Using Sail Specifications in Isabelle/HOL

May 11, 2018

Contents

1 Getting Started

This manual describes how to use Sail specifications for reasoning in Isabelle/HOL. For instructions on how to set up the Sail tool and its dependencies, see `INSTALL.md`. As an additional setup step for Isabelle generation, it is useful to build an Isabelle heap image of the Sail library. This will allow you to start Isabelle with the Sail library pre-loaded using the `-l Sail` option. For this purpose, run `make heap-img` in the `lib/isabelle` subdirectory of Sail and follow the instructions.

In order to generate theorem prover definitions, Sail specifications are first translated to Lem, which then generates definitions for Isabelle/HOL. Lem can also generate HOL4 definitions, though we have not yet tested that extensively for our ISA specifications. To produce Coq definitions, we envisage implementing a direct Sail-to-Coq backend, to preserve the Sail dependent types (it's possible that the Lem-to-Coq backend, which in general does not produce good Coq definitions, would actually produce usable Coq definitions for a monomorphised ISA specification, but we have not tested that).

The translation to Lem is activated by passing the `-lem` command line flag to Sail. For example, the following call in the `riscv` directory will generate Lem definitions for the RISC-V "duopod" (a fragment of the RISC-V specification with only two instructions, used for illustration purposes):

```
sail -lem -o riscv_duopod -lem_mwords -lem_lib Riscv_extras
  prelude.sail riscv_duopod.sail
```

This uses the following options:

- `-lem` activates the generation of Lem definitions.
- `-o riscv_duopod` specifies the prefix for the output filenames. This invocation of Sail will generate the files
  - `riscv_duopod_types.lem`, containing the definitions of the types used in the specification,
  - `riscv_duopod.lem`, containing the main definitions, e.g. of the instructions, and
  - `Riscv_duopod_lemmas.thy` containing generated helper lemmas, (currently) mainly simplification rules for lifting register reads and writes from the free monad to the state monad supported by Sail (cf. Section 3.3).
- `-lem_mwords` specifies that the generated definitions should use the machine word representation of bitvectors (cf. Section 3.2). This works out-of-the-box for the RISC-V specification, but might require monomorphisation (e.g. using the `-auto_mono` command line flag) for specifications that have functions that are polymorphic in bitvector lengths.
- `-lem_lib Riscv_extras` specifies an additional Lem library to be imported. It contains Lem implementations for some wrappers and primitive functions that are declared as external functions in the Sail source code, such as wrappers for reading and writing memory.

Isabelle definitions can then be generated by passing the `-isa` flag to Lem. In order for Lem to find the Sail library, the subdirectories `src/gen_lib` and `src/lem_interp` of Sail will have to be added to Lem's include path using the `-lib` option, e.g.

```
lem -isa -outdir . -lib ../src/lem_interp -lib ../src/gen_lib
  riscv_extras.lem riscv_duopod_types.lem riscv_duopod.lem
```

For further examples, see the `Makefile`s of the other specifications included in the Sail distribution.

2 An Example of a Sail Specification in Isabelle/HOL

A Sail specification typically comprises a *decode* function specifying a mapping from raw instruction opcodes to a more abstract representation, an *execute* function specifying the behaviour of instructions, further auxiliary functions and datatypes, and register declarations.

For example, in the RISC-V duopod, there are two instructions: a load instruction and an add instruction with one register and one immediate operand. Their abstract syntax is represented using the following datatype:

datatype $ast = ITYPE\ (12\ word \times 5\ word \times 5\ word \times iop)$
$\ \ |\ LOAD\ (12\ word \times 5\ word \times 5\ word)$

Both instructions take an immediate 12-bit argument (used as an offset in the case of the load instruction), and two 5-bit arguments encoding the source and the destination register, respectively. The *ITYPE* instruction takes another argument encoding the type of operation (where only addition is implemented in the "duopod" fragment of RISC-V).

The function $decode :: 32\ word \Rightarrow ast\ option$ is implemented in the Sail source code using bitvector pattern matching on the opcode. The Lem backend translates this to an if-then-else-cascade that compares the given opcode against one pattern after another:

$$
\begin{array}{l}
decode\ opcode = \\
(if\ subrange\text{-}vec\text{-}dec\ opcode\ 14\ 12 = vec\text{-}of\text{-}bits\ [B0,\ B0,\ B0] \wedge \\
\quad subrange\text{-}vec\text{-}dec\ opcode\ 6\ 0 = vec\text{-}of\text{-}bits\ [B0,\ B0,\ B1,\ B0,\ B0,\ B1,\ B1] \\
\ then\ let\ imm = subrange\text{-}vec\text{-}dec\ opcode\ 31\ 20; \\
\qquad\qquad rs1 = subrange\text{-}vec\text{-}dec\ opcode\ 19\ 15; \\
\qquad\qquad rd = subrange\text{-}vec\text{-}dec\ opcode\ 11\ 7 \\
\qquad in\ Some\ (ITYPE\ (imm,\ rs1,\ rd,\ RISCV\text{-}ADDI)) \\
\ else\ if\ subrange\text{-}vec\text{-}dec\ opcode\ 14\ 12 = vec\text{-}of\text{-}bits\ [B0,\ B1,\ B1] \wedge \\
\qquad\quad subrange\text{-}vec\text{-}dec\ opcode\ 6\ 0 = \\
\qquad\quad vec\text{-}of\text{-}bits\ [B0,\ B0,\ B0,\ B0,\ B0,\ B1,\ B1] \\
\qquad then\ let\ imm = subrange\text{-}vec\text{-}dec\ opcode\ 31\ 20; \\
\qquad\qquad\qquad rs1 = subrange\text{-}vec\text{-}dec\ opcode\ 19\ 15; \\
\qquad\qquad\qquad rd = subrange\text{-}vec\text{-}dec\ opcode\ 11\ 7 \\
\qquad\qquad in\ Some\ (LOAD\ (imm,\ rs1,\ rd)) \\
\qquad else\ None)
\end{array}
$$

This decode function is pure, although decoding might be effectful in other specifications (e.g., because the decoding depends on the register state). Sail uses its effect system to determine whether a function has side-effects and needs to be be monadic (cf. Section 3.3 for more details about the monads).

The *execute* function, for example, is monadic. Its clause for the load instruction of the RISC-V duopod is defined as follows, where $\gg=$ is infix syntax for the monadic bind:

$$
\begin{array}{l}
\textit{execute-LOAD imm rs rd} = \\
\textit{rX}\ (\textit{regbits-to-regno rs}) \gg= \\
(\lambda \textit{w--0}. \\
\quad \textit{let addr} = \textit{add-vec w--0}\ (\textit{EXTS 64 imm}) \\
\quad \textit{in MEMr addr 8} \gg= \textit{wX}\ (\textit{regbits-to-regno rd}))
\end{array}
$$

The instruction first reads the base address from the source register *rs*, then adds the offset given in the immediate argument *imm*, calls the *MEMr* auxiliary function to read eight bytes starting at the calculated address, and writes the result into the destination register *rd*.

Note that the *execute* function is special-cased in that Sail attempts to split it up into auxiliary functions (one per AST node) in order to avoid letting it become too large. The main *execute* function dispatches its inputs to the auxiliary functions:

$$
\begin{array}{l}
\textit{execute}\ (\textit{ITYPE}\ (\textit{imm1}, \textit{rs1}, \textit{rd1}, \textit{arg3.0})) = \textit{execute-ITYPE imm1 rs1 rd1 arg3.0} \\
\textit{execute}\ (\textit{LOAD}\ (\textit{imm}, \textit{rs}, \textit{rd})) = \textit{execute-LOAD imm rs rd}
\end{array}
$$

Apart from function and type definitions, Sail source code contains register declarations. A *regstate* record gets generated from these for use in the state monad, e.g.

$$
\begin{array}{l}
\textbf{record}\ \textit{regstate} = \\
\ \textit{Xs} :: (\textit{64 Word.word})\ \textit{list} \\
\ \textit{nextPC} :: \textit{64 Word.word} \\
\ \textit{PC} :: \textit{64 Word.word}
\end{array}
$$

In the RISC-V specification, the general-purpose register file is declared as one register *Xs* containing the 32 registers of 64 bits each, which gets mapped to a list of 64-bit words (see Section 3.1 for more information on vectors and lists in general). In addition to the register state record, a reference constant is generated for each register, e.g. *PC-ref*, which is used when the register is passed to Sail functions as an argument. These constants are records that contain the register name as a string, as well as getter and setter functions. We discuss them in more detail together with the monads in Section 3.3.

3 Sail Library

The overall theory graph of the Sail library is depicted in Figure 1. The library includes mappings of common operations on the basic types (Section 3.1), in particular bitvector operations for both the bitlist representation and the machine word representation of bitvectors (Section 3.2). It also includes theories defining the two monads currently supported: a state monad with exceptions and nondeterminism (cf. Section 3.3.1), and a free monad of an effects datatype (Section 3.3.2).

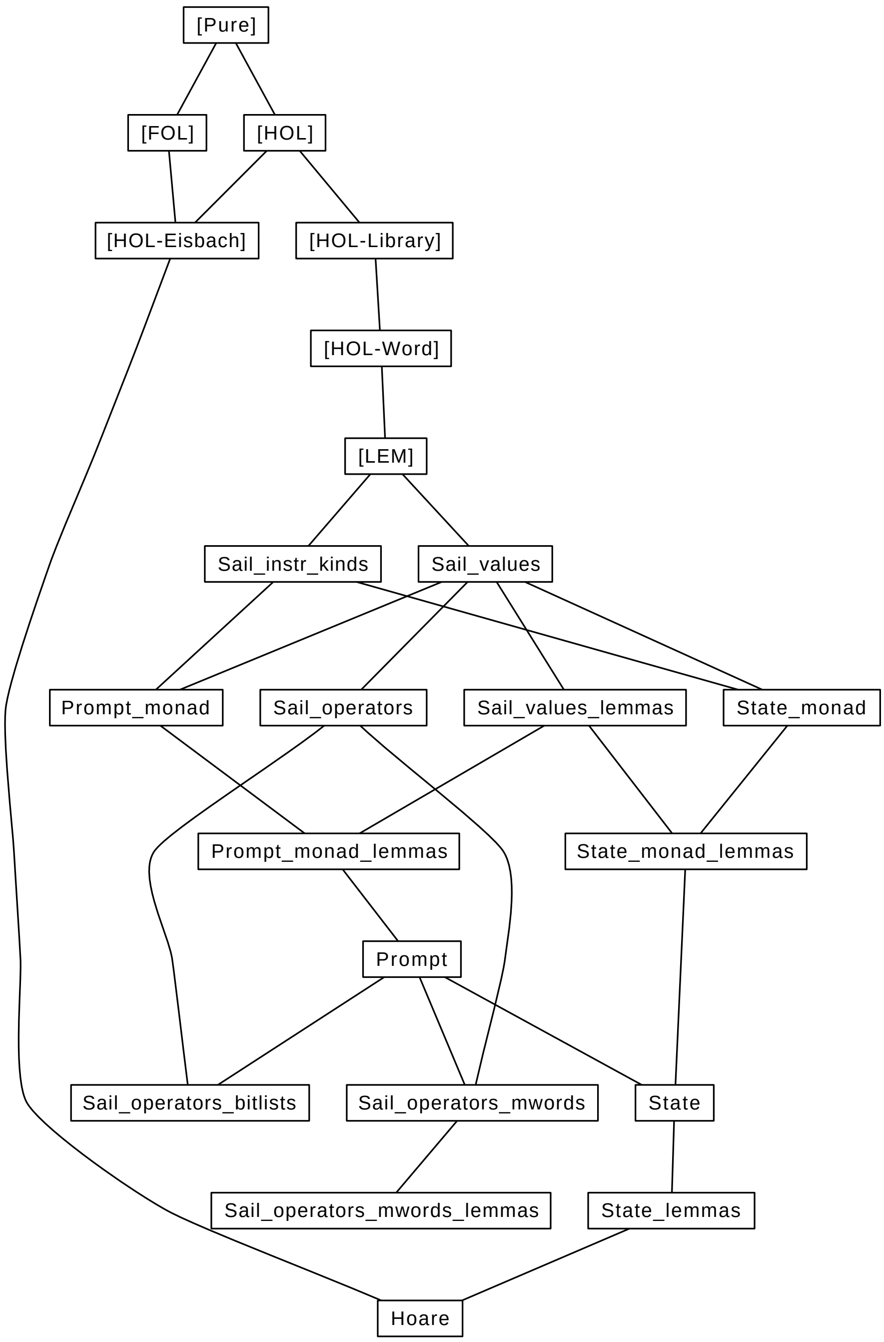

Figure 1: Session graph of the Sail library

The main definitions have been written in Lem and can therefore also be exported to theorem provers other than Isabelle. The Isabelle-specific parts of the library are contained in the theories named with the suffix `_lemmas`. They contain mostly simplification rules, but also congruence rules for the *bind* operations of the monads, for example, which are needed by the function package when processing recursive monadic functions.

3.1 Basic types

The basic Sail types `bool`, `string`, `list`, `unit` and `real` are directly mapped to the Isabelle types of the same name.

The numeric types `int`, `nat`, `atom`, and `range` are treated in Sail as integers with constraints. The latter are not currently translated to Lem or Isabelle, so these types are all mapped to the Isabelle type *int*.

Bits are represented by a type that can also represent undefined bits:

datatype *bitU* = *B0* | *B1* | *BU*

This provides one way to handle undefined cases of partial functions, such as division by zero. In general, the guiding principle in the Sail library is to make partiality of library functions explicit by returning an option type, and to provide wrappers implementing common ways to handle undefined cases. For example, the function *quot-vec* for bitvector division comes in the following variants:

- *quot-vec-maybe* returns an option type, with *quot-vec-maybe w 0* = *None*.
- *quot-vec-fail* is monadic and either returns the result or raises an exception.
- *quot-vec-oracle* is monadic and uses the *Undefined* effect in the exception case to fill the result with bits drawn from a bitstream oracle.
- *quot-vec* is pure and returns an arbitrary (but fixed) value in the exception case, currently defined as follows: For the bitlist representation of bitvectors, *quot-vec w 0* returns a list filled with *BU*, while for the machine word representation, the function gets mapped to Isabelle's division operation on machine words, which defines *w div 0* = *0*.

Which variant is to be used for a given specification can be chosen by using the corresponding binding for the Lem backend in the Sail source (typically in `prelude.sail`).

Vectors in Sail are mapped to lists in Isabelle, except for bitvectors, which are special-cased. Both increasing and decreasing indexing order are supported by having two versions for each operation that involves indexing, such as *update-list-inc* and *update-list-dec*, or *subrange-list-inc* and *subrange-list-dec*. These operations are defined in the theory *Sail-values*, while *Sail-values-lemmas* provides simplification rules such as

$access\text{-}list\text{-}inc\ xs\ i = xs\ !\ nat\ i$
$0 \leq i \Longrightarrow access\text{-}list\text{-}dec\ xs\ i = xs\ !\ (length\ xs - nat\ (i + 1))$

Note that, while Sail allows functions that are polymorphic in the indexing order, this kind of polymorphism is not currently supported by the translation to Lem. It is not needed by the currently existing specifications, however, since the indexing order is always fixed.

3.2 Bitvectors

The Lem backend of Sail supports two representations of bitvectors: bit lists and machine words. The former is less convenient for proofs, because it typically leads to many proof obligations about bitvector lengths. These are avoided with machine words, where length information is contained in the types, e.g. *64 word*. However, Isabelle/HOL does not support dependent types, which makes bitvector length polymorphism problematic. Sail includes an analysis and rewriting pass for monomorphising bitvector lengths, splitting up length-polymorphic functions into multiple clauses with concrete bitvector lengths. This is not enabled by default, however, so Sail generates Lem definitions using bit lists unless the `-lem_mwords` command line flag is used.

The theory *Sail-values* defines a (Lem) typeclass `Bitvector`, which provides an interface to some basic bitvector operations and has instantiations for both bit lists and machine words. It is mainly intended for internal use in the Sail library,[1] to implement library functions supporting either one of the bitvector representations. For use in Sail specifications, wrappers are defined in the theories `Sail_operators_bitlists` and `Sail_operators_mwords`, respectively. An import of the right theory is automatically added to the generated files, depending on which bitvector representation is used. Hence, bitvector operations can be referred to in the Sail source code using uniform names, e.g. *add-vec*, *update-vec-dec*, or *subrange-vec-inc*. The theory *Sail-operators-mwords-lemmas* sets up simplification rules that relate these operations to the native operations in Isabelle, e.g.

$add\text{-}vec\ l\ r = l + r$
$and\text{-}vec\ l\ r = l\ AND\ r$
$[\![0 \leq n;\ nat\ n < LENGTH('a)]\!] \Longrightarrow access\text{-}vec\text{-}dec\ w\ n = bitU\text{-}of\text{-}bool\ (w\ !!\ nat\ n)$

[1]Lem typeclasses are not very convenient to use in Isabelle, as they get translated to dictionaries that have to be passed to functions using the typeclass.

3.3 Monads

The definitions generated by Sail are designed to support reasoning in both concurrent and sequential settings. For the former, we use a free monad of an effect datatype that provides fine-grained information about the register and memory effects of monadic expressions, suitable for integration with relaxed memory models. For the sequential case, we use a state monad (with exceptions and nondeterminism).

The generated definitions use the free monad, and the sequential case is supported via a lifting to the state monad defined in the theory *State*. Simplification rules are set up in the theory *State-lemmas*, allowing seamless reasoning about the generated definitions in terms of the state monad.

3.3.1 State Monad

The state monad supports nondeterminism and exceptions and is defined in a standard way: a monadic expression maps a state to a set of results together with a corresponding successor state. The type (*'regs*, *'a*, *'e*) *monadS* is a synonym for

'regs sequential-state ⇒ ((*'a*, *'e*) *result* × *'regs sequential-state*) *set*

Here, *'a* and *'e* are parameters for the return value type and the exception type, respectively. The latter is instantiated in generated definitions with either the type *exception*, if the Sail source code defines that type, or with *unit* otherwise. A result of a monadic expression can be either a value, a non-recoverable failure, or an exception thrown (that may be caught using *try-catch*):

datatype *'e ex* = *Failure* (*char list*) | *Throw 'e*

datatype (*'a*, *'e*) *result* = *Value 'a* | *Ex* (*'e ex*)

The *sequential-state* record has the following fields:

- *regstate* contains the register state.
- *memstate* stores the memory, represented as a map from (*int*) addresses to (*bitU list*) bytes.
- Similarly, *tagstate* field stores a single bit per address, used by some specifications to model tagged memory.
- The *write-ea* field of type (*write-kind* × *int* × *int*) *option* stores the type, address, and size of the last announced memory write, if any.

- The *last-exclusive-operation-was-load* flag is used to determine whether exclusive operations can succeed.
- The function stored in the *next-bool* field together with the seed in the *seed* field are used as a random bit generator for undefined values. The *next-bool* function takes the current seed as an argument and returns a *bool* and the next seed.

The library defines several combinators and wrappers in addition to the standard monadic bind and return (called *bindS* and *returnS* here, where the suffix *S* differentiates them from the *bind* and *return* functions of the free monad). The functions *readS* and *updateS* provide direct access to the state, but there are more specific wrappers for common tasks such as

- *read-regS* and *write-regS* for accessing registers (taking a register reference as an argument),
- *read-memS* for reading memory,
- *write-mem-eaS* and *write-mem-valS* to announce and perform a memory write, respectively, and
- *undefined-boolS* gets a value from the random bit generator.

Nondeterminism can be introduced using *chooseS* to pick a value from a set, failure by *failS* or *exitS* (with or without failure message, respectively), assertions by *assert-expS* (causing a failure if the assertion fails), and exceptions by *throwS*. The latter can be caught using *try-catchS*, which takes a monadic expression and an exception handler as arguments.

The exception mechanism is also used to implement early returns by throwing and catching return values: A function body with one or more early returns of type $'a$ (and exception type $'e$) is lifted to a monadic expression with exception type $'a + 'e$ using *liftSR*, such that an early return of the value a throws *Inl a*, and a regular exception e is thrown as *Inr e*. The function body is then wrapped in *catch-early-returnS* to lower it back to the default monad and exception type. These liftings and lowerings are automatically inserted by Sail for functions with early returns.[2]

Finally, there are the loop combinators *foreachS*, *whileS*, and *untilS*. Loop bodies are required to be of type *unit* in the Sail source code, but during the translation to Lem they get rewritten into functions that take a tuple with the current values of local mutable variables that they might update as an (additional) argument, and return the updated values. Hence, the type of *foreachS*, for example, is

[2] To be precise, Sail's Lem backend uses the corresponding constructs for the free monad, but the state monad version presented here can be obtained using the monad transformation presented in the next section.

foreachS :: *'a list* ⇒ *'vars* ⇒ (*'a* ⇒ *'vars* ⇒ (*'regs*, *'vars*, *'e*) *monadS*) ⇒ (*'regs*, *'vars*, *'e*) *monadS*

Note that there is no general termination proof for *whileS* and *untilS*, so the termination predicates *whileS-dom* or *untilS-dom* have to be proved for concrete instances.

3.3.2 Free Monad

In addition to the state monad, the theory *Prompt-monad* defines a free monad of an effect datatype. A monadic expression either returns a pure value *a*, denoted *Done a*, or it has an effect. The latter can be a failure or an exception, or an effect together with a continuation. For example, *Read-reg ''PC'' k* represents a request to read the register *PC* and continue as *k*, which is a function that takes the register value as a parameter and returns another monadic expression. Another example is *Undefined k*, which requests a Boolean value from the execution context, e.g. to resolve an undefined bit to a concrete value. Again, the value is expected to be passed as an argument to the continuation *k*. The complete set of supported monadic outcomes is captured in the following datatype:

datatype (*'regval*, *'a*, *'e*) *monad* = *Done 'a*
  | *Read-mem read-kind* (*bitU list*) *nat*
    (*bitU list list* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Read-tag* (*bitU list*) (*bitU* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Write-ea write-kind* (*bitU list*) *nat* ((*'regval*, *'a*, *'e*) *monad*)
  | *Excl-res* (*bool* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Write-memv* (*bitU list list*) (*bool* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Write-tag* (*bitU list*) *bitU* (*bool* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Footprint* ((*'regval*, *'a*, *'e*) *monad*)
  | *Barrier barrier-kind* ((*'regval*, *'a*, *'e*) *monad*)
  | *Read-reg* (*char list*) (*'regval* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Write-reg* (*char list*) *'regval* ((*'regval*, *'a*, *'e*) *monad*)
  | *Undefined* (*bool* ⇒ (*'regval*, *'a*, *'e*) *monad*)
  | *Print* (*char list*) ((*'regval*, *'a*, *'e*) *monad*) | *Fail* (*char list*)
  | *Exception 'e*

The effects are designed to be usable as an interface to relaxed memory models. For example, *Footprint* tells the memory model that the register footprint of the instruction should be re-calculated. The Boolean parameters of the continuations of the *Write-memv*, *Write-tag*, and *Excl-res* effects allow the memory model to inform the instruction whether a memory write has succeeded (or may succeed).

The same set of combinators and wrappers as for the state monad is defined for this monad. The names are the same, but without the suffix *S*, e.g. *read-reg*, *write-mem-val*, *undefined-bool*, *throw*, *try-catch*, etc. (with the exception of the loop combinators, which are called *foreachM*, *whileM*, and *untilM*; the names *foreach*, *while*, and *until* are reserved for the pure versions of the loop combinators).

The monad is parametric in the register type used for the register effects. One technical complication is that, in general, this requires a single type that can subsume all the types of registers occurring in a specification. Otherwise, it would not be possible to find a single instantiation of the *monad* type to assign to a function that involves reading or writing multiple registers with different types, for example. To solve this problem, the translation from Sail to Lem generates a union type *register-value* with constructors for all register base types of the given specification and the built-in type constructors *vector*, *list*, and *option*. In the case of the RISC-V duopod, this is

datatype *register-value* = *Regval-vector* (*int* × *bool* × *register-value list*)
  | *Regval-list* (*register-value list*)
  | *Regval-option* (*register-value option*)
  | *Regval-vector-64-dec-bit* (*64 word*)

For example, a value of the (complete) *Xs* register file (whose Sail type is `vector(32, dec, vector(64, dec, bit))`) is represented as *Regval-vector* (*32*, *False*, *xs*), where *xs* is a list of words wrapped in *Regval-vector-64-dec-bit*.

Sail also generates conversion functions to and from *register-value*, e.g.

regval-of-vector-64-dec-bit :: *64 word* ⇒ *register-value*
vector-64-dec-bit-of-regval :: *register-value* ⇒ *64 word option*

where the latter is partial. The conversion functions for *Regval-vector*, *Regval-list*, and *Regval-option* are higher-order functions that take the corresponding conversion function for the encapsulated type as a parameter, e.g.

regval-of-vector :: (*'a* ⇒ *register-value*) ⇒ *int* ⇒ *bool* ⇒ *'a list* ⇒ *register-value*
vector-of-regval :: (*register-value* ⇒ *'a option*) ⇒ *register-value* ⇒ *'a list option*

The latter only returns a value if *all* elements of the vector can be successfully converted from *register-value* to *'a*.

For each register, the matching pair of conversion functions is recorded in its *register-ref* record, e.g.

PC-ref =
⦇*name* = *''PC''*, *read-from* = *PC*, *write-to* = *λv*. *PC-update* (*λ-*. *v*),
  of-regval = *vector-64-dec-bit-of-regval*,
  regval-of = *regval-of-vector-64-dec-bit*⦈

Xs-ref =
⦇*name* = *''Xs''*, *read-from* = *Xs*, *write-to* = *λv*. *Xs-update* (*λ-*. *v*),

$\quad$ *of-regval* $=$ *vector-of-regval vector-64-dec-bit-of-regval*,
$\quad$ *regval-of* $=$ *regval-of-vector regval-of-vector-64-dec-bit 32 False*$\|)$

The *read-reg* wrapper, for example, takes such a reference as a parameter, generates a *Read-reg* effect with the register name, and casts the register value received as input via *of-regval*. If the latter fails because the environment passed a value of the wrong type to the continuation, then *read-reg* halts with a *Failure*. The state monad wrappers *read-regS* and *write-regS* also take such a register reference as an argument, but use the getters and setters in the *read-from* and *write-to* fields to access the register state record:

$$\mathit{read\text{-}regS}\ \mathit{reg} = \mathit{readS}\ (\lambda s.\ \mathit{read\text{-}from}\ \mathit{reg}\ (\mathit{regstate}\ s))$$
$$\mathit{write\text{-}regS}\ \mathit{reg}\ v = \mathit{updateS}\ (\lambda s.\ s(\!|\mathit{regstate} := \mathit{write\text{-}to}\ \mathit{reg}\ v\ (\mathit{regstate}\ s)|\!))$$

Sail aims to generate Isabelle definitions that can be used with either the state or the free monad. To achieve this, the definitions are generated using the free monad, and a lifting to the state monad is provided together with simplification rules. These include generic simplification rules (proved in the theory *State-lemmas*) such as

$$\mathit{liftS}\ (\mathit{return}\ a) = \mathit{returnS}\ a$$
$$\mathit{liftS}\ (m \gg= f) = \mathit{bindS}\ (\mathit{liftS}\ m)\ (\mathit{liftS} \circ f)$$
$$\mathit{liftS}\ (\mathit{try\text{-}catch}\ m\ h) = \mathit{try\text{-}catchS}\ (\mathit{liftS}\ m)\ (\mathit{liftS} \circ h)$$

They also include more specific lemmas about register reads and writes: The lifting of these involves a back-and-forth conversion between the type of the register and the *register-value* type at the interface between the monads, which can fail in general. As long as the generated register references are used, however, it is guaranteed to succeed, and this is made explicit in lemmas such as

$$\mathit{liftS}\ (\mathit{read\text{-}reg}\ \mathit{PC\text{-}ref}) = \mathit{readS}\ (\mathit{PC} \circ \mathit{regstate})$$
$$\mathit{liftS}\ (\mathit{write\text{-}reg}\ \mathit{PC\text{-}ref}\ v) = \mathit{updateS}\ (\mathit{regstate\text{-}update}\ (\mathit{PC\text{-}update}\ (\lambda\text{-}.\ v)))$$

which are generated (together with their proofs) for each register and placed in a theory with the suffix `_lemmas`, e.g. `Riscv_duopod_lemmas`. The aim of these lemmas is to allow a smooth transition from the free to the state monad via simplification, as in the following example.

4 Example Proof

As a toy example for illustration, we prove that the add instruction in the RISC-V duopod actually performs an addition. We consider the sequential case and use the state monad. The theory *Hoare* defines (a shallow embedding of) a simple Hoare logic, where *PrePost P f Q* denotes a triple of a precondition *P*, monadic expression *f*, and postcondition *Q*. Its validity is defined by

$PrePost\ P\ f\ Q \equiv \forall s.\ P\ s \longrightarrow (\forall (r, s')\in f\ s.\ Q\ r\ s')$

There is also a quadruple variant, with separate postconditions for the regular and the exception case, defined as

$PrePostE\ P\ f\ Q\ E \equiv PrePost\ P\ f\ (\lambda v.\ case\ v\ of\ Value\ a \Rightarrow Q\ a \mid Ex\ e \Rightarrow E\ e)$

The theory includes standard proof rules for both of these variants, in particular rules giving weakest preconditions of the predefined primitives of the monad, collected under the names *PrePost-intro* and *PrePostE-intro*, respectively.

The instruction we are considering is defined as

execute-ITYPE imm rs1 rd RISCV-ADDI =
rX (*regbits-to-regno rs1*) ≫=
(λ*rs1-val*.
  let imm-ext = *EXTS 64 imm*
  in Let (*add-vec rs1-val imm-ext*) (*wX* (*regbits-to-regno rd*)))

We first declare two simplification rules and an abbreviation, for stating the lemma more conveniently: *getXs r s* reads general-purpose register *r* in state *s*, where register 0 is special-cased and hard-wired to the constant 0, as defined in the RISC-V specification.

abbreviation *getXs r s* ≡ *if r* = *0 then 0 else access-list-dec* (*Xs* (*regstate s*)) (*uint r*)

lemma *EXTS-scast*[*simp*]: *EXTS len w* = *scast w*
 by (*simp add*: *EXTS-def sign-extend-def*)

declare *regbits-to-regno-def*[*simp*]

We prove that a postcondition of the instruction is that the destination register holds the sum of the initial value of the source register and the immediate operand (unless the destination register is the constant zero register). Moreover, we require the instruction to succeed, so the postcondition for the exception case is *False*. In the precondition, we remember the initial value *v* of the source register for use in the postcondition (since it might get overwritten if *rs* = *rd*). We also explicitly assume that there are 32 general-purpose registers; due to the use of a list for the *Xs* register file, this information is currently not preserved by the translation.

lemma
 fixes *rs rd* :: *regbits* **and** *v* :: *64 word* **and** *imm* :: *12 word*

 defines *pre s* ≡ (*getXs rs s* = *v* ∧ *length* (*Xs* (*regstate s*)) = *32*)
 defines *instr* ≡ *execute* (*ITYPE* (*imm*, *rs*, *rd*, *RISCV-ADDI*))
 defines *post a s* ≡ (*rd* = *0* ∨ *getXs rd s* = *v* + (*scast imm*))

shows *PrePostE pre* (*liftS instr*) *post* (λ- -. *False*)

unfolding *pre-def instr-def post-def*
by (*simp add*: *rX-def wX-def cong*: *bindS-cong if-cong split del*: *if-split*)
(*rule PrePostE-strengthen-pre*, (*rule PrePostE-intro*)+, *auto simp*: *uint-0-iff*)

The proof begins with a simplification step, which not only unfolds the definitions of the auxiliary functions *rX* and *wX*, but also performs the lifting from the free monad to the state monad. We apply the rule *PrePostE-strengthen-pre* (in a backward manner) to allow a weaker precondition, then use the rules in *PrePostE-intro* to derive a weakest precondition, and then use *auto* to show that it is implied by the given precondition. For more serious proofs, one will want to set up specialised proof tactics. This example uses only basic proof methods, to make the reasoning steps more explicit.