type (i.e. it's not `range(0,1)`. This is intentional, as otherwise it would be possible to write expressions like `(1 : bit)+ 5` which would end up being equal to `6 : range(5, 6)`. This kind of implicit casting from bits to other numeric types would be highly undesirable. 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` (note 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 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 checked 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"),
  0b0000_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")
}
```

Matching on strings Unusually, Sail allows strings and concatenations of strings to be used in pattern-matching. The match operates in a simple left-to-right fashion.

```
match s {
  "hello" ^ "␣" ^ "world" => print("matched␣hello␣world"),
  _ => print("wildcard")
}
```

Note that a string match is always greedy, so

```
match s {
  "hello" ^ s ^ "world" => print("matched␣hello" ^ s ^ "world"),
  "hello" ^ s => print("matched␣hello" ^ s),
  _ => print("wildcard")
}
```

while syntactically valid, will never match the first case.

String matching is most often used with *mappings*, covered above, to allow parsing of strings containing, for example, integers:

```
match s {
  "int=" ^ int(x) ^ ";" => x
  _ => -1
}
```

This is intended to be used to parse assembly languages.

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 zs 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;
```

[4]although this is not checked right now

```
    ⋮
    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 always 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.

```
{
  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 right 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[3 .. 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:

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

```
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:

```
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 implicitly 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,
```

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

```
  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.

4.10.3 Unions

As an example, the `maybe` type à 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 4.8.1. 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.

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

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 4 <=_u
```

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)
```

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

`overload` *name* = { $name_1$, ..., $name_n$ }

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 its 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:

$$\begin{array}{c}\texttt{overload}\ \mathit{name}\ \texttt{=}\ \texttt{\{}\ \mathit{name}_1\ \texttt{\}}\\ \vdots \\ \texttt{overload}\ \mathit{name}\ \texttt{=}\ \texttt{\{}\ \mathit{name}_n\ \texttt{\}}\end{array}$$

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 their 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 equal to zero. If we overload these definitions as

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

Then the first will be applied when the constraints on its inputs can be resolved, and therefore the guarantees on its 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 overload.

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 mentioned 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 equivalent test `length(xs)<= length(ys)`. This way of writing expressions can be succinct, 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 surprisingly 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-pseudocodes 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 its 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 Sail Internals

5.1 Abstract Syntax Tree

The Sail abstract syntax tree (AST) is defined by an ott grammar in sail.ott. This generates a OCaml (and lem) version of the ast in `src/ast.ml` and `src/ast.lem`. Technically the OCaml version of the AST is generated by Lem, which allows the Sail OCaml source to seamlessly interoperate with parts written in Lem. Although we do not make much use of this, in principle it allows us to implement parts of Sail in Lem, which would enable them to be verified in Isabelle or HOL4.

The Sail AST looks something like:

```
and 'a exp =
 | E_aux of ( 'a exp_aux) * 'a annot

and 'a exp_aux = (* expression *)
 | E_block of ( 'a exp) list (* sequential block *)
 | E_id of id (* identifier *)
 | E_lit of lit (* literal constant *)
 | E_cast of typ * ( 'a exp) (* cast *)
 | E_app of id * ( 'a exp) list (* function application *)
 | E_tuple of ( 'a exp) list (* tuple *)
 | E_if of ( 'a exp) * ( 'a exp) * ( 'a exp) (* conditional *)
 ...
```

Each constructor is prefixed by a unique code (in this case `E` for expressions), and is additionally wrapped by an *aux* constructor which attaches an annotation to each node in the AST, consisting of an arbitrary type that parameterises the AST, and a location identifying the position of the AST node in the source code:

```
type 'a annot = l * 'a
```

There are various types of locations:

```
type l =
  | Unknown
  | Unique of int * l
  | Generated of l
  | Range of Lexing.position * Lexing.position
  | Documented of string * l
```

`Range` defines a range of positions given by the parser, the `Documented` constructor attaches doc-comments to locations. `Generated` is used to signify that code is generated based on code from some other location. `Unique` is used internally to tag locations with unique integers so they can be referred to later. `Unknown` is used for Sail that has no obvious corresponding location, although this should be avoided as much possible as it leads to poor error messages. Ast nodes programatically generated initially have `Unknown` locations, but `Ast_util.locate` can be used to recursively attach `Generated` locations for error purposes.

A convention in the Sail source is that a single variable `l` is always a location, usually the closest location.

5.1.1 AST utilities

There are various functions for manipulating the AST defined in ast_util. These include constructor functions like `mk_exp` for generating untyped expressions and various functions for destructuring AST nodes. It also defines various useful Sets and Maps, e.g. sets of identifiers, type variables etc. Note that type variables in Sail are often internally referred to as *kids*, I think this is because type variables are defined as having a specific *kind*, i.e. `'n : Int` is a type variable with kind `Int`. (Although `kinded_id` is technically a separate type which is a type variable * kind pair).

5.1.2 Parse AST

There is a separate AST parse_ast.ml which the parser generates. It is very similar to the main AST except it contains some additional syntactic sugar. The parse ast is desugared by the initial_check file, which performs some basic checks in addition from mapping the parse AST to the main AST.

5.2 Overall Structure

The main entry point for Sail is the file sail.ml. Each backend option e.g. `-c`, `-lem`, `-ocaml` etc, is referred to as a *target*, and the `set_target` function is used to set the `opt_target` variable, for example

```
( "-c",
  Arg.Tuple [set_target "c"; Arg.Set Initial_check.opt_undefined_gen],
  "␣output␣a␣C␣translated␣version␣of␣the␣input");
```

defines the `-c` option. Each Sail option is configured via via `opt_` variables defined at the top of each relevant file. In this case we tell Sail that when we generate C, we also want to generated `undefined_X` functions for each type `X`. In general such `opt_` variables should be set when we start Sail and remain immutable thereafter.

The first function called by `main` is `Sail.load_files`. This function parses all the files passed to Sail, and then concatenates their ASTs. The pre-processor is then run, which evaluates `$directive` statements in Sail, such as

```
$include <prelude.sail>
```

Unlike the C pre-processor the Sail pre-processor operates over actual Sail ASTs rather than strings. This can recursively include other files into the AST, as well as add/remove parts of the AST with `$ifdef` etc. Directives that are used are preserved in the AST, so they also function as a useful way to pass auxiliary information to the various Sail backends.

The initial check mentioned above is then run to desugar the AST, and then the type-checker is run which produces a fully type-checked AST. Type annotations are attached to every node (for which an annotation makes sense) using the aux constructors. The type-checker is discussed in more details later.

After type-checking the Sail scattered definitions are de-scattered into single functions.

All the above is shared by each target and performed by the `load_files` function.

The next step is to apply various target-specific rewrites to the AST before passing it to the backend for each target.

The file rewrites.ml defines a list of rewrites:

```
let all_rewrites = [
    ("pat_string_append", Basic_rewriter rewrite_defs_pat_string_append);
    ("mapping_builtins", Basic_rewriter rewrite_defs_mapping_patterns);
    ("mono_rewrites", Basic_rewriter mono_rewrites);
    ("toplevel_nexps", Basic_rewriter rewrite_toplevel_nexps);
    ("monomorphise", Basic_rewriter monomorphise);
    ...
```

and each target specifies a list of rewrites to apply like so:

```
let rewrites_interpreter = [
    ("no_effect_check", []);
    ("realise_mappings", []);
    ("toplevel_string_append", []);
    ("pat_string_append", []);
    ("mapping_builtins", []);
    ("undefined", [Bool_arg false]);
    ("vector_concat_assignments", []);
    ("tuple_assignments", []);
    ("simple_assignments", [])
  ]
```

Once these rewrites have occurred the `Sail.target` function is called which invokes the backend for each target, e.g. for OCaml:

```
| Some "ocaml" ->
   let ocaml_generator_info =
     match !opt_ocaml_generators with
     | [] -> None
     | _ -> Some (Ocaml_backend.orig_types_for_ocaml_generator ast, !opt_ocaml_generators)
   in
   let out = match !opt_file_out with None -> "out" | Some s -> s in
   Ocaml_backend.ocaml_compile out ast ocaml_generator_info
```

There is also a `Sail.prover_regstate` function that allows the register state to be set up in a prover-agnostic way for each of the theorem-prover targets.

5.3 Type Checker

The Sail type-checker is contained within src/type_check.ml. This file is long, but is structured as follows:

1. The type checking environment is defined. The functions that operate on the typing environment are contained with a separate `Env` module. The purpose of this module is to hide the internal representation of the typing environment and ensure that the main type-checker code can only interact with it using the functions defined in this module, which can be set up to guarantee any invariant we need. Code outside the type-checker itself can only interact with an even more restricted subset of these functions exported via the mli interface file.

2. Helper definitions for subtyping and constraint solving are set up. The code that actually talks to an external SMT solver is contained within a separate file src/constraint.ml, whereas the code here sets up the interface between the typing environment and the solver.

3. Next comes the definitions for unification and instantiation of types. There is some additional code (3.5) to handle subtyping with existential types, which can use unification to instantiate existentially quantified type variables.

4. Sail allows some type-level constructs to appear in term-level variables, but these are then eliminated during type-checking in a process called *sizeof*-rewriting, after the (somewhat-awkwardly) named *sizeof* keyword.

5. Finally all the typing rules are given. Sail has a bi-directional type-checking approach. So there are checking rules like `check_exp`, `check_pat`, etc., as well as inference rules like `infer_exp`, `infer_pat`, etc.

6. Effects added by the previous typing rules are now propagated upwards through the AST

7. Finally we have rules for checking and processing toplevel definitions of functions and datatypes.

The interface by which the rest of Sail can interact with the type-checker and type annotations is strictly controlled by it's mli interface file. We try to keep much of the type-checking internals as abstract as possible.

5.4 Rewriter

The rewriting framework used by the various rewrites mentioned previously is defined in src/rewriter.ml. It contains various large structs with functions for every single AST node, and allows data to be threaded through each re-write in various ways. Most of the re-writes are defined in src/rewrites.ml, although the re-writer is used for other rewrite like passes such as e.g. constant folding in src/constant_fold.ml which combines the rewriter with the Sail interpreter.

The rewriter also acts as the interface to the Sail monomorphisation code, in src/monomorphise.ml.

5.5 Jib

The C and SMT backends for Sail use a custom intermediate representation (IR) called *Jib* (it's a type of Sail). Like the full AST this is defined as an ott grammar in language/jib.ott. The sail "-ir" target allows Sail to be converted into this IR without any further processing.

The Jib related files are contained within a sub-directory `src/jib/`. First we convert Sail to A-normal form (ANF) in src/jib/anf.ml, then src/jib/jib_compile.ml converts this into Jib.

The Jib representation has the advantage of being much simpler than the full Sail AST, and it removes a lot of the typing complexity, as types are replaced with simpler ones (`ctyp`). Note that many Jib-related types are prefixed by `c` as it was originally only used when generating C.

The key optimisation we do when generating Jib is we analyse the Sail types using SMT to try to fit arbitrary-precision types into smaller fixed-precision machine-word types that exist within Jib. To aid in this we have a specialisation pass that removes polymorphism by creating specialised copies of functions based on how their type-quantifiers are instantiated. This can be used in addition to the Sail monomorphisation above.

Once we have generated Jib, the code generator from Jib into C is fairly straightforward.

5.6 Jib to SMT translation

Starting with some Sail such as:

```
default order dec

$include <prelude.sail>

register r : bits(32)

$property
function property(xs: bits(32)) -> bool = {
  ys : bits(32) = 0x1234_5678;
  if (r[0] == bitzero) then {
    ys = 0xffff_ffff
  } else {
    ys = 0x0000_0000
  };
  (ys == sail_zeros(32) | ys == sail_ones(32))
}
```

We first compile to Jib, then inline all functions and flatten the resulting code into a list of instructions as below. The Sail->Jib step can be parameterised in a few ways so it differs slightly to when we compile Sail to C. First, a specialisation pass specialises integer-polymorphic functions and builtins, which is reflected in the name mangling scheme so, e.g.

```
zz7mzJzK0zCz0z7nzJzK32#bitvector_access
```

is a specialised version of bitvector access for 'n => 32 & 'm => 0. This lets us generate optimal SMT operations for monomorphic code, as the SMTLIB operations like ZeroExtend and Extract are only defined for natural number constants and bitvectors of known lengths. We also have to treat zero-length bitvectors differently to C, as SMT does not allow zero-length bitvectors, and unlike when we compile to C, we can have fixed-precision bitvectors of greater that 64-bits in the generated Jib.

```
var ys#u12_l#9 : fbits(32, dec)
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0x12345678)
var gs#2#u12_l#15 : bool
var gs#1#u12_l#17 : bit
gs#1#u12_l#17 : bit = zz7mzJzK0zCz0z7nzJzK32#bitvector_access(R, 0l)
gs#2#u12_l#15 : bool = eq_bit(gs#1#u12_l#17, UINT64_C(0))
kill gs#1#u12_l#17 : bit
var gs#6#u12_l#16 : unit
jump gs#2#u12_l#15 then_13
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0x00000000)
```

```
gs#6#u12_l#16 : unit = UNIT
goto endif_14
then_13:
ys#u12_l#9 : fbits(32, dec) = UINT64_C(0xFFFFFFFF)
gs#6#u12_l#16 : unit = UNIT
endif_14:
kill gs#2#u12_l#15 : bool
var gs#5#u12_l#10 : bool
var gs#3#u12_l#14 : fbits(32, dec)
gs#3#u12_l#14 : fbits(32, dec) = zz7nzJzK32#sail_zeros(32l)
gs#5#u12_l#10 : bool = zz7nzJzK32#eq_bits(ys#u12_l#9, gs#3#u12_l#14)
kill gs#3#u12_l#14 : fbits(32, dec)
var gs#7#u12_l#11 : bool
jump gs#5#u12_l#10 then_11
var gs#4#u12_l#12 : fbits(32, dec)
var gs#0#u9_l#13 : fbits(32, dec)
gs#0#u9_l#13 : fbits(32, dec) = zz7nzJzK32#sail_zeros(32l)
gs#4#u12_l#12 : fbits(32, dec) = zz7nzJzK32#not_vec(gs#0#u9_l#13)
kill gs#0#u9_l#13 : fbits(32, dec)
goto end_inline_10
end_inline_10:
gs#7#u12_l#11 : bool = zz7nzJzK32#eq_bits(ys#u12_l#9, gs#4#u12_l#12)
kill gs#4#u12_l#12 : fbits(32, dec)
goto endif_12
then_11:
gs#7#u12_l#11 : bool = true
endif_12:
return : bool = gs#7#u12_l#11
kill gs#5#u12_l#10 : bool
kill ys#u12_l#9 : fbits(32, dec)
end
undefined bool
```

The above Jib is then turned into a control-flow-graph in SSA form.

The conditionals that affect control-flow are put into separate nodes (in yellow), so we can easily compute the global path conditional for each block (stored in a function pi(cond0, ... , condn) by using the yellow conditional nodes between each node and the start node.

From this form the conversion to SMT is as follows:

A variable declaration such as

```
var x : fbits(32, dec)
```

would become

```
(declare-const x (BitVec 32))
```

A call to a builtin

```
x : T = f(y0, ... , yn)
```

would become

```
(define-const x T' exp)
```

where exp encodes the builtin f using SMT bitvector operations

Phi functions are mapped to muxers as follows - for a function `phi(x0,...,xn)` we turn that into an if-then-else statement which selects `x0` to `xn` based on the global path conditionals of the parent blocks corresponding to each argument. Each phi function in a block always has the same number of arguments as the number of parent nodes, and the arguments are in the same order as the node index for each parent.

The above scheme generates a lot of declare-const and define-const statements that may not be needed so we do some simple dead-code elimination and constant propagation, which results in the following SMT:

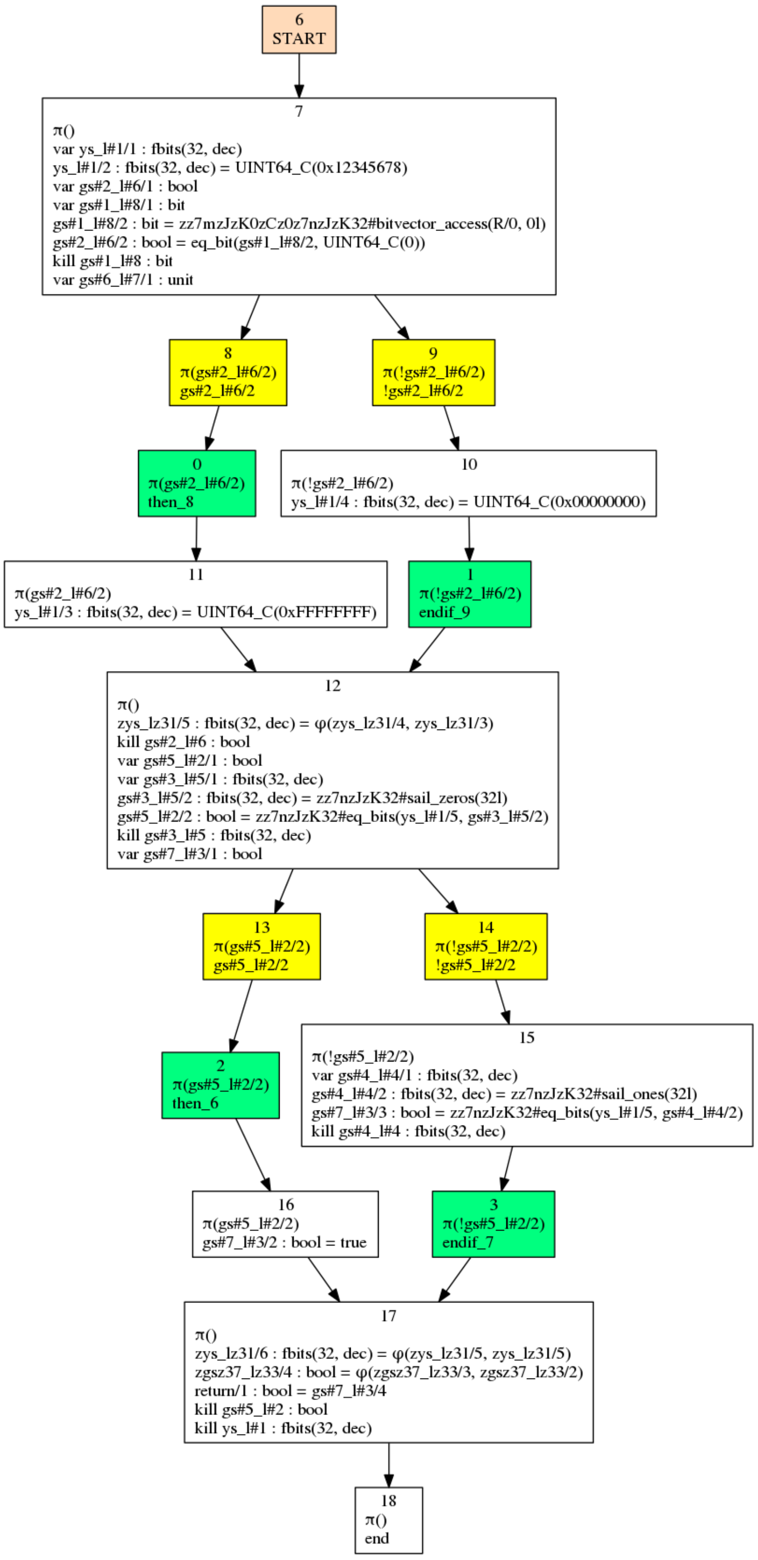

6
START
7
π()
var ys_l#1/1 : fbits(32, dec)
ys_l#1/2 : fbits(32, dec) = UINT64_C(0x12345678)
var gs#2_l#6/1 : bool
var gs#1_l#8/1 : bit
gs#1_l#8/2 : bit = zz7mzJzK0zCz0z7nzJzK32#bitvector_access(R/0, 0l)
gs#2_l#6/2 : bool = eq_bit(gs#1_l#8/2, UINT64_C(0))
kill gs#1_l#8 : bit
var gs#6_l#7/1 : unit
8
π(gs#2_l#6/2)
gs#2_l#6/2
9
π(!gs#2_l#6/2)
!gs#2_l#6/2
0
π(gs#2_l#6/2)
then_8
10
π(!gs#2_l#6/2)
ys_l#1/4 : fbits(32, dec) = UINT64_C(0x00000000)
11
π(gs#2_l#6/2)
ys_l#1/3 : fbits(32, dec) = UINT64_C(0xFFFFFFFF)
1
π(!gs#2_l#6/2)
endif_9
12
π()
zys_lz31/5 : fbits(32, dec) = φ(zys_lz31/4, zys_lz31/3)
kill gs#2_l#6 : bool
var gs#5_l#2/1 : bool
var gs#3_l#5/1 : fbits(32, dec)
gs#3_l#5/2 : fbits(32, dec) = zz7nzJzK32#sail_zeros(32l)
gs#5_l#2/2 : bool = zz7nzJzK32#eq_bits(ys_l#1/5, gs#3_l#5/2)
kill gs#3_l#5 : fbits(32, dec)
var gs#7_l#3/1 : bool
13
π(gs#5_l#2/2)
gs#5_l#2/2
14
π(!gs#5_l#2/2)
!gs#5_l#2/2
2
π(gs#5_l#2/2)
then_6
15
π(!gs#5_l#2/2)
var gs#4_l#4/1 : fbits(32, dec)
gs#4_l#4/2 : fbits(32, dec) = zz7nzJzK32#sail_ones(32l)
gs#7_l#3/3 : bool = zz7nzJzK32#eq_bits(ys_l#1/5, gs#4_l#4/2)
kill gs#4_l#4 : fbits(32, dec)
16
π(gs#5_l#2/2)
gs#7_l#3/2 : bool = true
3
π(!gs#5_l#2/2)
endif_7
17
π()
zys_lz31/6 : fbits(32, dec) = φ(zys_lz31/5, zys_lz31/5)
zgsz37_lz33/4 : bool = φ(zgsz37_lz33/3, zgsz37_lz33/2)
return/1 : bool = gs#7_l#3/4
kill gs#5_l#2 : bool
kill ys_l#1 : fbits(32, dec)
18
π()
end

Figure 1: SSA graph

```
(set-logic QF_AUFBVDT)
(declare-const zR/0 (_ BitVec 32))
(define-const zysz3u12_lz30/5 (_ BitVec 32) (ite (not (= ((_ extract 0 0) zR/0) #b0)) #
    ↪ x00000000 #xFFFFFFFF))
(assert (and (not (ite (not (= zysz3u12_lz30/5 #x00000000)) (= zysz3u12_lz30/5 (bvnot #
    ↪ x00000000)) true))))
(check-sat)
```

For monomorphic-bitvector manipulating code we can generate very compact SMT. Both specialisation and monomorphisation can be used to help monomorphise bitvectors. For variable-length bitvectors we can represent them as length-bitvector pairs, up to a certain bounded max length (default 256). This is less efficient but unavoidable in certain places. Integers are currently mapped to either 128 bit bitvectors (or any configurable max length) or 64 bit bitvectors.

The one slightly tricky thing to support is register references, e.g.

```
(*r) : T = 0xFFFF_FFFF
```

where `r` is a register reference. For this we look for all registers with type T (as a Jib type, where type-equality is trivial), and convert the above to

```
if r = ref R1 then
  R1 = 0xFFFF_FFFF
else if r = ref R2 then
  R2 = 0xFFFF_FFFF
else ...
```

if we did some smarter constant folding (e.g. propagating the GPRs letbinding in some of our specs) this could potentially generate SMT that is just as good as a manual if-then-else implementation of the register read/write functions. This step is done before converting to SSA so each register in the if-then-else cascade gets the correct indices.

A Sail Grammar

This appendix contains a grammar for the Sail language that is automatically generated from the Menhir parser definition. First, we need some lexical conventions:

Id	::=	`[a-zA-Z?_][a-zA-Z?_0-9']`*
TyVar	::=	`'[a-zA-Z?_][a-zA-Z?_0-9']`*
Bin	::=	`0b[01_]`$^+$
Hex	::=	`0x[0-9A-Fa-f_]`$^+$
Num	::=	`-`$^?$`[0-9]`$^+$
Real	::=	`-`$^?$`[0-9]`$^+$`.[0-9]`$^+$
String	::=	`"..."`

Now the actual grammar:

⟨id⟩	::=	Id
	|	`operator` Op0
	|	`operator` Op1
	|	`operator` Op2
	|	`operator` Op3
	|	`operator` Op4
	|	`operator` Op5
	|	`operator` Op6
	|	`operator` Op7
	|	`operator` Op8
	|	`operator` Op9
	|	`operator` Op0l
	|	`operator` Op1l
	|	`operator` Op2l
	|	`operator` Op3l
	|	`operator` Op4l
	|	`operator` Op5l
	|	`operator` Op6l
	|	`operator` Op7l
	|	`operator` Op8l
	|	`operator` Op9l
	|	`operator` Op0r
	|	`operator` Op1r
	|	`operator` Op2r
	|	`operator` Op3r
	|	`operator` Op4r
	|	`operator` Op5r
	|	`operator` Op6r
	|	`operator` Op7r
	|	`operator` Op8r
	|	`operator` Op9r
	|	`operator +`
	|	`operator -`
	|	`operator *`
	|	`operator ==`
	|	`operator !=`
	|	`operator <`
	|	`operator >`
	|	`operator <=`
	|	`operator >=`
	|	`operator &`
	|	`operator |`

	|	`operator ^`
⟨op0⟩	::=	Op0
⟨op1⟩	::=	Op1
⟨op2⟩	::=	Op2
⟨op3⟩	::=	Op3
⟨op4⟩	::=	Op4
⟨op5⟩	::=	Op5
⟨op6⟩	::=	Op6
⟨op7⟩	::=	Op7
⟨op8⟩	::=	Op8
⟨op9⟩	::=	Op9
⟨op0l⟩	::=	Op0l
⟨op1l⟩	::=	Op1l
⟨op2l⟩	::=	Op2l
⟨op3l⟩	::=	Op3l
⟨op4l⟩	::=	Op4l
⟨op5l⟩	::=	Op5l
⟨op6l⟩	::=	Op6l
⟨op7l⟩	::=	Op7l
⟨op8l⟩	::=	Op8l
⟨op9l⟩	::=	Op9l
⟨op0r⟩	::=	Op0r
⟨op1r⟩	::=	Op1r
⟨op2r⟩	::=	Op2r
⟨op3r⟩	::=	Op3r
⟨op4r⟩	::=	Op4r
⟨op5r⟩	::=	Op5r
⟨op6r⟩	::=	Op6r

```
     ⟨op7r⟩  ::=  Op7r

     ⟨op8r⟩  ::=  Op8r

     ⟨op9r⟩  ::=  Op9r

      ⟨kid⟩  ::=  TyVar

   ⟨lchain⟩  ::=  ⟨typ5⟩ <= ⟨typ5⟩
              |   ⟨typ5⟩ < ⟨typ5⟩
              |   ⟨typ5⟩ <= ⟨lchain⟩
              |   ⟨typ5⟩ < ⟨lchain⟩

   ⟨rchain⟩  ::=  ⟨typ5⟩ >= ⟨typ5⟩
              |   ⟨typ5⟩ > ⟨typ5⟩
              |   ⟨typ5⟩ >= ⟨rchain⟩
              |   ⟨typ5⟩ > ⟨rchain⟩

    ⟨tyarg⟩  ::=  ( ⟨typ⟩+, )

  ⟨typ_eof⟩  ::=  ⟨typ⟩ Eof

      ⟨typ⟩  ::=  ⟨typ0⟩

     ⟨typ0⟩  ::=  ⟨typ1⟩ ⟨op0⟩ ⟨typ1⟩
              |   ⟨typ0l⟩ ⟨op0l⟩ ⟨typ1⟩
              |   ⟨typ1⟩ ⟨op0r⟩ ⟨typ0r⟩
              |   ⟨typ1⟩

    ⟨typ0l⟩  ::=  ⟨typ1⟩ ⟨op0⟩ ⟨typ1⟩
              |   ⟨typ0l⟩ ⟨op0l⟩ ⟨typ1⟩
              |   ⟨typ1⟩

    ⟨typ0r⟩  ::=  ⟨typ1⟩ ⟨op0⟩ ⟨typ1⟩
              |   ⟨typ1⟩ ⟨op0r⟩ ⟨typ0r⟩
              |   ⟨typ1⟩

     ⟨typ1⟩  ::=  ⟨typ2⟩ ⟨op1⟩ ⟨typ2⟩
              |   ⟨typ1l⟩ ⟨op1l⟩ ⟨typ2⟩
              |   ⟨typ2⟩ ⟨op1r⟩ ⟨typ1r⟩
              |   ⟨typ2⟩

    ⟨typ1l⟩  ::=  ⟨typ2⟩ ⟨op1⟩ ⟨typ2⟩
              |   ⟨typ1l⟩ ⟨op1l⟩ ⟨typ2⟩
              |   ⟨typ2⟩

    ⟨typ1r⟩  ::=  ⟨typ2⟩ ⟨op1⟩ ⟨typ2⟩
              |   ⟨typ2⟩ ⟨op1r⟩ ⟨typ1r⟩
              |   ⟨typ2⟩

     ⟨typ2⟩  ::=  ⟨typ3⟩ ⟨op2⟩ ⟨typ3⟩
              |   ⟨typ2l⟩ ⟨op2l⟩ ⟨typ3⟩
              |   ⟨typ3⟩ ⟨op2r⟩ ⟨typ2r⟩
              |   ⟨typ3⟩ | ⟨typ2r⟩
              |   ⟨typ3⟩
```

```
⟨typ2l⟩  ::=  ⟨typ3⟩ ⟨op2⟩ ⟨typ3⟩
           |  ⟨typ2l⟩ ⟨op2l⟩ ⟨typ3⟩
           |  ⟨typ3⟩

⟨typ2r⟩  ::=  ⟨typ3⟩ ⟨op2⟩ ⟨typ3⟩
           |  ⟨typ3⟩ ⟨op2r⟩ ⟨typ2r⟩
           |  ⟨typ3⟩ | ⟨typ2r⟩
           |  ⟨typ3⟩

 ⟨typ3⟩  ::=  ⟨typ4⟩ ⟨op3⟩ ⟨typ4⟩
           |  ⟨typ3l⟩ ⟨op3l⟩ ⟨typ4⟩
           |  ⟨typ4⟩ ⟨op3r⟩ ⟨typ3r⟩
           |  ⟨typ4⟩ & ⟨typ3r⟩
           |  ⟨typ4⟩

⟨typ3l⟩  ::=  ⟨typ4⟩ ⟨op3⟩ ⟨typ4⟩
           |  ⟨typ3l⟩ ⟨op3l⟩ ⟨typ4⟩
           |  ⟨typ4⟩

⟨typ3r⟩  ::=  ⟨typ4⟩ ⟨op3⟩ ⟨typ4⟩
           |  ⟨typ4⟩ ⟨op3r⟩ ⟨typ3r⟩
           |  ⟨typ4⟩ & ⟨typ3r⟩
           |  ⟨typ4⟩

 ⟨typ4⟩  ::=  ⟨typ5⟩ ⟨op4⟩ ⟨typ5⟩
           |  ⟨typ4l⟩ ⟨op4l⟩ ⟨typ5⟩
           |  ⟨typ5⟩ ⟨op4r⟩ ⟨typ4r⟩
           |  ⟨lchain⟩
           |  ⟨rchain⟩
           |  ⟨typ5⟩ == ⟨typ5⟩
           |  ⟨typ5⟩ != ⟨typ5⟩
           |  ⟨typ5⟩

⟨typ4l⟩  ::=  ⟨typ5⟩ ⟨op4⟩ ⟨typ5⟩
           |  ⟨typ4l⟩ ⟨op4l⟩ ⟨typ5⟩
           |  ⟨typ5⟩

⟨typ4r⟩  ::=  ⟨typ5⟩ ⟨op4⟩ ⟨typ5⟩
           |  ⟨typ5⟩ ⟨op4r⟩ ⟨typ4r⟩
           |  ⟨typ5⟩

 ⟨typ5⟩  ::=  ⟨typ6⟩ ⟨op5⟩ ⟨typ6⟩
           |  ⟨typ5l⟩ ⟨op5l⟩ ⟨typ6⟩
           |  ⟨typ6⟩ ⟨op5r⟩ ⟨typ5r⟩
           |  ⟨typ6⟩

⟨typ5l⟩  ::=  ⟨typ6⟩ ⟨op5⟩ ⟨typ6⟩
           |  ⟨typ5l⟩ ⟨op5l⟩ ⟨typ6⟩
           |  ⟨typ6⟩

⟨typ5r⟩  ::=  ⟨typ6⟩ ⟨op5⟩ ⟨typ6⟩
           |  ⟨typ6⟩ ⟨op5r⟩ ⟨typ5⟩
           |  ⟨typ6⟩

 ⟨typ6⟩  ::=  ⟨typ7⟩ ⟨op6⟩ ⟨typ7⟩
           |  ⟨typ6l⟩ ⟨op6l⟩ ⟨typ7⟩
```

```
              |    ⟨typ7⟩ ⟨op6r⟩ ⟨typ6r⟩
              |    ⟨typ6l⟩ + ⟨typ7⟩
              |    ⟨typ6l⟩ - ⟨typ7⟩
              |    ⟨typ7⟩

   ⟨typ6l⟩   ::=   ⟨typ7⟩ ⟨op6⟩ ⟨typ7⟩
              |    ⟨typ6l⟩ ⟨op6l⟩ ⟨typ7⟩
              |    ⟨typ6l⟩ + ⟨typ7⟩
              |    ⟨typ6l⟩ - ⟨typ7⟩
              |    ⟨typ7⟩

   ⟨typ6r⟩   ::=   ⟨typ7⟩ ⟨op6⟩ ⟨typ7⟩
              |    ⟨typ7⟩ ⟨op6r⟩ ⟨typ6r⟩
              |    ⟨typ7⟩

    ⟨typ7⟩   ::=   ⟨typ8⟩ ⟨op7⟩ ⟨typ8⟩
              |    ⟨typ7l⟩ ⟨op7l⟩ ⟨typ8⟩
              |    ⟨typ8⟩ ⟨op7r⟩ ⟨typ7r⟩
              |    ⟨typ7l⟩ * ⟨typ8⟩
              |    ⟨typ8⟩

   ⟨typ7l⟩   ::=   ⟨typ8⟩ ⟨op7⟩ ⟨typ8⟩
              |    ⟨typ7l⟩ ⟨op7l⟩ ⟨typ8⟩
              |    ⟨typ7l⟩ * ⟨typ8⟩
              |    ⟨typ8⟩

   ⟨typ7r⟩   ::=   ⟨typ8⟩ ⟨op7⟩ ⟨typ8⟩
              |    ⟨typ8⟩ ⟨op7r⟩ ⟨typ7r⟩
              |    ⟨typ8⟩

    ⟨typ8⟩   ::=   ⟨typ9⟩ ⟨op8⟩ ⟨typ9⟩
              |    ⟨typ8l⟩ ⟨op8l⟩ ⟨typ9⟩
              |    ⟨typ9⟩ ⟨op8r⟩ ⟨typ8r⟩
              |    2 ^ ⟨typ9⟩
              |    - ⟨typ9⟩
              |    ⟨typ9⟩

   ⟨typ8l⟩   ::=   ⟨typ9⟩ ⟨op8⟩ ⟨typ9⟩
              |    ⟨typ8l⟩ ⟨op8l⟩ ⟨typ9⟩
              |    2 ^ ⟨typ9⟩
              |    - ⟨typ9⟩
              |    ⟨typ9⟩

   ⟨typ8r⟩   ::=   ⟨typ9⟩ ⟨op8⟩ ⟨typ9⟩
              |    ⟨typ9⟩ ⟨op8r⟩ ⟨typ8r⟩
              |    2 ^ ⟨typ9⟩
              |    - ⟨typ9⟩
              |    ⟨typ9⟩

    ⟨typ9⟩   ::=   ⟨kid⟩ in { Num$^{+}_{,}$ }
              |    ⟨atomic_typ⟩ ⟨op9⟩ ⟨atomic_typ⟩
              |    ⟨typ9l⟩ ⟨op9l⟩ ⟨atomic_typ⟩
              |    ⟨atomic_typ⟩ ⟨op9r⟩ ⟨typ9r⟩
              |    ⟨atomic_typ⟩

   ⟨typ9l⟩   ::=   ⟨atomic_typ⟩ ⟨op9⟩ ⟨atomic_typ⟩
```

| ⟨typ9l⟩ ⟨op9l⟩ ⟨atomic_typ⟩
| ⟨atomic_typ⟩

⟨typ9r⟩ ::= ⟨atomic_typ⟩ ⟨op9⟩ ⟨atomic_typ⟩
| ⟨atomic_typ⟩ ⟨op9r⟩ ⟨typ9r⟩
| ⟨atomic_typ⟩

⟨atomic_typ⟩ ::= ⟨id⟩
| _
| ⟨kid⟩
| ⟨lit⟩
| dec
| inc
| ⟨id⟩ ⟨tyarg⟩
| register (⟨typ⟩)
| (⟨typ⟩)
| (⟨typ⟩ , ⟨typ⟩$^{+}_{,}$)
| {| Num$^{+}_{,}$ |}
| { ⟨kopt⟩$^{+}$. ⟨typ⟩ }
| { ⟨kopt⟩$^{+}$, ⟨typ⟩ . ⟨typ⟩ }
| { ⟨id⟩ : ⟨typ⟩ . ⟨typ⟩ }

⟨kind⟩ ::= Int
| Type
| Order
| Bool

⟨kopt⟩ ::= (constant ⟨kid⟩$^{+}$: ⟨kind⟩)
| (⟨kid⟩$^{+}$: ⟨kind⟩)
| ⟨kid⟩

⟨typquant⟩ ::= ⟨kopt⟩$^{+}$, ⟨typ⟩
| ⟨kopt⟩$^{+}$

⟨effect⟩ ::= barr
| depend
| rreg
| wreg
| rmem
| rmemt
| wmem
| wmv
| wmvt
| eamem
| exmem
| undef
| unspec
| nondet
| escape
| configuration

⟨effect_set⟩ ::= { ⟨effect⟩$^{+}_{,}$ }
| pure

⟨typschm⟩ ::= ⟨typ⟩ -> ⟨typ⟩
| forall ⟨typquant⟩ . ⟨typ⟩ -> ⟨typ⟩

	|	⟨typ⟩ `->` ⟨typ⟩ `effect` ⟨effect_set⟩
	|	`forall` ⟨typquant⟩ `.` ⟨typ⟩ `->` ⟨typ⟩ `effect` ⟨effect_set⟩
	|	⟨typ⟩ `<->` ⟨typ⟩
	|	`forall` ⟨typquant⟩ `.` ⟨typ⟩ `<->` ⟨typ⟩
	|	⟨typ⟩ `<->` ⟨typ⟩ `effect` ⟨effect_set⟩
	|	`forall` ⟨typquant⟩ `.` ⟨typ⟩ `<->` ⟨typ⟩ `effect` ⟨effect_set⟩
⟨typschm_eof⟩	::=	⟨typschm⟩ Eof
⟨pat1⟩	::=	⟨atomic_pat⟩
	|	⟨atomic_pat⟩ `@` $\langle\text{atomic\_pat}\rangle^{+}_{\texttt{@}}$
	|	⟨atomic_pat⟩ `::` ⟨pat1⟩
	|	⟨atomic_pat⟩ `^` $\langle\text{atomic\_pat}\rangle^{+}_{\texttt{\^{}}}$
⟨pat⟩	::=	⟨pat1⟩
	|	⟨pat1⟩ `as` ⟨typ⟩
	|	⟨pat1⟩ `match` ⟨typ⟩
⟨atomic_pat⟩	::=	`_`
	|	⟨lit⟩
	|	⟨id⟩
	|	⟨kid⟩
	|	⟨id⟩ `()`
	|	⟨id⟩ `(` $\langle\text{pat}\rangle^{+}_{,}$ `)`
	|	⟨atomic_pat⟩ `:` ⟨typ⟩
	|	`(` ⟨pat⟩ `)`
	|	`(` ⟨pat⟩ `,` $\langle\text{pat}\rangle^{+}_{,}$ `)`
	|	`[` $\langle\text{pat}\rangle^{+}_{,}$ `]`
	|	`[| |]`
	|	`[|` $\langle\text{pat}\rangle^{+}_{,}$ `|]`
⟨lit⟩	::=	`true`
	|	`false`
	|	`()`
	|	Num
	|	`undefined`
	|	`bitzero`
	|	`bitone`
	|	Bin
	|	Hex
	|	String
	|	Real
⟨exp_eof⟩	::=	⟨exp⟩ Eof
⟨exp⟩	::=	⟨exp0⟩
	|	⟨atomic_exp⟩ `=` ⟨exp⟩
	|	`let` ⟨letbind⟩ `in` ⟨exp⟩
	|	`var` ⟨atomic_exp⟩ `=` ⟨exp⟩ `in` ⟨exp⟩
	|	`*` ⟨atomic_exp⟩
	|	`{` ⟨block⟩ `}`
	|	`return` ⟨exp⟩
	|	`throw` ⟨exp⟩
	|	`if` ⟨exp⟩ `then` ⟨exp⟩ `else` ⟨exp⟩
	|	`if` ⟨exp⟩ `then` ⟨exp⟩

```
           |  match ⟨exp⟩ { ⟨case⟩^+_, }
           |  try ⟨exp⟩ catch { ⟨case⟩^+_, }
           |  foreach ( ⟨id⟩ Id ⟨atomic_exp⟩ Id ⟨atomic_exp⟩ by ⟨atomic_exp⟩ in ⟨typ⟩
              ) ⟨exp⟩
           |  foreach ( ⟨id⟩ Id ⟨atomic_exp⟩ Id ⟨atomic_exp⟩ by ⟨atomic_exp⟩ ) ⟨exp⟩
           |  foreach ( ⟨id⟩ Id ⟨atomic_exp⟩ Id ⟨atomic_exp⟩ ) ⟨exp⟩
           |  repeat [termination_measure { ⟨exp⟩ }] ⟨exp⟩ until ⟨exp⟩
           |  while [termination_measure { ⟨exp⟩ }] ⟨exp⟩ do ⟨exp⟩
           |  InternalPLet ⟨pat⟩ = ⟨exp⟩ in ⟨exp⟩
           |  InternalReturn ⟨exp⟩

 ⟨exp0⟩  ::=  ⟨exp1⟩ ⟨op0⟩ ⟨exp1⟩
           |  ⟨exp0l⟩ ⟨op0l⟩ ⟨exp1⟩
           |  ⟨exp1⟩ ⟨op0r⟩ ⟨exp0r⟩
           |  ⟨exp1⟩

⟨exp0l⟩  ::=  ⟨exp1⟩ ⟨op0⟩ ⟨exp1⟩
           |  ⟨exp0l⟩ ⟨op0l⟩ ⟨exp1⟩
           |  ⟨exp1⟩

⟨exp0r⟩  ::=  ⟨exp1⟩ ⟨op0⟩ ⟨exp1⟩
           |  ⟨exp1⟩ ⟨op0r⟩ ⟨exp0r⟩
           |  ⟨exp1⟩

 ⟨exp1⟩  ::=  ⟨exp2⟩ ⟨op1⟩ ⟨exp2⟩
           |  ⟨exp1l⟩ ⟨op1l⟩ ⟨exp2⟩
           |  ⟨exp2⟩ ⟨op1r⟩ ⟨exp1r⟩
           |  ⟨exp2⟩

⟨exp1l⟩  ::=  ⟨exp2⟩ ⟨op1⟩ ⟨exp2⟩
           |  ⟨exp1l⟩ ⟨op1l⟩ ⟨exp2⟩
           |  ⟨exp2⟩

⟨exp1r⟩  ::=  ⟨exp2⟩ ⟨op1⟩ ⟨exp2⟩
           |  ⟨exp2⟩ ⟨op1r⟩ ⟨exp1r⟩
           |  ⟨exp2⟩

 ⟨exp2⟩  ::=  ⟨exp3⟩ ⟨op2⟩ ⟨exp3⟩
           |  ⟨exp2l⟩ ⟨op2l⟩ ⟨exp3⟩
           |  ⟨exp3⟩ ⟨op2r⟩ ⟨exp2r⟩
           |  ⟨exp3⟩ | ⟨exp2r⟩
           |  ⟨exp3⟩

⟨exp2l⟩  ::=  ⟨exp3⟩ ⟨op2⟩ ⟨exp3⟩
           |  ⟨exp2l⟩ ⟨op2l⟩ ⟨exp3⟩
           |  ⟨exp3⟩

⟨exp2r⟩  ::=  ⟨exp3⟩ ⟨op2⟩ ⟨exp3⟩
           |  ⟨exp3⟩ ⟨op2r⟩ ⟨exp2r⟩
           |  ⟨exp3⟩ | ⟨exp2r⟩
           |  ⟨exp3⟩

 ⟨exp3⟩  ::=  ⟨exp4⟩ ⟨op3⟩ ⟨exp4⟩
           |  ⟨exp3l⟩ ⟨op3l⟩ ⟨exp4⟩
           |  ⟨exp4⟩ ⟨op3r⟩ ⟨exp3r⟩
           |  ⟨exp4⟩ & ⟨exp3r⟩
```

```
          |    ⟨exp4⟩

⟨exp3l⟩  ::=   ⟨exp4⟩ ⟨op3⟩ ⟨exp4⟩
          |    ⟨exp3l⟩ ⟨op3l⟩ ⟨exp4⟩
          |    ⟨exp4⟩

⟨exp3r⟩  ::=   ⟨exp4⟩ ⟨op3⟩ ⟨exp4⟩
          |    ⟨exp4⟩ ⟨op3r⟩ ⟨exp3r⟩
          |    ⟨exp4⟩ & ⟨exp3r⟩
          |    ⟨exp4⟩

 ⟨exp4⟩  ::=   ⟨exp5⟩ ⟨op4⟩ ⟨exp5⟩
          |    ⟨exp5⟩ < ⟨exp5⟩
          |    ⟨exp5⟩ > ⟨exp5⟩
          |    ⟨exp5⟩ <= ⟨exp5⟩
          |    ⟨exp5⟩ >= ⟨exp5⟩
          |    ⟨exp5⟩ != ⟨exp5⟩
          |    ⟨exp5⟩ == ⟨exp5⟩
          |    ⟨exp4l⟩ ⟨op4l⟩ ⟨exp5⟩
          |    ⟨exp5⟩ ⟨op4r⟩ ⟨exp4r⟩
          |    ⟨exp5⟩

⟨exp4l⟩  ::=   ⟨exp5⟩ ⟨op4⟩ ⟨exp5⟩
          |    ⟨exp4l⟩ ⟨op4l⟩ ⟨exp5⟩
          |    ⟨exp5⟩

⟨exp4r⟩  ::=   ⟨exp5⟩ ⟨op4⟩ ⟨exp5⟩
          |    ⟨exp5⟩ ⟨op4r⟩ ⟨exp4r⟩
          |    ⟨exp5⟩

 ⟨exp5⟩  ::=   ⟨exp6⟩ ⟨op5⟩ ⟨exp6⟩
          |    ⟨exp5l⟩ ⟨op5l⟩ ⟨exp6⟩
          |    ⟨exp6⟩ ⟨op5r⟩ ⟨exp5r⟩
          |    ⟨exp6⟩ @ ⟨exp5r⟩
          |    ⟨exp6⟩ :: ⟨exp5r⟩
          |    ⟨exp6⟩

⟨exp5l⟩  ::=   ⟨exp6⟩ ⟨op5⟩ ⟨exp6⟩
          |    ⟨exp5l⟩ ⟨op5l⟩ ⟨exp6⟩
          |    ⟨exp6⟩

⟨exp5r⟩  ::=   ⟨exp6⟩ ⟨op5⟩ ⟨exp6⟩
          |    ⟨exp6⟩ ⟨op5r⟩ ⟨exp5r⟩
          |    ⟨exp6⟩ @ ⟨exp5r⟩
          |    ⟨exp6⟩ :: ⟨exp5r⟩
          |    ⟨exp6⟩

 ⟨exp6⟩  ::=   ⟨exp7⟩ ⟨op6⟩ ⟨exp7⟩
          |    ⟨exp6l⟩ ⟨op6l⟩ ⟨exp7⟩
          |    ⟨exp7⟩ ⟨op6r⟩ ⟨exp6r⟩
          |    ⟨exp6l⟩ + ⟨exp7⟩
          |    ⟨exp6l⟩ - ⟨exp7⟩
          |    ⟨exp7⟩

⟨exp6l⟩  ::=   ⟨exp7⟩ ⟨op6⟩ ⟨exp7⟩
          |    ⟨exp6l⟩ ⟨op6l⟩ ⟨exp7⟩
```

```
              |    ⟨exp6l⟩ + ⟨exp7⟩
              |    ⟨exp6l⟩ - ⟨exp7⟩
              |    ⟨exp7⟩

⟨exp6r⟩      ::=   ⟨exp7⟩ ⟨op6⟩ ⟨exp7⟩
              |    ⟨exp7⟩ ⟨op6r⟩ ⟨exp6r⟩
              |    ⟨exp7⟩

⟨exp7⟩       ::=   ⟨exp8⟩ ⟨op7⟩ ⟨exp8⟩
              |    ⟨exp7l⟩ ⟨op7l⟩ ⟨exp8⟩
              |    ⟨exp8⟩ ⟨op7r⟩ ⟨exp7r⟩
              |    ⟨exp7l⟩ * ⟨exp8⟩
              |    ⟨exp8⟩

⟨exp7l⟩      ::=   ⟨exp8⟩ ⟨op7⟩ ⟨exp8⟩
              |    ⟨exp7l⟩ ⟨op7l⟩ ⟨exp8⟩
              |    ⟨exp7l⟩ * ⟨exp8⟩
              |    ⟨exp8⟩

⟨exp7r⟩      ::=   ⟨exp8⟩ ⟨op7⟩ ⟨exp8⟩
              |    ⟨exp8⟩ ⟨op7r⟩ ⟨exp7r⟩
              |    ⟨exp8⟩

⟨exp8⟩       ::=   ⟨exp9⟩ ⟨op8⟩ ⟨exp9⟩
              |    ⟨exp8l⟩ ⟨op8l⟩ ⟨exp9⟩
              |    ⟨exp9⟩ ⟨op8r⟩ ⟨exp8r⟩
              |    ⟨exp9⟩ ^ ⟨exp8r⟩
              |    2 ^ ⟨exp9⟩
              |    ⟨exp9⟩

⟨exp8l⟩      ::=   ⟨exp9⟩ ⟨op8⟩ ⟨exp9⟩
              |    ⟨exp8l⟩ ⟨op8l⟩ ⟨exp9⟩
              |    2 ^ ⟨exp9⟩
              |    ⟨exp9⟩

⟨exp8r⟩      ::=   ⟨exp9⟩ ⟨op8⟩ ⟨exp9⟩
              |    ⟨exp9⟩ ⟨op8r⟩ ⟨exp8r⟩
              |    ⟨exp9⟩ ^ ⟨exp8r⟩
              |    2 ^ ⟨exp9⟩
              |    ⟨exp9⟩

⟨exp9⟩       ::=   ⟨atomic_exp⟩ ⟨op9⟩ ⟨atomic_exp⟩
              |    ⟨exp9l⟩ ⟨op9l⟩ ⟨atomic_exp⟩
              |    ⟨atomic_exp⟩ ⟨op9r⟩ ⟨exp9r⟩
              |    ⟨atomic_exp⟩

⟨exp9l⟩      ::=   ⟨atomic_exp⟩ ⟨op9⟩ ⟨atomic_exp⟩
              |    ⟨exp9l⟩ ⟨op9l⟩ ⟨atomic_exp⟩
              |    ⟨atomic_exp⟩

⟨exp9r⟩      ::=   ⟨atomic_exp⟩ ⟨op9⟩ ⟨atomic_exp⟩
              |    ⟨atomic_exp⟩ ⟨op9r⟩ ⟨exp9r⟩
              |    ⟨atomic_exp⟩

⟨case⟩       ::=   ⟨pat⟩ => ⟨exp⟩
              |    ⟨pat⟩ if ⟨exp⟩ => ⟨exp⟩
```

⟨block⟩	::=	⟨exp⟩
	|	`let` ⟨letbind⟩ `;` ⟨block⟩
	|	`var` ⟨atomic_exp⟩ `=` ⟨exp⟩ `;` ⟨block⟩
	|	⟨exp⟩ `;`
	|	⟨exp⟩ `;` ⟨block⟩
⟨letbind⟩	::=	⟨pat⟩ `=` ⟨exp⟩
⟨atomic_exp⟩	::=	⟨atomic_exp⟩ `:` ⟨atomic_typ⟩
	|	⟨lit⟩
	|	⟨id⟩ `->` ⟨id⟩ `()`
	|	⟨id⟩ `->` ⟨id⟩ `(` ⟨exp⟩$^{+}_{,}$ `)`
	|	⟨atomic_exp⟩ `.` ⟨id⟩ `()`
	|	⟨atomic_exp⟩ `.` ⟨id⟩ `(` ⟨exp⟩$^{+}_{,}$ `)`
	|	⟨atomic_exp⟩ `.` ⟨id⟩
	|	⟨id⟩
	|	⟨kid⟩
	|	`ref` ⟨id⟩
	|	⟨id⟩ `()`
	|	⟨id⟩ `(` ⟨exp⟩$^{+}_{,}$ `)`
	|	`exit ()`
	|	`exit (` ⟨exp⟩ `)`
	|	`sizeof (` ⟨typ⟩ `)`
	|	`constraint (` ⟨typ⟩ `)`
	|	`assert (` ⟨exp⟩ `)`
	|	`assert (` ⟨exp⟩ `,` ⟨exp⟩ `)`
	|	⟨atomic_exp⟩ `[` ⟨exp⟩ `]`
	|	⟨atomic_exp⟩ `[` ⟨exp⟩ `..` ⟨exp⟩ `]`
	|	⟨atomic_exp⟩ `[` ⟨exp⟩ `,` ⟨exp⟩ `]`
	|	`struct {` ⟨fexp_exp⟩$^{+}_{,}$ `}`
	|	`{` ⟨exp⟩ `with` ⟨fexp_exp⟩$^{+}_{,}$ `}`
	|	`[ ]`
	|	`[` ⟨exp⟩$^{+}_{,}$ `]`
	|	`[` ⟨exp⟩ `with` ⟨atomic_exp⟩ `=` ⟨exp⟩ `]`
	|	`[` ⟨exp⟩ `with` ⟨atomic_exp⟩ `..` ⟨atomic_exp⟩ `=` ⟨exp⟩ `]`
	|	`[| |]`
	|	`[|` ⟨exp⟩$^{+}_{,}$ `|]`
	|	`(` ⟨exp⟩ `)`
	|	`(` ⟨exp⟩ `,` ⟨exp⟩$^{+}_{,}$ `)`
⟨fexp_exp⟩	::=	⟨atomic_exp⟩ `=` ⟨exp⟩
	|	TildeTilde ⟨id⟩
⟨funcl_patexp⟩	::=	⟨pat⟩ `=` ⟨exp⟩
	|	`(` ⟨pat⟩ `if` ⟨exp⟩ `) =` ⟨exp⟩
⟨funcl_patexp_typ⟩	::=	⟨pat⟩ `=` ⟨exp⟩
	|	⟨pat⟩ `->` ⟨typ⟩ `=` ⟨exp⟩
	|	`forall` ⟨typquant⟩ `.` ⟨pat⟩ `->` ⟨typ⟩ `=` ⟨exp⟩
	|	`(` ⟨pat⟩ `if` ⟨exp⟩ `) =` ⟨exp⟩
	|	`forall` ⟨typquant⟩ `. (` ⟨pat⟩ `if` ⟨exp⟩ `) ->` ⟨typ⟩ `=` ⟨exp⟩
⟨funcl⟩	::=	⟨id⟩ ⟨funcl_patexp⟩

⟨funcl_doc⟩	::=	Doc ⟨funcl_doc⟩
	|	⟨funcl⟩
⟨funcls⟩	::=	⟨id⟩ ⟨funcl_patexp_typ⟩
	|	⟨id⟩ ⟨funcl_patexp⟩ `and` ⟨funcl_doc⟩$^{+}_{\texttt{and}}$
⟨funcl_typ⟩	::=	`forall` ⟨typquant⟩ `.` ⟨typ⟩
	|	⟨typ⟩
⟨index_range⟩	::=	⟨typ⟩
	|	⟨typ⟩ `..` ⟨typ⟩
⟨r_id_def⟩	::=	⟨id⟩ `:` ⟨index_range⟩
⟨r_def_body⟩	::=	⟨r_id_def⟩
	|	⟨r_id_def⟩ `,`
	|	⟨r_id_def⟩ `,` ⟨r_def_body⟩
⟨param_kopt⟩	::=	⟨kid⟩ `:` ⟨kind⟩
	|	⟨kid⟩
⟨typaram⟩	::=	`(` ⟨param_kopt⟩$^{+}_{,}$ `)` `,` ⟨typ⟩
	|	`(` ⟨param_kopt⟩$^{+}_{,}$ `)`
⟨type_def⟩	::=	Doc ⟨type_def⟩
	|	`type` ⟨id⟩ ⟨typaram⟩ `=` ⟨typ⟩
	|	`type` ⟨id⟩ `=` ⟨typ⟩
	|	`type` ⟨id⟩ ⟨typaram⟩ `->` ⟨kind⟩ `=` ⟨typ⟩
	|	`type` ⟨id⟩ `:` ⟨kind⟩ `=` ⟨typ⟩
	|	`struct` ⟨id⟩ `= {` ⟨struct_fields⟩ `}`
	|	`struct` ⟨id⟩ ⟨typaram⟩ `= {` ⟨struct_fields⟩ `}`
	|	`enum` ⟨id⟩ `=` ⟨id⟩$^{+}_{\mid}$
	|	`enum` ⟨id⟩ `= {` ⟨enum⟩ `}`
	|	`enum` ⟨id⟩ `with` (⟨id⟩ `->` ⟨typ⟩)$^{+}_{,}$ `= {` ⟨enum⟩ `}`
	|	`newtype` ⟨id⟩ `=` ⟨type_union⟩
	|	`newtype` ⟨id⟩ ⟨typaram⟩ `=` ⟨type_union⟩
	|	`union` ⟨id⟩ `= {` ⟨type_unions⟩ `}`
	|	`union` ⟨id⟩ ⟨typaram⟩ `= {` ⟨type_unions⟩ `}`
	|	`bitfield` ⟨id⟩ `:` ⟨typ⟩ `= {` ⟨r_def_body⟩ `}`
⟨enum⟩	::=	⟨id⟩
	|	⟨id⟩ `=>` ⟨exp⟩
	|	⟨id⟩ `,` ⟨enum⟩
	|	⟨id⟩ `=>` ⟨exp⟩ `,` ⟨enum⟩
⟨struct_field⟩	::=	⟨id⟩ `:` ⟨typ⟩
⟨struct_fields⟩	::=	⟨struct_field⟩
	|	⟨struct_field⟩ `,`
	|	⟨struct_field⟩ `,` ⟨struct_fields⟩
⟨type_union⟩	::=	Doc ⟨type_union⟩
	|	⟨id⟩ `:` ⟨typ⟩
	|	⟨id⟩ `:` ⟨typ⟩ `->` ⟨typ⟩
	|	⟨id⟩ `: {` ⟨struct_fields⟩ `}`

⟨type_unions⟩	::=	⟨type_union⟩
	|	⟨type_union⟩ `,`
	|	⟨type_union⟩ `,` ⟨type_unions⟩
⟨rec_measure⟩	::=	`{` ⟨pat⟩ `=>` ⟨exp⟩ `}`
⟨fun_def⟩	::=	Doc ⟨fun_def⟩
	|	`function` [⟨rec_measure⟩] ⟨funcls⟩
⟨mpat⟩	::=	⟨atomic_mpat⟩
	|	⟨atomic_mpat⟩ `as` ⟨id⟩
	|	⟨atomic_mpat⟩ `@` ⟨atomic_mpat⟩$^{+}_{@}$
	|	⟨atomic_mpat⟩ `::` ⟨mpat⟩
	|	⟨atomic_mpat⟩ `^` ⟨atomic_mpat⟩$^{+}_{\hat{}}$
⟨atomic_mpat⟩	::=	⟨lit⟩
	|	⟨id⟩
	|	⟨id⟩ `()`
	|	⟨id⟩ `(` ⟨mpat⟩$^{+}_{,}$ `)`
	|	`(` ⟨mpat⟩ `)`
	|	`(` ⟨mpat⟩ `,` ⟨mpat⟩$^{+}_{,}$ `)`
	|	`[` ⟨mpat⟩$^{+}_{,}$ `]`
	|	`[| |]`
	|	`[|` ⟨mpat⟩$^{+}_{,}$ `|]`
	|	⟨atomic_mpat⟩ `:` ⟨typ⟩
⟨mpexp⟩	::=	⟨mpat⟩
	|	⟨mpat⟩ `if` ⟨exp⟩
⟨mapcl⟩	::=	⟨mpexp⟩ `<->` ⟨mpexp⟩
	|	⟨mpexp⟩ `=>` ⟨exp⟩
	|	⟨mpexp⟩ `<-` ⟨exp⟩
⟨mapcl_doc⟩	::=	Doc ⟨mapcl_doc⟩
	|	⟨mapcl⟩
⟨map_def⟩	::=	Doc ⟨map_def⟩
	|	`mapping` ⟨id⟩ `=` `{` ⟨mapcl_doc⟩$^{+}_{,}$ `}`
	|	`mapping` ⟨id⟩ `:` ⟨typschm⟩ `=` `{` ⟨mapcl_doc⟩$^{+}_{,}$ `}`
⟨let_def⟩	::=	`let` ⟨letbind⟩
⟨externs⟩	::=	⟨id⟩ `:` String
	|	`_` `:` String
	|	⟨id⟩ `:` String `,` ⟨externs⟩
⟨val_spec_def⟩	::=	Doc ⟨val_spec_def⟩
	|	`val` ⟨id⟩ `:` ⟨typschm⟩
	|	`val` `cast` ⟨id⟩ `:` ⟨typschm⟩
	|	`val` ⟨id⟩ `=` String `:` ⟨typschm⟩
	|	`val` `cast` ⟨id⟩ `=` String `:` ⟨typschm⟩
	|	`val` String `:` ⟨typschm⟩
	|	`val` `cast` String `:` ⟨typschm⟩
	|	`val` ⟨id⟩ `=` `{` ⟨externs⟩ `}` `:` ⟨typschm⟩

| `val cast` ⟨id⟩ `= {` ⟨externs⟩ `} :` ⟨typschm⟩

⟨register_def⟩ ::= Doc ⟨register_def⟩
| `register` ⟨id⟩ `:` ⟨typ⟩
| `register` ⟨effect_set⟩ ⟨effect_set⟩ ⟨id⟩ `:` ⟨typ⟩
| `register configuration` ⟨id⟩ `:` ⟨typ⟩ `=` ⟨exp⟩

⟨default_def⟩ ::= `default` ⟨kind⟩ `inc`
| `default` ⟨kind⟩ `dec`

⟨scattered_def⟩ ::= Doc ⟨scattered_def⟩
| `scattered union` ⟨id⟩ ⟨typaram⟩
| `scattered union` ⟨id⟩
| `scattered function` ⟨id⟩
| `scattered mapping` ⟨id⟩
| `scattered mapping` ⟨id⟩ `:` ⟨funcl_typ⟩
| `function clause` ⟨funcl⟩
| `union clause` ⟨id⟩ `=` ⟨type_union⟩
| `mapping clause` ⟨id⟩ `=` ⟨mapcl⟩
| `end` ⟨id⟩

⟨loop_measure⟩ ::= `until` ⟨exp⟩
| `repeat` ⟨exp⟩
| `while` ⟨exp⟩

⟨def⟩ ::= ⟨fun_def⟩
| ⟨map_def⟩
| (`infix` | `infixl` | `infixr`) [0-9] operator
| ⟨val_spec_def⟩
| ⟨type_def⟩
| ⟨let_def⟩
| ⟨register_def⟩
| `overload` ⟨id⟩ `= {` $\langle\text{id}\rangle^{+}_{,}$ `}`
| `overload` ⟨id⟩ `=` $\langle\text{id}\rangle^{+}_{|}$
| ⟨scattered_def⟩
| ⟨default_def⟩
| `mutual {` $\langle\text{fun\_def}\rangle^{+}$ `}`
| Pragma
| `termination_measure` ⟨id⟩ ⟨pat⟩ `=` ⟨exp⟩
| `termination_measure` ⟨id⟩ $\langle\text{loop\_measure}\rangle^{+}_{,}$

⟨def_eof⟩ ::= ⟨def⟩ Eof

⟨file⟩ ::= $\langle\text{def}\rangle^{+}$ Eof
| Eof

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] 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.

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

[10] 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.

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

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

[13] 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.