Rust Verification with SAW

The SAW Development Team

Jan 28, 2025

Contents

1 Introduction

SAW is a special-purpose programming environment developed by Galois to help orchestrate and track the results of the large collection of proof tools necessary for analysis and verification of complex software artifacts.

SAW adopts the functional paradigm, and largely follows the structure of many other typed functional languages, with some special features specifically targeted at the coordination of verification and analysis tasks.

This tutorial introduces the details of the language by walking through several examples, and showing how simple verification tasks can be described. The complete examples are available on GitHub (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code). Most of the examples make use of inline specifications written in Cryptol, a language originally designed for high-level descriptions of cryptographic algorithms. For readers unfamiliar with Cryptol, various documents describing its use are available here (http://cryptol.net/documentation.html).

This tutorial also include a *case study* on how to use SAW to verify a real-world implementation of the Salsa20 stream cipher based on the `stream-ciphers` (https://github.com/RustCrypto/stream-ciphers) Rust project. The code for this case study is also available on GitHub (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20).

2 Prerequisites

In order to run the examples in this tutorial, you will need to install the following prerequisite tools:

- SAW itself, which can be installed by following the instructions here (https://github.com/GaloisInc/saw-script#manual-installation).
- The Z3 and Yices SMT solvers. Z3 can be downloaded from here (https://github.com/Z3Prover/z3/releases), and Yices can be downloaded from here (https://yices.csl.sri.com/).
- The `mir-json` tool, which can be installed by following the instructions here (https://github.com/GaloisInc/mir-json#installation-instructions).

3 About `mir-json`

We are interested in verifying code written in Rust, but Rust is an extremely rich programming language that has many distinct language features. To make the process of verifying Rust simpler, SAW targets an intermediate language used in the Rust compiler called MIR (https://blog.rust-lang.org/2016/04/19/MIR.html) (short for "Mid-level IR"). MIR takes the variety of different features and syntactic extensions in Rust and boils them down to a minimal subset that is easier for a computer program to analyze.

The process of extracting MIR code that is suitable for SAW's needs is somewhat tricky, so we wrote a suite of tools called `mir-json` to automate this process. `mir-json` provides a plugin for tools like `rustc` and `cargo` that lets you compile Rust code as you normally would and produce an additional `.json` file as output. This `.json` file contains the MIR code that was produced internally by the Rust compiler, with some additional minor tweaks to make it suitable for SAW's needs.

`mir-json` is not a single tool but rather a suite of related tools that leverage the same underlying plugin. For SAW purposes, the two `mir-json` tools that are most relevant are:

- `saw-rustc`: A thin wrapper around `rustc` (the Rust compiler), which is suitable for individual `.rs` files.
- `cargo-saw-build`: A thin wrapper around the `cargo build` command, which is suitable for `cargo`-based Rust projects.

Most of the examples in this tutorial involve self-contained examples, which will use `saw-rustc`. Later in the tutorial, we will examine a Salsa20 case study that involves a `cargo`-based project, which will use `cargo-saw-build`.

3.1 A first example with `saw-rustc`

Let's try out `saw-rustc` on a small example file, which we'll name `first-example.rs`:

```
pub fn id_u8(x: u8) -> u8 {
    x
}
```

This is the identity function, but specialized to the type `u8`. We can compile this with `saw-rustc` like so:

```
$ saw-rustc first-example.rs
<snip>
note: Emitting MIR for first_example/abef32c5::id_u8

linking 1 mir files into first-example.linked-mir.json
<snip>
```

`saw-rustc` prints out some additional information that `rustc` alone does not print, and we have displayed the parts of this information that are most interesting. In particular:

- `saw-rustc` notes that is is `Emitting MIR for first_example/abef32c5::id_u8`, where `first_example/abef32c5::id_u8` is the full *identifier* used to uniquely refer to the `id_u8` function. It's entirely possible that the `abef32c5` bit will look different on your machine; we'll talk more about identifiers in the "Identifiers" section.
- Once `saw-rustc` produced a MIR JSON file named `first-example.linked-mir.json`. This is an important bit of information, as SAW will ingest this JSON file.

If you'd like, you can inspect the `first-example.linked-mir.json` file with JSON tools (e.g., `jq` (https://jqlang.github.io/jq/)), but it is not important to understand everything that is going on there. This is machine-generated JSON, and as such, it is not meant to be particularly readable by human eyes.

3.2 The `SAW_RUST_LIBRARY_PATH` environment variable

Rust has a large set of standard libraries that ship with the compiler, and parts of the standard library are quite low-level and tricky. SAW's primary objective is to provide a tool that can analyze code in a tractable way. For this reason, SAW sometimes needs to invoke simplified versions of Rust standard library functions that are more reasonable for an SMT-based tool like SAW to handle. These simplified functions are equivalent in behavior, but avoid using problematic code patterns (e.g., gnarly pointer arithmetic or the `transmute` (https://doc.rust-lang.org/std/intrinsics/fn.transmute.html) function).

If you are only ever compiling self-contained pieces of code with `saw-rustc`, there is a good chance that you can get away without needing to use SAW's custom version of the Rust standard libraries. However, if you ever need to build something more complicated than that (e.g., the Salsa20 case

study later in this tutorial), then you *will* need the custom libraries. For this reason, it is worthwhile to teach SAW the location of the custom libraries now.

At present, the best way to obtain the custom version of the Rust standard libraries is to perform the following steps:

1. Clone the `crucible` (https://github.com/GaloisInc/crucible) repo like so:

   ```
   $ git clone https://github.com/GaloisInc/crucible
   ```

2. Navigate to the `crux-mir` (https://github.com/GaloisInc/crucible/tree/master/crux-mir) subdirectory of the `crucible` checkout:

   ```
   $ cd crucible/crux-mir/
   ```

3. Run the `translate_libs.sh` script:

   ```
   $ ./translate_libs.sh
   ```

   This will compile the custom versions of the Rust standard libraries using `mir-json`, placing the results under the `rlibs` subdirectory.

4. Finally, define a `SAW_RUST_LIBRARY_PATH` environment variable that points to the newly created `rlibs` subdirectory:

   ```
   $ export SAW_RUST_LIBRARY_PATH=<...>/crucible/crux-mir/rlibs
   ```

An upcoming release of SAW will include these custom libraries pre-built, which will greatly simplify the steps above. Either way, you will need to set the `SAW_RUST_LIBRARY_PATH` environment variable to point to the location of the custom libraries.

3.3 A note about generics

The `id_u8` function above is likely not how most Rust programmers would define the identity function. Instead, it would seem more natural to define it generically, that is, by parameterizing the function by a type parameter:

```
pub fn id<A>(x: A) -> A {
    x
}
```

If you compile this with `saw-rustc`, however, the resulting JSON file will lack a definition for `id`! We can see this by using `jq`:

```
$ saw-rustc generics-take-1.rs
<snip>
$ jq . generics-take-1.linked-mir.json
```

(continues on next page)

(continued from previous page)

```
{
  "fns": [],
  "adts": [],
  "statics": [],
  "vtables": [],
  "traits": [],
  "intrinsics": [],
  "tys": [],
  "roots": []
}
```

What is going on here? This is the result of an important design choice that SAW makes: *SAW only supports verifying monomorphic functions*. To be more precise, SAW's approach to symbolic simulation requires all of the code being simulated to have fixed types without any type parameters.

In order to verify a function using generics in your Rust code, you must provide a separate, monomorphic function that calls into the generic function. For example, you can rewrite the example above like so:

```
pub fn id<A>(x: A) -> A {
    x
}

pub fn id_u8(x: u8) -> u8 {
    id(x)
}
```

If you compile this version with `saw-rustc`, you'll see:

```
$ saw-rustc generics-take-2.rs
<snip>
note: Emitting MIR for generics_take_2/8b1bf337::id_u8

note: Emitting MIR for generics_take_2/8b1bf337::id::_
↪instaddce72e1232152c[0]

linking 1 mir files into generics-take-2.linked-mir.json
<snip>
```

This time, the resulting JSON file contains a definition for `id_u8`. The reason that this works is because when `id_u8` calls `id`, the Rust compile will generate a specialized version of `id` where `A` is instantiated with the type `u8`. This specialized version of `id` is named `id::_instaddce72e1232152c[0]` in the output above. (You don't have to remember this name, thankfully!)

Although the resulting JSON file contains a definition for `id_u8`, it does *not* contain a definition for the generic `id` function. As a result, SAW will only be able to verify the `id_u8` function from this file. If you are ever in doubt about which functions are accessible for verification with SAW, you can check this with `jq` like so:

```
$ jq '.intrinsics | map(.name)' generics-take-2.linked-mir.json
[
  "generics_take_2/8b1bf337::id_u8",
  "generics_take_2/8b1bf337::id::_instaddce72e1232152c[0]"
]
```

Here, "intrinsics" are monomorphic functions that are visible to SAW. Note that `saw-rustc` will optimize away all definitions that are not accessible from one of these intrinsic functions. This explains why the original program that only defined a generic `id` function resulted in a definition-less JSON file, as that program did not contain monomorphic functions (and therefore no intrinsics).

Generally speaking, we prefer to verify functions that are defined directly in the Rust source code, such as `id_u8`, as these functions' names are more stable than the specialized versions of functions that the compiler generates, such as `id::_instaddce72e1232152c[0]`. Do note that SAW is capable of verifying both types of functions, however. (We will see an example of verifying an autogenerated function in the Salsa20 case study later in this tutorial.)

3.4 Identifiers

When you compile a function named `id_u8`, `saw-rustc` will expand it to a much longer name such as `first_example/abef32c5::id_u8`. This longer name is called an *identifier*, and it provides a globally unique name for that function. In the small examples we've seen up to this point, there hasn't been any risk of name collisions, but you could imagine compiling this code alongside another file (or crate) that also defines an `id_u8` function. If that happens, then it is essential that we can tell apart all of the different `id_u8` functions, and identifiers provide us the mechanism for doing so.

Let's take a closer look at what goes into an identifier. In general, an identifier will look like the following:

```
<crate name>/<disambiguator>::<function path>
```

`<crate name>` is the name of the crate in which the function is defined. All of the examples we've seen up to this point have been defined in standalone files, and as a result, the crate name has been the same as the file name, but without the `.rs` extension and with all hyphens replaced with underscores (e.g., `first-example.rs` is given the crate name `first_example`). In `cargo`-based projects, the crate name will likely differ from the file name.

`<disambiguator>` is a hash of the crate and its dependencies. In extreme cases, it is possible for two different crates to have identical crate names, in which case the disambiguator must be used to distinguish between the two crates. In the common case, however, most crate names will correspond to exactly one disambiguator. (More on this in a bit.)

`<function path>` is the path to the function within the crate. Sometimes, this is as simple as the function name itself. In other cases, a function path may involve multiple *segments*, depending on the module hierarchy for the program being verified. For instance, a `read` function located in `core/src/ptr/mod.rs` will have the identifier:

```
core/<disambiguator>::ptr::read
```

Where `core` is the crate name and `ptr::read` is the function path, which has two segments `ptr` and `read`. There are also some special forms of segments that appear for functions defined in certain language constructs. For instance, if a function is defined in an `impl` block, then it will have `{impl}` as one of its segments, e.g.,

```
core/<disambiguator>::ptr::const_ptr::{impl}::offset
```

The most cumbersome part of writing an identifier is the disambiguator, as it is extremely sensitive to changes in the code (not to mention hard to remember and type). Luckily, the vast majority of crate names correspond to exactly one disambiguator, and we can exploit this fact to abbreviate identifiers that live in such crates. For instance, we can abbreviate this identifier:

```
core/<disambiguator>::ptr::read
```

To simply:

```
core::ptr::read
```

We will adopt the latter, shorter notation throughout the rest of the tutorial. SAW also understands this shorthand, so we will also use this notation when passing identifiers to SAW commands.

4 SAW basics

4.1 A first SAW example

We now have the knowledge necessary to compile Rust code in a way that is suitable for SAW. Let's put our skills to the test and verify something! We will build on the example from above, which we will put into a file named `saw-basics.rs`:

```
pub fn id<A>(x: A) -> A {
    x
}

pub fn id_u8(x: u8) -> u8 {
    id(x)
}
```

Our goal is to verify the correctness of the `id_u8` function. However, it is meaningless to talk about whether a function is correct without having a *specification* for how the function should behave. This is

where SAW enters the picture. SAW provides a scripting language named *SAWScript* that allows you to write a precise specification for describing a function's behavior. For example, here is a specification that captures the intended behavior of `id_u8`:

```
let id_u8_spec = do {
  x <- mir_fresh_var "x" mir_u8;
  mir_execute_func [mir_term x];
  mir_return (mir_term x);
};
```

At a high level, this specification says that `id_u8` is a function that accepts a single argument of type `u8`, and it returns its argument unchanged. Nothing too surprising there, but this example illustrates many of the concepts that one must use when working with SAW. Let's unpack what this is doing, line by line:

- In SAWScript, specifications are ordinary values that are defined with `let`. In this example, we are defining a specification named `id_u8_spec`.
- Specifications are defined using "`do`-notation". That is, they are assembled by writing `do { <stmt>; <stmt>; ...; <stmt>; }`, where each `<stmt>` is a statement that declares some property about the function being verified. A statement can optionally bind a variable that can be passed to later statements, which is accomplished by writing `<var> <- <stmt>`.
- The `x <- mir_fresh_var "x" mir_u8;` line declares that `x` is a fresh variable of type `u8` (represented by `mir_u8` in SAWScript) that has some unspecified value. In SAW parlance, we refer to these unspecified values as *symbolic* values. SAW uses an SMT solver under the hood to reason about symbolic values.

  The `"x"` string indicates what name the variable `x` should have when sent to the underlying SMT solver. This is primarily meant as a debugging aid, and it is not required that the string match the name of the SAWScript variable. (For instance, you could just as well have passed `"x_smt"` or something else.)
- The `mir_execute_func [mir_term x];` line declares that the function should be invoked with `x` as the argument. For technical reasons, we pass `mir_term x` to `mir_execute_func` rather than just `x`; we will go over what `mir_term` does later in the tutorial.
- Finally, the `mir_return (mir_term x);` line declares that the function should return `x` once it has finished.

Now that we have a specification in hand, it's time to prove that `id_u8` actually adheres to the spec. To do so, we need to load the MIR JSON version of `id_u8` into SAW, which is done with the `mir_load_module` command:

```
m <- mir_load_module "saw-basics.linked-mir.json";
```

This `m` variable contains the definition of `id_u8`, as well as the other code defined in the program. We can then pass `m` to the `mir_verify` command, which actually verifies that `id_u8` behaves according to `id_u8_spec`:

```
mir_verify m "saw_basics::id_u8" [] false id_u8_spec z3;
```

Here is what is going on in this command:

- The `m` and `"saw_basics::id_u8"` arguments instruct SAW to verify the `id_u8` function located in the `saw_basics` crate defined in `m`. Note that we are using the shorthand identifier notation here, so we are allowed to omit the disambiguator for the `saw_basics` crate.
- The `[]` argument indicates that we will not provide any function overrides to use when SAW simulates the `id_u8` function. (We will go over how overrides work later in the tutorial.)
- The `false` argument indicates that SAW should not use path satisfiability checking when analyzing the function. Path satisfiability checking is an advanced SAW feature that we will not be making use of in this tutorial, so we will always use `false` here.
- The `id_u8_spec` argument indicates that `id_u8` should be checked against the specification defined by `id_u8_spec`.
- The `z3` argument indicates that SAW should use the Z3 SMT solver to solve any proof goals that are generated during verification. SAW also supports other SMT solvers, although we will mostly use Z3 in this tutorial.

Putting this all together, our complete `saw-basics.saw` file is:

```
enable_experimental;

let id_u8_spec = do {
  x <- mir_fresh_var "x" mir_u8;
  mir_execute_func [mir_term x];
  mir_return (mir_term x);
};

m <- mir_load_module "saw-basics.linked-mir.json";

mir_verify m "saw_basics::id_u8" [] false id_u8_spec z3;
```

One minor detail that we left out until just now is that the SAW's interface to MIR is still experimental, so you must explicitly opt into it with the `enable_experimental` command.

Now that everything is in place, we can check this proof like so:

```
$ saw saw-basics.saw



[16:14:07.006] Loading file "saw-basics.saw"
[16:14:07.009] Verifying saw_basics/f77ebf43::id_u8[0] ...
```

(continues on next page)

(continued from previous page)

```
[16:14:07.017] Simulating saw_basics/f77ebf43::id_u8[0] ...
[16:14:07.017] Checking proof obligations saw_basics/f77ebf43::id_u8[0]␣
↪...
[16:14:07.017] Proof succeeded! saw_basics/f77ebf43::id_u8[0]
```

Tada! SAW was successfully able to prove that `id_u8` adheres to its spec.

4.2 Cryptol

The spec in the previous section is nice and simple. It's also not very interesting, as it's fairly obvious at a glance that `id_u8`'s implementation is correct. Most of the time, we want to verify more complicated functions where the correspondence between the specification and the implementation is not always so clear.

For example, consider this function, which multiplies a number by two:

```
pub fn times_two(x: u32) -> u32 {
    x << 1 // Gotta go fast
}
```

The straightforward way to implement this function would be to return `2 * x`, but the author of this function *really* cared about performance. As such, the author applied a micro-optimization that computes the multiplication with a single left-shift (`<<`). This is the sort of scenario where we are pretty sure that the optimized version of the code is equivalent to the original version, but it would be nice for SAW to check this.

Let's write a specification for the `times_two` function:

```
let times_two_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_execute_func [mir_term x];
  mir_return (mir_term {{ 2 * x }});
};
```

This spec introduces code delimited by double curly braces `{{ ... }}`, which is a piece of syntax that we haven't seen before. The code in between the curly braces is written in Cryptol (http://cryptol.net/documentation.html), a language designed for writing high-level specifications of various algorithms. Cryptol supports most arithmetic operations, so `2 * x` works exactly as you would expect. Also note that the `x` variable was originally bound in the SAWScript language, but it is possible to embed `x` into the Cryptol language by referencing `x` within the curly braces. (We'll say more about how this embedding works later.)

`{{ 2 * x }}` takes the Cryptol expression `2 * x` and lifts it to a SAW expression. As such, this SAW spec declares that the function takes a single `u32`-typed argument `x` and returns `2 * x`. We could have also wrote the specification to declare that the function returns `x << 1`, but that would

have defeated the point of this exercise: we specifically want to check that the function against a spec that is as simple and readable as possible.

Our full SAW file is:

```
enable_experimental;

let times_two_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_execute_func [mir_term x];
  mir_return (mir_term {{ 2 * x }});
};

m <- mir_load_module "times-two.linked-mir.json";

mir_verify m "times_two::times_two" [] false times_two_spec z3;
```

Which we can verify is correct like so:

```
$ saw times-two.saw



[17:51:35.469] Loading file "times-two.saw"
[17:51:35.497] Verifying times_two/6f4e41af::times_two[0] ...
[17:51:35.512] Simulating times_two/6f4e41af::times_two[0] ...
[17:51:35.513] Checking proof obligations times_two/6f4e41af::times_
↪two[0] ...
[17:51:35.527] Proof succeeded! times_two/6f4e41af::times_two[0]
```

Nice! Even though the `times_two` function does not literally return `2 * x`, SAW is able to confirm that the function behaves as if it were implemented that way.

4.3 `Term`s and other types

Now that we know how Cryptol can be used within SAW, we can go back and explain what the `mir_term` function does. It is helpful to examine the type of `mir_term` by using SAW's interactive mode. To do so, run the `saw` binary without any other arguments:

```
$ saw
```

Then run `enable_experimental` (to enable MIR-related commands) and run `:type mir_term`:

```
sawscript> enable_experimental
sawscript> :type mir_term
Term -> MIRValue
```

Here, we see that `mir_term` accepts a `Term` as an argument and returns a `MIRValue`. In this context, the `Term` type represents a Cryptol value, and the `MIRValue` type represents SAW-related MIR values. `Term`s can be thought of as a subset of `MIRValue`s, and the `mir_term` function is used to promote a `Term` to a `MIRValue`.

Most other MIR-related commands work over `MIRValue`s, as can be seen with SAW's `:type` command:

```
sawscript> :type mir_execute_func
[MIRValue] -> MIRSetup ()
sawscript> :type mir_return
MIRValue -> MIRSetup ()
```

Note that `MIRSetup` is the type of statements in a MIR specification, and two `MIRSetup`-typed commands can be chained together by using `do`-notation. Writing `MIRSetup ()` means that the statement does not return anything interesting, and the use of `()` here is very much analogous to how `()` is used in Rust. There are other `MIRSetup`-typed commands that *do* return something interesting, as is the case with `mir_fresh_var`:

```
sawscript> :type mir_fresh_var
String -> MIRType -> MIRSetup Term
```

This command returns a `MIRSetup Term`, which means that when you write `x <- mir_fresh_var ... ...` in a MIR specification, then `x` will be bound at type `Term`.

Values of type `Term` have the property that they can be embedded into Cryptol expression that are enclosed in double curly braces `{{ ... }}`. This is why our earlier `{{ 2 * x }}` example works, as `x` is of type `Term`.

4.4 Preconditions and postconditions

As a sanity check, let's write a naïve version of `times_two` that explicitly returns `2 * x`:

```
pub fn times_two_ref(x: u32) -> u32 {
    2 * x
}
```

It seems like we should be able to verify this `times_two_ref` function using the same spec that we used for `times_two`:

```
mir_verify m "times_two::times_two_ref" [] false times_two_spec z3;
```

Somewhat surprisingly, SAW fails to verify this function:

```
$ saw times-two-ref-fail.saw
```

(continues on next page)

(continued from previous page)

```
[18:58:22.578] Loading file "times-two-ref-fail.saw"
[18:58:22.608] Verifying times_two/56182919::times_two_ref[0] ...
[18:58:22.621] Simulating times_two/56182919::times_two_ref[0] ...
[18:58:22.622] Checking proof obligations times_two/56182919::times_two_
↪ref[0] ...
[18:58:22.640] Subgoal failed: times_two/56182919::times_two_ref[0]␣
↪attempt to compute `const 2_u32 * move _2`, which would overflow
[18:58:22.640] SolverStats {solverStatsSolvers = fromList ["SBV->Z3"],␣
↪solverStatsGoalSize = 375}
[18:58:22.640] ----------Counterexample----------
[18:58:22.640]   x: 2147483648
[18:58:22.640] Stack trace:
"mir_verify" (times-two-ref-fail.saw:11:1-11:11)
Proof failed.
```

The "`which would overflow`" portion of the error message suggests what went wrong. When a Rust program is compiled with debug settings (which is the default for `rustc` and `saw-rustc`), arithmetic operations such as multiplication will check if the result of the operation can fit in the requested number of bits. If not, the program will raise an error.

In this case, we must make the result of multiplication fit in a `u32`, which can represent values in the range `0` to `2^^32 - 1` (where `^^` is Cryptol's exponentiation operator). But it is possible to take a number in this range, multiply it by two, and have the result fall outside of the range. In fact, SAW gives us a counterexample with exactly this number: `2147483648`, which can also be written as `2^^31`. Multiplying this by two yields `2^^32`, which is just outside of the range of values expressible with `u32`. SAW's duties include checking that a function cannot fail at runtime, so this function falls afoul of that check.

Note that we didn't have this problem with the original definition of `times_two` because the semantics of `<<` are such that if the result is too large to fit in the requested type, then the result will *overflow*, i.e., wrap back around to zero and count up. This means that `(2^^31) << 1` evaluates to `0` rather than raising an error. Cryptol's multiplication operation also performs integer overflow (unlike Rust in debug settings), which is why we didn't notice any overflow-related issues when verifying `times_two`.

There are two possible ways that we can repair this. One way is to rewrite `times_two_ref` to use Rust's `wrapping_mul` (https://doc.rust-lang.org/std/primitive.u32.html#method.wrapping_mul) function, a variant of multiplication that always uses integer overflow. This work around the issue, but it is a bit more verbose.

The other way is to make our spec more precise such that we only verify `times_two_ref` for particular inputs. Although `times_two_ref` will run into overflow-related issues when the argument is `2^^31` or greater, it is perfectly fine for inputs smaller than `2^^31`. We can encode such an

assumption in SAW by adding a *precondition*. To do so, we write a slightly modified version of `times_two_spec`:

```
let times_two_ref_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_precond {{ x < 2^^31 }};
  mir_execute_func [mir_term x];
  mir_return (mir_term {{ 2 * x }});
};
```

The most notable change is the `mir_precond {{ x < 2^^31 }};` line. `mir_precond` (where "`precond`" is short for "precondition") is a command that takes a `Term` argument that contains a boolean predicate, such as `{{ x < 2^^31 }}`. Declaring a precondition requires that this predicate must hold during verification, and any values of `x` that do not satisfy this predicate are not considered.

By doing this, we have limited the range of the function from `0` to `2^^31 - 1`, which is exactly the range of values for which `times_two_ref` is well defined. SAW will confirm this if we run it:

```
mir_verify m "times_two::times_two_ref" [] false times_two_ref_spec z3;
```

```
[19:23:53.480] Verifying times_two/56182919::times_two_ref[0] ...
[19:23:53.496] Simulating times_two/56182919::times_two_ref[0] ...
[19:23:53.497] Checking proof obligations times_two/56182919::times_two_
↪ref[0] ...
[19:23:53.531] Proof succeeded! times_two/56182919::times_two_ref[0]
```

We can add as many preconditions to a spec as we see fit. For instance, if we only want to verify `times_two_ref` for positive integers, we could add an additional assumption:

```
let times_two_ref_positive_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_precond {{ 0 < x }}; // The input must be positive
  mir_precond {{ x < 2^^31 }};
  mir_execute_func [mir_term x];
  mir_return (mir_term {{ 2 * x }});
};
```

In addition to preconditions, SAW also supports postconditions. Whereas preconditions represent conditions that must hold *before* invoking a function, postconditions represent conditions that must hold *after* invoking a function. We have already seen one type of postcondition in the form of the `mir_return` command, which imposes a postcondition on what the return value must be equal to.

We can introduce additional postconditions with the `mir_postcond` command. For example, if we call `times_two_ref` with a positive argument, then it should be the case that the return value should be strictly greater than the argument value. We can check for this using `mir_postcond` like so:

```
let times_two_ref_positive_postcond_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_precond {{ 0 < x }}; // The input must be positive
  mir_precond {{ x < 2^^31 }};
  mir_execute_func [mir_term x];
  mir_postcond {{ x < (2 * x) }}; // Argument value < return value
  mir_return (mir_term {{ 2 * x }});
};
```

An additional convenience that SAW offers is the `mir_assert` command. `mir_assert` has the same type as `mir_precond` and `mir_postcond`, but `mir_assert` can be used to declare both preconditions *and* postconditions. The difference is where `mir_assert` appears in a specification. If `mir_assert` is used before the call to `mir_execute_func`, then it declares a precondition. If `mir_assert` is used after the call to `mir_execute_func`, then it declares a postcondition.

For example, we can rewrite `times_two_ref_positive_postcond_spec` to use `mir_assert`s like so:

```
let times_two_ref_positive_postcond_assert_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_assert {{ 0 < x }}; // The input must be positive
  mir_assert {{ x < 2^^31 }};
  mir_execute_func [mir_term x];
  mir_assert {{ x < (2 * x) }}; // Argument value < return value
  mir_return (mir_term {{ 2 * x }});
};
```

The choice of whether to use `mir_precond`/`mir_postcond` versus `mir_assert` is mostly a matter personal taste.

5 Reference types

All of the examples we have seen up to this point involve simple integer types such as `u8` and `u32`. While these are useful, Rust's type system features much more than just integers. A key part of Rust's type system are its reference types. For example, in this `read_ref` function:

```
pub fn read_ref(r: &u32) -> u32 {
    *r
}
```

The function reads the value that `r` (of type `&u32`) points to and returns it. Writing SAW specifications involving references is somewhat trickier than with other types of values because we must also specify what memory the reference points to. SAW provides a special command for doing this called `mir_alloc`:

```
sawscript> :type mir_alloc
MIRType -> MIRSetup MIRValue
```

`mir_alloc` will allocate a reference value with enough space to hold a value of the given `MIRType`. Unlike `mir_fresh_var`, `mir_alloc` returns a `MIRValue` instead of a `Term`. We mentioned before that `Term`s are only a subset of `MIRValue`s, and this is one of the reasons why. Cryptol does not have a notion of reference values, but `MIRValue`s do. As a result, you cannot embed the result of a call to `mir_alloc` in a Cryptol expression.

`mir_alloc` must be used with some care. Here is a first, not-quite-correct attempt at writing a spec for `read_ref` using `mir_alloc`:

```
let read_ref_fail_spec = do {
  r <- mir_alloc mir_u32;
  mir_execute_func [r];
  // mir_return ??;
};

m <- mir_load_module "ref-basics.linked-mir.json";

mir_verify m "ref_basics::read_ref" [] false read_ref_fail_spec z3;
```

As the comment suggests, it's not entirely clear what this spec should return. We can't return `r`, since `read_ref` returns something of type `u32`, not `&u32`. On the other hand, we don't have any values of type `u32` lying around that are obviously the right thing to use here. Nevertheless, it's not required for a SAW spec to include a `mir_return` statement, so let's see what happens if we verify this as-is:

```
$ saw ref-basics-fail.saw



[20:13:27.224] Loading file "ref-basics-fail.saw"
[20:13:27.227] Verifying ref_basics/54ae7b63::read_ref[0] ...
[20:13:27.235] Simulating ref_basics/54ae7b63::read_ref[0] ...
[20:13:27.235] Stack trace:
"mir_verify" (ref-basics-fail.saw:11:1-11:11)
Symbolic execution failed.
Abort due to assertion failure:
  ref-basics.rs:2:5: 2:7: error: in ref_basics/54ae7b63::read_ref[0]
  attempted to read empty mux tree
```

Clearly, SAW didn't like what we gave it. The reason this happens is although we allocated memory for the reference `r`, we never told SAW what value should live in that memory. When SAW simulated the `read_ref` function, it attempted to dereference `r`, which pointed to uninitialized memory. This is constitutes an error in SAW, which is what this "`attempted to read empty mux tree`"

business is about.

SAW provides a `mir_points_to` command to declare what value a reference should point to:

```
sawscript> :type mir_points_to
MIRValue -> MIRValue -> MIRSetup ()
```

Here, the first `MIRValue` argument represents a reference value, and the second `MIRValue` argument represents the value that the reference should point to. In our spec for `read_ref`, we can declare that the reference should point to a symbolic `u32` value like so:

```
let read_ref_spec = do {
  r_ref <- mir_alloc mir_u32;
  r_val <- mir_fresh_var "r_val" mir_u32;
  mir_points_to r_ref (mir_term r_val);
  mir_execute_func [r_ref];
  mir_return (mir_term r_val);
};
```

We have renamed `r` to `r_ref` in this revised spec to more easily distinguish it from `r_val`, which is the value that `r_ref` is declared to point to using `mir_points_to`. In this version of the spec, it is clear that we should return `r_val` using `mir_return`, as `r_val` is exactly the value that will be computed by dereferencing `r_ref`.

This pattern, where a call to `mir_alloc`/`mir_alloc_mut` to followed by a call to `mir_points_to`, is common with function specs that involve references. Later in the tutorial, we will see other examples of `mir_points_to` where the reference argument does not come from `mir_alloc`/`mir_alloc_mut`.

The argument to `read_ref` is an immutable reference, so the implementation of the function is not allowed to modify the memory that the argument points to. Rust also features mutable references that do permit modifying the underlying memory, as seen in this `swap` function:

```
pub fn swap(a: &mut u32, b: &mut u32) {
    let a_tmp: u32 = *a;
    let b_tmp: u32 = *b;

    *a = b_tmp;
    *b = a_tmp;
}
```

A corresponding spec for `swap` is:

```
let swap_spec = do {
  a_ref <- mir_alloc_mut mir_u32;
  a_val <- mir_fresh_var "a_val" mir_u32;
```

(continues on next page)

(continued from previous page)

```
  mir_points_to a_ref (mir_term a_val);

  b_ref <- mir_alloc_mut mir_u32;
  b_val <- mir_fresh_var "b_val" mir_u32;
  mir_points_to b_ref (mir_term b_val);

  mir_execute_func [a_ref, b_ref];

  mir_points_to a_ref (mir_term b_val);
  mir_points_to b_ref (mir_term a_val);
};
```

There are two interesting things worth calling out in this spec:

1. Instead of allocating the reference values with `mir_alloc`, we instead use `mir_alloc_mut`. This is a consequence of the fact that `&mut u32` is a different type from `&mut` in Rust (and in MIR), and and such, we need a separate `mir_alloc_mut` to get the types right.

2. This spec features calls to `mir_points_to` before *and* after `mir_execute_func`. This is because the values that `a_ref` and `b_ref` point to before calling the function are different than the values that they point to after calling the function. The two uses to `mir_points_to` after the function has been called swap the order of `a_val` and `b_val`, reflecting the fact that the `swap` function itself swaps the values that the references point to.

6 Compound data types

Besides integer types and reference types, Rust also features a variety of other interesting data types. This part of the tutorial will briefly go over some of these data types and how to interface with them in SAW.

6.1 Array types

Rust includes array types where the length of the array is known ahead of time. For instance, this `index` function takes an `arr` argument that must contain exactly three `u32` values:

```
pub fn index(arr: [u32; 3], idx: usize) -> u32 {
    arr[idx]
}
```

While Rust is good at catching many classes of programmer errors at compile time, one thing that it cannot catch in general is out-of-bounds array accesses. In this `index` example, calling the function with a value of `idx` ranging from `0` to `2` is fine, but any other choice of `idx` will cause the program to crash, since the `idx` will be out of the bounds of `arr`.

SAW is suited to checking for these sorts of out-of-bound accesses. Let's write an incorrect spec for `index` to illustrate this:

```
let index_fail_spec = do {
  arr <- mir_fresh_var "arr" (mir_array 3 mir_u32);
  idx <- mir_fresh_var "idx" mir_usize;

  mir_execute_func [mir_term arr, mir_term idx];

  mir_return (mir_term {{ arr @ idx }});
};

m <- mir_load_module "arrays.linked-mir.json";

mir_verify m "arrays::index" [] false index_fail_spec z3;
```

Before we run this with SAW, let's highlight some of the new concepts that this spec uses:

1. The type of the `arr` variable is specified using `mir_array 3 mir_u32`. Here, the `mir_array` function takes the length of the array and the element type as arguments, just as in Rust.
2. The spec declares the return value to be `{{ arr @ idx }}`, where `@` is Cryptol's indexing operator. Also note that it is completely valid to embed a MIR array type into a Cryptol expression, as Cryptol has a sequence type that acts much like arrays do in MIR.

As we hinted above, this spec is wrong, as it says that this should work for *any* possible values of `idx`. SAW will catch this mistake:

```
$ saw arrays-fail.saw



[21:03:05.374] Loading file "arrays-fail.saw"
[21:03:05.411] Verifying arrays/47a26581::index[0] ...
[21:03:05.425] Simulating arrays/47a26581::index[0] ...
[21:03:05.426] Checking proof obligations arrays/47a26581::index[0] ...
[21:03:05.445] Subgoal failed: arrays/47a26581::index[0] index out of␣
↪bounds: the length is move _4 but the index is _3
[21:03:05.445] SolverStats {solverStatsSolvers = fromList ["SBV->Z3"],␣
↪solverStatsGoalSize = 53}
[21:03:05.445] ----------Counterexample----------
[21:03:05.445]   idx: 2147483648
[21:03:05.445] Stack trace:
"mir_verify" (arrays-fail.saw:14:1-14:11)
Proof failed.
```

We can repair this spec by adding some preconditions:

```
let index_spec = do {
  arr <- mir_fresh_var "arr" (mir_array 3 mir_u32);
  idx <- mir_fresh_var "idx" mir_usize;
  mir_precond {{ 0 <= idx }}; // Lower bound of idx
  mir_precond {{ idx <= 2 }}; // Upper bound of idx

  mir_execute_func [mir_term arr, mir_term idx];

  mir_return (mir_term {{ arr @ idx }});
};
```

An alternative way of writing this spec is by using SAW's `mir_array_value` command:

```
sawscript> :type mir_array_value
MIRType -> [MIRValue] -> MIRValue
```

Here, the `MIRType` argument represents the element type, and the list of `MIRValue` arguments are the element values of the array. We can rewrite `index_spec` using `mir_array_value` like so:

```
let index_alt_spec = do {
  arr0 <- mir_fresh_var "arr0" mir_u32;
  arr1 <- mir_fresh_var "arr1" mir_u32;
  arr2 <- mir_fresh_var "arr2" mir_u32;
  let arr = mir_array_value mir_u32 [mir_term arr0, mir_term arr1, mir_
↪term arr2];

  idx <- mir_fresh_var "idx" mir_usize;
  mir_precond {{ 0 <= idx }}; // Lower bound of idx
  mir_precond {{ idx <= 2 }}; // Upper bound of idx

  mir_execute_func [arr, mir_term idx];

  mir_return (mir_term {{ [arr0, arr1, arr2] @ idx }});
```

Here, `[arr0, arr1, arr2]` is Cryptol notation for constructing a length-3 sequence consisting of `arr0`, `arr1`, and `arr2` as the elements. `index_alt_spec` is equivalent to `index_spec`, albeit more verbose. For this reason, it is usually preferable to use `mir_fresh_var` to create an entire symbolic array rather than creating separate symbolic values for each element and combining them with `mir_array_value`.

There are some situations where `mir_array_value` is the only viable choice, however. Consider this variant of the `index` function:

```
pub fn index_ref_arr(arr: [&u32; 3], idx: usize) -> u32 {
    *arr[idx]
}
```

When writing a SAW spec for `index_ref_arr`, we can't just create a symbolic variable for `arr` using `mir_alloc (mir_array 3 ...)`, as the reference values in the array wouldn't point to valid memory. Instead, we must individually allocate the elements of `arr` using separate calls to `mir_alloc` and then build up the array using `mir_array_value`. (As an exercise, try writing and verifying a spec for `index_ref_arr`).

6.2 Tuple types

Rust includes tuple types where the elements of the tuple can be of different types. For example:

```
pub fn flip(x: (u32, u64)) -> (u64, u32) {
    (x.1, x.0)
}
```

SAW includes a `mir_tuple` function for specifying the type of a tuple value. In addition, one can embed MIR tuples into Cryptol, as Cryptol also includes tuple types whose fields can be indexed with `.0`, `.1`, etc. Here is a spec for `flip` that makes use of all these features:

```
let flip_spec = do {
  x <- mir_fresh_var "x" (mir_tuple [mir_u32, mir_u64]);

  mir_execute_func [mir_term x];

  mir_return (mir_term {{ (x.1, x.0) }});
};
```

SAW also includes a `mir_tuple_value` function for constructing a tuple value from other `MIRValue`s:

```
sawscript> :type mir_tuple_value
[MIRValue] -> MIRValue
```

`mir_tuple_value` plays a similar role for tuples as `mir_array_value` does for arrays.

6.3 Struct types

Rust supports the ability for users to define custom struct types. Structs are uniquely identified by their names, so if you have two structs like these:

```
pub struct S(u32, u64);
pub struct T(u32, u64);
```

Then even though the fields of the `S` and `T` structs are the same, they are *not* the same struct. This is a type system feature that Cryptol does not have, and for this reason, it is not possible to embed MIR struct values into Cryptol. It is also not possible to use `mir_fresh_var` to create a symbolic struct value. Instead, one can use the `mir_struct_value` command:

```
sawscript> :type mir_struct_value
MIRAdt -> [MIRValue] -> MIRValue
```

Like with `mir_array_value` and `mir_tuple_value`, the `mir_struct_value` function takes a list of `MIRValue`s as arguments. What makes `mir_struct_value` unique is its `MIRAdt` argument, which we have not seen up to this point. In this context, "`Adt`" is shorthand for "algebraic data type (https://en.wikipedia.org/wiki/Algebraic_data_type)", and Rust's structs are an example of ADTs. (Rust also supports enums, another type of ADT that we will see later in this tutorial.)

ADTs in Rust are named entities, and as such, they have unique identifiers in the MIR JSON file in which they are defined. Looking up these identifiers can be somewhat error-prone, so SAW offers a `mir_find_adt` command that computes an ADT's identifier and returns the `MIRAdt` associated with it:

```
sawscript> :type mir_find_adt
MIRModule -> String -> [MIRType] -> MIRAdt
```

Here, `MIRModule` correspond to the MIR JSON file containing the ADT definition, and the `String` is the name of the ADT whose identifier we want to look up. The list of `MIRType`s represent types to instantiate any type parameters to the struct (more on this in a bit).

As an example, we can look up the `S` and `T` structs from above like so:

```
m <- mir_load_module "structs.linked-mir.json";

let s_adt = mir_find_adt m "structs::S" [];
let t_adt = mir_find_adt m "structs::T" [];
```

We pass an empty list of `MIRType`s to each use of `mir_find_adt`, as neither `S` nor `T` have any type parameters. An example of a struct that does include type parameters can be seen here:

```
pub struct Foo<A, B>(A, B);
```

As mentioned before, SAW doesn't support generic definitions out of the box, so the only way that we can make use of the `Foo` struct is by looking up a particular instantiation of `Foo`'s type parameters. If we define a function like this, for example:

```
pub fn make_foo() -> Foo<u32, u64> {
    Foo(27, 42)
}
```

Then this function instantiates `Foo`'s `A` type parameter with `u32` and the `B` type parameter with `u64`. We can use `mir_find_adt` to look up this particular instantiation of `Foo` like so:

```
let foo_adt = mir_find_adt m "structs::Foo" [mir_u32, mir_u64];
```

In general, a MIR JSON file can have many separate instantiations of a single struct's type parameters, and each instantiation must be looked up separately using `mir_find_adt`.

Having looked up `Foo<u32, u64>` using `mir_find_adt`, let's use the resulting `MIRAdt` in a spec:

```
let make_foo_spec = do {
  mir_execute_func [];

  let ret = mir_struct_value
              foo_adt
              [mir_term {{ 27 : [32] }}, mir_term {{ 42 : [64] }}];
  mir_return ret;
};

mir_verify m "structs::make_foo" [] false make_foo_spec z3;
```

Note that we are directly writing out the values `27` and `42` in Cryptol. Cryptol's numeric literals can take on many different types, so in order to disambiguate which type they should be, we give each numeric literal an explicit type annotation. For instance, the expression `27 : [32]` means that `27` should be a 32-bit integer.

Symbolic structs

Let's now verify a function that takes a struct value as an argument:

```
pub struct Bar(u8, u16, Foo<u32, u64>);

pub fn do_stuff_with_bar(b: Bar) {
    // ...
}
```

Moreover, let's verify this function for all possible `Bar` values. One way to do this is to write a SAW spec that constructs a struct value whose fields are themselves symbolic:

```
let bar_adt = mir_find_adt m "structs::Bar" [];

let do_stuff_with_bar_spec1 = do {
  z1 <- mir_fresh_var "z1" mir_u32;
  z2 <- mir_fresh_var "z2" mir_u64;
  let z = mir_struct_value
            foo_adt
```

(continues on next page)

(continued from previous page)

```
            [mir_term z1, mir_term z2];

  x <- mir_fresh_var "x" mir_u8;
  y <- mir_fresh_var "y" mir_u16;
  let b = mir_struct_value
            bar_adt
            [mir_term x, mir_term y, z];

  mir_execute_func [b];

  // ...
};
```

This is a rather tedious process, however, as we had to repeatedly use `mir_fresh_var` to create a fresh, symbolic value for each field. Moreover, because `mit_fresh_var` does not work for structs, we had to recursively apply this process in order to create a fresh `Foo` value. It works, but it takes a lot of typing to accomplish.

To make this process less tedious, SAW offers a `mir_fresh_expanded_value` command that allows one to create symbolic values of many more types. While `mir_fresh_var` is limited to those MIR types that can be directly converted to Cryptol, `mir_fresh_expanded_value` can create symbolic structs by automating the process of creating fresh values for each field. This process also applies recursively for struct fields, such as the `Foo` field in `Bar`.

As an example, a much shorter way to write the spec above using `mir_fresh_expanded_value` is:

```
let do_stuff_with_bar_spec2 = do {
  b <- mir_fresh_expanded_value "b" (mir_adt bar_adt);

  mir_execute_func [b];

  // ...
};
```

That's it! Note that the string `"b"` is used as a prefix for all fresh names that `mir_fresh_expanded_value` generates, so if SAW produces a counterexample involving this symbolic struct value, one can expect to see names such as `b_0`, `b_1`, etc. for the fields of the struct.

`mir_fresh_expanded_value` makes it easier to construct large, compound values that consist of many smaller, inner values. The drawback is that you can't refer to these inner values in the postconditions of a spec. As a result, there are some functions for which `mir_fresh_expanded_value` isn't suitable, so keep this in mind before reaching for it.

6.4 Enum types

Besides structs, another form of ADT that Rust supports are enums. Each enum has a number of different *variants* that describe the different ways that an enum value can look like. A famous example of a Rust enum is the `Option` type, which is defined by the standard library like so:

```
enum Option<T> {
    None,
    Some(T),
}
```

`Option` is commonly used in Rust code to represent a value that may be present (`Some`) or absent (`None`). For this reason, we will use `Option` as our motivating example of an enum in this section.

First, let's start by defining some functions that make use of `Option`'s variants:

```
pub fn i_found_something(x: u32) -> Option<u32> {
    Some(x)
}

pub fn i_got_nothing() -> Option<u32> {
    None
}
```

Both functions return an `Option<u32>` value, but each function returns a different variant. In order to tell these variants apart, we need a SAW function which can construct an enum value that allows the user to pick which variant they want to construct. The `mir_enum_value function does exactly that:

```
sawscript> :type mir_enum_value
MIRAdt -> String -> [MIRValue] -> MIRValue
```

Like `mir_struct_value`, `mir_enum_value` also requires a `MIRAdt` argument in order to discern which particular enum you want. Unlike `mir_struct_value`, however, it also requires a `String` which variant of the enum you want. In the case of `Option`, this `String` will either be `"None"` or `"Some"`. Finally, the `[MIRValue]` arguments represent the fields of the enum variant.

Let's now verify some enum-related code with SAW. First, we must look up the `Option<u32>` ADT, which works just as if you had a struct type:

```
let option_u32 = mir_find_adt m "core::option::Option" [mir_u32];
```

Next, we can use this ADT to construct enum values. We shall use `mir_enum_value` to create a `Some` value in the spec for `i_found_something`:

```
let i_found_something_spec = do {
  x <- mir_fresh_var "x" mir_u32;
```

(continues on next page)

(continued from previous page)

```
  mir_execute_func [mir_term x];

  let ret = mir_enum_value option_u32 "Some" [mir_term x];
  mir_return ret;
};

mir_verify m "enums::i_found_something" [] false i_found_something_spec␣
↪z3;
```

Note that while we used the full identifier `core::option::Option` to look up the `Option` ADT, we do not need to use the `core::option` prefix when specifying the `"Some"` variant. This is because SAW already knows what the prefix should be from the `option_u32` ADT, so the `"Some"` shorthand suffices.

Similarly, we can also write a spec for `i_got_nothing`, which uses the `None` variant:

```
let i_got_nothing_spec = do {
  mir_execute_func [];

  let ret = mir_enum_value option_u32 "None" [];
  mir_return ret;
};

mir_verify m "enums::i_got_nothing" [] false i_got_nothing_spec z3;
```

Symbolic enums

In order to create a symbolic struct, one could create symbolic fields and pack them into a larger struct value using `mir_struct_value`. The same process is not possible with `mir_enum_value`, however, as a symbolic enum value would need to range over *all* possible variants in an enum.

Just as `mir_fresh_expanded_value` supports creating symbolic structs, `mir_fresh_expanded_value` also supports creating symbolic enum values. For example, given this function that accepts an `Option<u32>` value as an argument:

```
pub fn do_stuff_with_option(o: Option<u32>) {
    // ...
}
```

We can write a spec for this function that considers all possible `Option<u32>` values like so:

```
let do_stuff_with_option_spec = do {
  o <- mir_fresh_expanded_value "o" (mir_adt option_u32);
```

(continues on next page)

(continued from previous page)

```
  mir_execute_func [o];

  // ...
};
```

Here, `o` can be a `None` value, or it can be a `Some` value with a symbolic field.

6.5 Slices

Slices are a particular type of reference that allow referencing contiguous sequences of elements in a collection, such as an array. Unlike ordinary references (e.g., `&u32`), SAW does not permit allocating a slice directly. Instead, one must take a slice of an existing reference. To better illustrate this distinction, consider this function:

```
pub fn sum_of_prefix(s: &[u32]) -> u32 {
    s[0].wrapping_add(s[1])
}
```

`sum_of_prefix` takes a slice to a sequence of `u32`s as an argument, indexes into the first two elements in the sequence, and adds them together. There are many possible ways we can write a spec for this function, as the slice argument may be backed by many different sequences. For example, the slice might be backed by an array whose length is exactly two:

```
    let a1 = [1, 2];
    let s1 = &a1[..];
    let sum1 = sum_of_prefix(s1);
```

We could also make a slice whose length is longer than two:

```
    let a2 = [1, 2, 3, 4, 5];
    let s2 = &a2[..];
    let sum2 = sum_of_prefix(s2);
```

Alternatively, the slice might be a subset of an array whose length is longer than two:

```
    let a3 = [1, 2, 3, 4, 5];
    let s3 = &a3[0..2];
    let sum3 = sum_of_prefix(s3);
```

All of these are valid ways of building the slice argument to `sum_of_prefix`. Let's try to write SAW specifications that construct these different forms of slices. To do so, we will need SAW functions that take a reference to a collection (e.g., an array) and converts them into a slice reference. The `mir_slice_value` function is one such function:

```
sawscript> :type mir_slice_value
MIRValue -> MIRValue
```

`mir_slice_value arr_ref` is the SAW equivalent of writing `arr_ref[..]`. That is, if `arr_ref` is of type `&[T; N]`, then `mir_slice_value arr_ref` is of type `&[T]`. Note that `arr_ref` must be a *reference* to an array, not an array itself.

Let's use `mir_slice_value` to write a spec for `sum_of_prefix` when the slice argument is backed by an array of length two:

```
let sum_of_prefix_spec1 = do {
  a_ref <- mir_alloc (mir_array 2 mir_u32);
  a_val <- mir_fresh_var "a" (mir_array 2 mir_u32);
  mir_points_to a_ref (mir_term a_val);

  let s = mir_slice_value a_ref;

  mir_execute_func [s];

  mir_return (mir_term {{ (a_val @ 0) + (a_val @ 1) }});
};
```

The first part of this spec allocates an array reference `a_ref` and declares that it points to a fresh array value `a_val`. The next part declares a slice `s` that is backed by the entirety of `a_ref`, which is then passed as an argument to the function itself. Finally, the return value is declared to be the sum of the first and second elements of `a_val`, which are the same values that back the slice `s` itself.

As noted above, the `sum_of_prefix` function can work with slices of many different lengths. Here is a slight modification to this spec that declares it to take a slice of length 5 rather than a slice of length 2:

```
let sum_of_prefix_spec2 = do {
  a_ref <- mir_alloc (mir_array 5 mir_u32);
  a_val <- mir_fresh_var "a" (mir_array 5 mir_u32);
  mir_points_to a_ref (mir_term a_val);

  let s = mir_slice_value a_ref;

  mir_execute_func [s];

  mir_return (mir_term {{ (a_val @ 0) + (a_val @ 1) }});
};
```

Both of these examples declare a slice whose length matches the length of the underlying array. In general, there is no reason that these have to be the same, and it is perfectly fine for a slice's length to be less than the the length of the underlying array. In Rust, for example, we can write a slice of a

subset of an array by writing `&arr_ref[0..2]`. The SAW equivalent of this can be achieved with the `mir_slice_range_value` function:

```
sawscript> :type mir_slice_range_value
MIRValue -> Int -> Int -> MIRValue
```

`mir_slice_range_value` takes takes two additional `Int` arguments that represent (1) the index to start the slice from, and (2) the index at which the slice ends. For example, `mir_slice_range_value arr_ref 0 2` creates a slice that is backed by the first element (index `0`) and the second element (index `1`) of `arr_ref`. Note that the range `[0..2]` is half-open, so this range does *not* include the third element (index `2`).

For example, here is how to write a spec for `sum_of_prefix` where the slice is a length-2 subset of the original array:

```
let sum_of_prefix_spec3 = do {
  a_ref <- mir_alloc (mir_array 5 mir_u32);
  a_val <- mir_fresh_var "a" (mir_array 5 mir_u32);
  mir_points_to a_ref (mir_term a_val);

  let s = mir_slice_range_value a_ref 0 2;

  mir_execute_func [s];

  mir_return (mir_term {{ (a_val @ 0) + (a_val @ 1) }});
};
```

Note that both `Int` arguments to `mir_slice_range_value` must be concrete (i.e., not symbolic). (See the section below if you want an explanation for why they are not allowed to be symbolic.)

Aside: slices of arbitrary length

After reading the section about slices above, one might reasonably wonder: is there a way to write a more general spec for `sum_of_prefix`: that covers all possible slice lengths `n`, where `n` is greater than or equal to 2? In this case, the answer is "no".

This is a fundamental limitation of the way SAW's symbolic execution works. The full reason for why this is the case is somewhat technical (keep reading if you want to learn more), but the short answer is that if SAW attempts to simulate code whose length is bounded by a symbolic integer, then SAW will go into an infinite loop. To avoid this pitfall, the `mir_slice_range_value` function very deliberately requires the start and end values to be concrete integers, as allowing these values to be symbolic would allow users to inadvertently introduce infinite loops in their specifications.

A longer answer as to why SAW loops forever on computations that are bounded by symbolic lengths: due to the way SAW's symblolic execution works, it creates a complete model of the behavior of a function for all possible inputs. The way that SAW achieves this is by exploring all possible execution paths through a program. If a program involves a loop, for example, then SAW will unroll all iterations

of the loop to construct a model of the loop's behavior. Similarly, if a sequence (e.g., a slice or array) has an unspecified length, then SAW must consider all possible lengths of the array.

SAW's ability to completely characterize the behavior of all paths through a function is one of its strengths, as this allows it to prove theorems that other program verification techniques would not. This strength is also a weakness, however. If a loop has a symbolic number of iterations, for example, then SAW will spin forever trying to unroll the loop. Similarly, if a slice were to have a symbolic length, then SAW would spin forever trying to simulate the program for all possible slice lengths.

In general, SAW cannot prevent users from writing programs whose length is bounded by a symbolic value. For now, however, SAW removes one potential footgun by requiring that slice values always have a concrete length.

7 Overrides and compositional verification

Up until this point, all uses of `mir_verify` in this tutorial have provided an empty list (`[]`) of overrides. This means that any time SAW has simulated a function which calls another function, it will step into the definition of the callee function and verify its behavior alongside the behavior of the callee function. This is a fine thing to do, but it can be inefficient. For example, consider a function like this:

```
pub fn f(x: u32) -> u32 {
    let x1 = g(x);
    let x2 = g(x1);
    g(x2)
}
```

Here, the caller function `f` invokes the callee function `g` three separate times. If we verify `f` with `mir_verify` as we have done up until this point, then SAW must analyze the behavior of `g` three separate times. This is wasteful, and especially so if `g` is a large and complicated function.

This is where *compositional verification* enters the picture. The idea behind compositional verification is that when we prove properties of a caller function, we can reuse properties that we have already proved about callee functions. These properties are captured as *override specifications*, which are also referred to by the shorthand term *overrides*. When a caller invokes a callee with a corresponding override specification, the override's properties are applied without needing to re-simulate the entire function.

As it turns out, the command needed to produce an override specification is already familiar to us—it's `mir_verify`! If you examine the type of this command:

```
sawscript> :type mir_verify
MIRModule -> String -> [MIRSpec] -> Bool -> MIRSetup () -> ProofScript␣
↪() -> TopLevel MIRSpec
```

The returned value is a `MIRSpec`, which captures the behavior of the function that was verified as an

override spec. This override can then be passed to another call to `mir_verify` to use as part of a larger proof.

Let's now try compositional verification in practice. To do so, we will first prove a spec for the `g` function above. For demonstration purposes, we will pick a simplistic implementation of `g`:

```
pub fn g(x: u32) -> u32 {
    x.wrapping_add(1)
}
```

Note that we don't really *have* to use compositional verification when `g` is this simple, as SAW is capable of reasoning about `g`'s behavior directly when proving a spec for `f`. It's still worth going along with this exercise, however, as the same principles of compositional verification apply whether the implementation of `g` is small or large.

The first step of compositional verification is to prove a spec for `g`, the callee function:

```
let g_spec = do {
  x <- mir_fresh_var "x" mir_u32;

  mir_execute_func [mir_term x];

  mir_return (mir_term {{ x + 1 }});
};

g_ov <- mir_verify m "overrides::g" [] false g_spec z3;
```

There's nothing that different about this particular proof from the proofs we've seen before. The only notable difference is that we bind the result of calling `mir_verify` to a `MIRSpec` value that we name `g_ov` (short for "`g` override"). This part is important, as we will need to use `g_ov` shortly.

The next step is to write a spec for `f`. Since `g` adds `1` to its argument, `f` will add `3` to its argument:

```
let f_spec = do {
  x <- mir_fresh_var "x" mir_u32;

  mir_execute_func [mir_term x];

  mir_return (mir_term {{ x + 3 }});
};
```

Again, nothing too surprising. Now let's prove `f` against `f_spec` by using `g_ov` as a compositional override:

```
mir_verify m "overrides::f" [g_ov] false f_spec z3;
```

Here, note that instead of passing an empty list (`[]`) as we have done before, we now pass a list

containing `g_ov`. This informs `mir_verify` that whenever it simulates a call to `g`, it should reuse the properties captured in `g_ov`. In general, we can pass as many overrides as we want (we will see examples of this later in the tutorial), but for now, one override will suffice.

Let's run the proof of `f` against `f_spec`, making sure to pay attention to the output of SAW:

```
[19:06:17.392] Verifying overrides/96c5af24::f[0] ...
[19:06:17.406] Simulating overrides/96c5af24::f[0] ...
[19:06:17.407] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[19:06:17.407] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[19:06:17.407] Applied override! overrides/96c5af24::g[0]
[19:06:17.407] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[19:06:17.407] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[19:06:17.407] Applied override! overrides/96c5af24::g[0]
[19:06:17.407] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[19:06:17.407] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[19:06:17.407] Applied override! overrides/96c5af24::g[0]
[19:06:17.407] Checking proof obligations overrides/96c5af24::f[0] ...
[19:06:17.422] Proof succeeded! overrides/96c5af24::f[0]
```

We've now proven `f` compositionally! The first two lines ("`Verifying ...`" and "`Simulating ...`") and the last two lines ("`Checking proof obligations ...`" and "`Proof succeeded! ...`") are the same as before, but this time, we have some additional lines of output in between:

- Whenever SAW prints "`Matching <N> overrides of <function>`", that's when you know that SAW is about to simulate a call to `<function>`. At that point, SAW will check to see how many overrides (`<N>`) for `<function>` are available.
- Whenever SAW prints "`Brancing on <N> override variants of <function>`", SAW is trying to figure out which of the `<N>` overrides to apply. In this example, there is only a single override, so the choice is easy. In cases where there are multiple overrides, however, SAW may have to work harder (possibly even consulting an SMT solver) to figure out which override to use.
- If SAW successfully picks an override to apply, it will print "`Applied override! ...`".

In the example above, we used a single `g` override that applies for all possible arguments. In general, however, there is no requirement that overrides must work for all arguments. In fact, it is quite common for SAW verification efforts to write different specifications for the same function, but with different arguments. We can then provide multiple overrides for the same function as part of a compositional verification, and SAW will be able to pick the right override depending on the shape of the argument when invoking the function being overridden.

For example, let's suppose that we wrote different `g` specs, one where the argument to `g` is even, and another where the argument to `g` is odd:

```
let g_even_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_precond {{ x % 2 == 0 }};

  mir_execute_func [mir_term x];

  mir_return (mir_term {{ x + 1 }});
};

let g_odd_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_precond {{ x % 2 == 1 }};

  mir_execute_func [mir_term x];

  mir_return (mir_term {{ x + 1 }});
};

g_even_ov <- mir_verify m "overrides::g" [] false g_even_spec z3;
g_odd_ov  <- mir_verify m "overrides::g" [] false g_odd_spec  z3;
```

We can then prove `f` compositionally by passing both of the `g` overrides to `mir_verify`:

```
mir_verify m "overrides::f" [g_even_ov, g_odd_ov] false f_spec z3;
```

Like before, this will successfully verify. The only different now is that SAW will print output involving two overrides instead of just one:

```
[20:48:07.649] Simulating overrides/96c5af24::f[0] ...
[20:48:07.650] Matching 2 overrides of  overrides/96c5af24::g[0] ...
[20:48:07.650] Branching on 2 override variants of overrides/
↪96c5af24::g[0] ...
[20:48:07.652] Applied override! overrides/96c5af24::g[0]
...
```

Keep in mind that if you provide at least one override for a function as part of a compositional verification, then SAW *must* apply an override whenever it invokes that function during simulation. If SAW cannot find a matching override, then the verification will fail. For instance, consider what would happen if you tried proving `f` like so:

```
mir_verify m "overrides::f" [g_even_ov] false f_spec z3;
```

This time, we supply one override for `g` that only matches when the argument is even. This is a problem, as SAW will not be able to find a matching override when the argument is odd. Indeed, SAW will fail to verify this:

```
[20:53:29.588] Verifying overrides/96c5af24::f[0] ...
[20:53:29.602] Simulating overrides/96c5af24::f[0] ...
[20:53:29.602] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[20:53:29.602] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[20:53:29.603] Applied override! overrides/96c5af24::g[0]
[20:53:29.603] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[20:53:29.603] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[20:53:29.604] Applied override! overrides/96c5af24::g[0]
[20:53:29.604] Matching 1 overrides of  overrides/96c5af24::g[0] ...
[20:53:29.604] Branching on 1 override variants of overrides/
↪96c5af24::g[0] ...
[20:53:29.605] Applied override! overrides/96c5af24::g[0]
[20:53:29.605] Symbolic simulation completed with side conditions.
[20:53:29.606] Checking proof obligations overrides/96c5af24::f[0] ...
[20:53:29.623] Subgoal failed: overrides/96c5af24::f[0] No override␣
↪specification applies for overrides/96c5af24::g[0].
Arguments:
- c@26:bv
Run SAW with --sim-verbose=3 to see a description of each override.
[20:53:29.623] SolverStats {solverStatsSolvers = fromList ["SBV->Z3"],␣
↪solverStatsGoalSize = 388}
[20:53:29.624] ----------Counterexample----------
[20:53:29.624]   x: 1
...
Proof failed.
```

Here, we can see that `No override specification applies`, and SAW also generates a counterexample of `x: 1`. Sure enough, `1` is an odd number!

7.1 Overrides and mutable references

Compositional overrides provide great power, as they effectively allow you to skip over certain functions when simulating them and replace them with simpler implementations. With great power comes great responsibility, however. In particular, one must be careful when using overrides for functions that modify mutable references. If an override does not properly capture the behavior of a mutable reference, it could potentially lead to incorrect proofs.

This is the sort of thing that is best explained with an example, so consider these two functions:

```
pub fn side_effect(a: &mut u32) {
    *a = 0;
}
```

(continues on next page)

(continued from previous page)

```
pub fn foo(x: u32) -> u32 {
    let mut b: u32 = x;
    side_effect(&mut b);
    b
}
```

The `side_effect` function does not return anything interesting; it is only ever invoked to perform a side effect of changing the mutable reference `a` to point to `0`. The `foo` function invokes `side_effect`, and as a result, it will always return `0`, regardless of what the argument to `foo` is. No surprises just yet.

Now let's make a first attempt at verifying `foo` using compositional verification. First, we will write a spec for `side_effect`:

```
let side_effect_spec = do {
  a_ref <- mir_alloc_mut mir_u32;
  a_val <- mir_fresh_var "a_val" mir_u32;
  mir_points_to a_ref (mir_term a_val);
  mir_execute_func [a_ref];
};
```

`side_effect_spec` is somewhat odd. Although it goes through the effort of allocating a mutable reference `a_ref` and initializing it, nothing about this spec states that `a_ref` will point to `0` after the function has been invoked. This omission is strange, but not outright wrong—the spec just underspecifies what the behavior of the function is. Indeed, SAW will successfully verify this spec using `mir_verify`:

```
side_effect_ov <- mir_verify m "overrides_mut::side_effect" [] false␣
↪side_effect_spec z3;
```

Next, let's try to write a spec for `foo`:

```
let foo_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_execute_func [mir_term x];
  mir_return (mir_term {{ x }});
};
```

At this point, alarm bells should be going off in your head. This spec incorrectly states that `foo(x)` should return `x`, but it should actually return `0`! This looks wrong, but consider what would happen if you tried to verify this compositionally using our `side_effect_ov` override:

```
mir_verify m "overrides_mut::foo" [side_effect_ov] false foo_spec z3;
```

If SAW were to simulate `foo(x)`, it would invoke create a temporary variable `b` and assign it to the value `x`, and then it would invoke `side_effect(&mut b)`. At this point, the `side_effect_ov` override would apply. According to `side_effect_spec`, the argument to `side_effect` is not modified at all after the function returns. This means that when the `foo` function returns `b`, it will still retain its initial value of `x`. This shows that if we were to use `side_effect_ov`, we could prove something that's blatantly false!

Now that we've made you sweat a little bit, it's time for some good news: SAW won't *actually* let you prove `foo_spec`. If you try this compositional proof in practice, SAW will catch your mistake:

```
[14:50:29.170] Verifying overrides_mut/11e47708::foo[0] ...
[14:50:29.181] Simulating overrides_mut/11e47708::foo[0] ...
[14:50:29.181] Matching 1 overrides of  overrides_mut/11e47708::side_
↪effect[0] ...
[14:50:29.181] Branching on 1 override variants of overrides_mut/
↪11e47708::side_effect[0] ...
...
State of memory allocated in precondition (at overrides-mut-fail.
↪saw:6:12) not described in postcondition
```

The line of code that SAW points to in the "`State of memory ...`" error message is:

```
  a_ref <- mir_alloc_mut mir_u32;
```

SAW informs us that although we allocated the mutable reference `a_ref`, we never indicated what it should point to after the function has returned. This is an acceptable (if somewhat unusual) thing to do when verifying `side_effect_spec` using `mir_verify`, but it is *not* acceptable to do this when using this spec as an override. To avoid unsound behavior like what is described above, any override that allocates a mutable reference in its preconditions *must* declare what its value should be in the postconditions, no exceptions.

Thankfully, repairing this spec is relatively straightforward. Simply add a `mir_points_to` statement in the postconditions of `side_effect_spec`:

```
let side_effect_spec = do {
  a_ref <- mir_alloc_mut mir_u32;
  a_val <- mir_fresh_var "a_val" mir_u32;
  mir_points_to a_ref (mir_term a_val);
  mir_execute_func [a_ref];

  // This is new
  mir_points_to a_ref (mir_term {{ 0 : [32] }});
};
```

Then use the correct return value in `foo_spec`:

```
let foo_spec = do {
  x <- mir_fresh_var "x" mir_u32;
  mir_execute_func [mir_term x];

  // This is new
  mir_return (mir_term {{ 0 : [32] }});
};
```

And now the compositional proof of `foo_spec` works!

7.2 Unsafe overrides

Now that we've made it this far into the tutorial, it's time to teach you a more advanced technique: *unsafe* overrides. Up until this point, we have relied on SAW to check all of our work, and this is usually what you'd want from a formal verification tool. In certain circumstances, however, it can be useful to say "I know what I'm doing, SAW—just believe me when I say this spec is valid!" In order to say this, you can use `mir_unsafe_assume_spec`:

```
sawscript> :type mir_unsafe_assume_spec
MIRModule -> String -> MIRSetup () -> TopLevel MIRSpec
```

`mir_unsafe_assume_spec` is `mir_verify`'s cousin who likes to live a little more dangerously. Unlike `mir_verify`, the specification that you pass to `mir_unsafe_assume_spec` (the `MIRSetup ()` argument) is *not* checked for full correctness. That is, `mir_unsafe_assume_spec` will bypass SAW's usual symbolic execution pipeline, which is why one does not need to pass a `ProofScript` argument (e.g., `z3`) to `mir_unsafe_assume_spec`. SAW will believe whatever spec you supply `mir_unsafe_assume_spec` to be valid, and the `MIRSpec` that `mir_unsafe_assume_spec` returns can then be used in later compositional verifications.

Why would you want to do this? The main reason is that writing proofs can be difficult, and sometimes, there are certain functions in a SAW verification effort that are disproportionately harder to write a spec for than others. It is tempting to write specs for each function in sequence, but this can run the risk of getting stuck on a particularly hard-to-verify function, blocking progress on other parts of the proofs.

In these situations, `mir_unsafe_assume_spec` can be a useful prototyping tool. One can use `mir_unsafe_assume_spec` to assume a spec for the hard-to-verify function and then proceed with the remaining parts of the proof. Of course, you should make an effort to go back and prove the hard-to-verify function's spec later, but it can be nice to try something else first.

For example, here is how one can unsafely assume `g_spec` and use it in a compositional proof of `f_spec`:

```
g_ov <- mir_unsafe_assume_spec m "overrides::g" g_spec;
mir_verify m "overrides::f" [g_ov] false f_spec z3;
```

It should be emphasized that when we say "`unsafe`", we really mean it. `mir_unsafe_assume_spec` can be used to prove specs that are blatantly wrong, so use it with caution.

8 Static items

Sometimes, Rust code makes use of *static items* (https://doc.rust-lang.org/reference/items/static-items.html), which are definitions that are defined in a precise memory location for the entire duration of the program. As such, static items can be thought of as a form of global variables.

8.1 Immutable static items

There are two kinds of static items in Rust: mutable static items (which have a `mut` keyword) and immutable static items (which lack `mut`). Immutable static items are much easier to deal with, so let's start by looking at an example of a program that uses immutable static data:

```
static ANSWER: u32 = 42;

pub fn answer_to_the_ultimate_question() -> u32 {
    ANSWER
}
```

This function will return `ANSWER`, i.e., `42`. We can write a spec that says as much:

```
let answer_to_the_ultimate_question_spec1 = do {
  mir_execute_func [];

  mir_return (mir_term {{ 42 : [32] }});
};
```

This works, but it is somewhat unsatisfying, as it requires hard-coding the value of `ANSWER` into the spec. Ideally, we'd not have to think about the precise implementation of static items like `ANSWER`. Fortunately, SAW makes this possible by providing a `mir_static_initializer` function which computes the initial value of a static item at the start of the program:

```
sawscript> :type mir_static_initializer
String -> MIRValue
```

In this case, `mir_static_initializer "statics::ANSWER"` is equivalent to writing `mir_term {{ 42 : [32] }}`, so this spec is also valid:

```
let answer_to_the_ultimate_question_spec2 = do {
  mir_execute_func [];
```

(continues on next page)

(continued from previous page)

```
  mir_return (mir_static_initializer "statics::ANSWER");
};
```

Like `mir_verify`, the `mir_static_initializer` function expects a full identifier as an argument, so we must write `"statics::ANSWER"` instead of just "ANSWER".

At the MIR level, there is a unique reference to every static item. You can obtain this reference by using the `mir_static` function:

```
sawscript> :type mir_static
String -> MIRValue
```

Here is one situation in which you would need to use a *reference* to a static item (which `mir_static` computes) rather than the *value* of a static item (which `mir_static_initializer` computes):

```
pub fn answer_in_ref_form() -> &'static u32 {
    &ANSWER
}
```

A spec for this function would look like this:

```
let answer_in_ref_form_spec = do {
  mir_execute_func [];

  mir_return (mir_static "statics::ANSWER");
};
```

That's about all there is to say regarding immutable static items.

8.2 Mutable static items

Mutable static items are a particularly tricky aspect of Rust. Including a mutable static item in your program is tantamount to having mutable global state that any function can access and modify. They are so tricky, in fact, that Rust does not even allow you to use them unless you surround them in an `unsafe` block:

```
static mut MUT_ANSWER: u32 = 42;

pub fn mut_answer_to_the_ultimate_question() -> u32 {
    unsafe { MUT_ANSWER }
}
```

The `mir_static_initializer` and `mut_static` functions work both immutable and mutable static items, so we can write specs for mutable items using mostly the same techniques as for writing specs for immutable items. We must be careful, however, as SAW is pickier when it comes

to specifying the initial values of mutable static items. For example, here is naïve attempt at porting the spec for `answer_to_the_ultimate_question` over to its mutable static counterpart, `mut_answer_to_the_ultimate_question`:

```
let mut_answer_to_the_ultimate_question_spec = do {
  mir_execute_func [];

  mir_return (mir_static_initializer "statics::MUT_ANSWER");
};
```

This looks plausible, but SAW will fail to verify it:

```
[21:52:32.738] Verifying statics/28a97e47::mut_answer_to_the_ultimate_
↪question[0] ...
[21:52:32.745] Simulating statics/28a97e47::mut_answer_to_the_ultimate_
↪question[0] ...
...
Symbolic execution failed.
Abort due to assertion failure:
  statics.rs:14:14: 14:24: error: in statics/28a97e47::mut_answer_to_
↪the_ultimate_question[0]
  attempted to read empty mux tree
```

Oh no! Recall that we have seen this "`attempted to read empty mux tree`" error message once before when discussing reference types. This error arose when we attempted to read from uninitialized memory from a reference value. The same situation applies here. A static item is backed by a reference, and SAW deliberately does *not* initialize the memory that a mutable static reference points to upon program startup. Since we did not initialize `MUT_ANSWER`'s reference value in the preconditions of the spec, SAW crashes at simulation time when it attempts to read from the uninitialized memory.

The solution to this problem is to perform this initialization explicitly using `mir_points_to` in the preconditions of the spec. For example, this is a valid spec:

```
let mut_answer_to_the_ultimate_question_spec = do {
  let mut_answer = "statics::MUT_ANSWER";
  let mut_answer_init = mir_static_initializer mut_answer;
  mir_points_to (mir_static mut_answer) mut_answer_init;

  mir_execute_func [];

  mir_return mut_answer_init;
};
```

We don't necessarily have to use `mir_static_initializer` as the starting value for `MUT_ANSWER`, however. This spec, which uses `27` as the starting value, is equally valid:

```
let mut_answer_to_the_ultimate_question_alt_spec = do {
  let alt_answer = mir_term {{ 27 : [32] }};
  mir_points_to (mir_static "statics::MUT_ANSWER") alt_answer;

  mir_execute_func [];

  mir_return alt_answer;
};
```

At this point, you are likely wondering: why do we need to explicitly initialize mutable static references but not immutable static references? After all, when we wrote a spec for `answer_to_the_ultimate_question` earlier, we managed to get away with not initializing the reference for `ANSWER` (which is an immutable static item). The difference is that the value of a mutable static item can change over the course of a program, and SAW requires that you be very careful in specifying what a mutable static value is at the start of a function. For example, consider a slightly extended version of the earlier Rust code we saw:

```
static mut MUT_ANSWER: u32 = 42;

pub fn mut_answer_to_the_ultimate_question() -> u32 {
    unsafe { MUT_ANSWER }
}

pub fn alternate_universe() -> u32 {
    unsafe {
        MUT_ANSWER = 27;
    }
    mut_answer_to_the_ultimate_question()
}
```

Suppose someone were to ask you "what value does `mut_answer_to_the_ultimate_question` return?" This is not a straightforward question to answer, as the value that it returns depends on the surrounding context. If you were to call `mut_answer_to_the_ultimate_question` right as the program started, it would return `42`. If you were to call `mut_answer_to_the_ultimate_question` as part of the implementation of `alternate_universe`, however, then it would return `27`! This is an inherent danger of using mutable static items, as they can be modified at any time by any function. For this reason, SAW requires you to be explicit about what the initial values of mutable static items should be.

Mutable static items and compositional overrides

In the "Overrides and mutable references" section, we discussed the potential pitfalls of using mutable references in compositional overrides. Mutable static items are also mutable values that are backed by references, and as such, they are also subject to the same pitfalls. Let's see an example of this:

```
static mut A: u32 = 42;

pub fn side_effect() {
    unsafe {
        A = 0;
    }
}

pub fn foo() -> u32 {
    side_effect();
    unsafe { A }
}
```

The setup is almost the same, except that instead of passing a mutable reference as an argument to `side_effect`, we instead declare a mutable static item `A` that is shared between `side_effect` and `foo`. We could potentially write SAW specs for `side_effect` and `foo` like these:

```
let a = "statics_compositional::A";

let side_effect_spec = do {
  mir_points_to (mir_static a) (mir_static_initializer a);
  mir_execute_func [];
};

let foo_spec = do {
  let a_init = mir_static_initializer a;
  mir_points_to (mir_static a) a_init;
  mir_execute_func [];
  mir_return a_init;
};

side_effect_ov <- mir_verify m "statics_compositional::side_effect" []␣
↪false side_effect_spec z3;
mir_verify m "statics_compositional::foo" [side_effect_ov] false foo_
↪spec z3;
```

Note that we have once again underspecified the behavior of `side_effect`, as we do not say what `A`'s value should be in the postconditions of `side_effect_spec`. Similarly, `foo_spec` is wrong, as it should return `0` rather than the initial value of `A`. By similar reasoning as before, we run the risk that using `side_effect_ov` could lead use to prove something incorrect. Thankfully, SAW can

also catch this sort of mistake:

```
[15:46:38.525] Verifying statics_compositional/16fea9c0::side_effect[0]␣
↪...
[15:46:38.533] Simulating statics_compositional/16fea9c0::side_
↪effect[0] ...
[15:46:38.533] Checking proof obligations statics_compositional/
↪16fea9c0::side_effect[0] ...
[15:46:38.533] Proof succeeded! statics_compositional/16fea9c0::side_
↪effect[0]
[15:46:38.533] Verifying statics_compositional/16fea9c0::foo[0] ...
[15:46:38.542] Simulating statics_compositional/16fea9c0::foo[0] ...
[15:46:38.542] Matching 1 overrides of  statics_compositional/
↪16fea9c0::side_effect[0] ...
[15:46:38.542] Branching on 1 override variants of statics_
↪compositional/16fea9c0::side_effect[0] ...
...
State of mutable static variable "statics_compositional/16fea9c0::A[0]"␣
↪not described in postcondition
```

To repair this proof, add a `mir_points_to` statement in the postconditions of `side_effect_spec`:

```
let side_effect_spec = do {
  mir_points_to (mir_static a) (mir_static_initializer a);
  mir_execute_func [];

  // This is new
  mir_points_to (mir_static a) (mir_term {{ 0 : [32] }});
};
```

And then correct the behavior of `foo_spec`:

```
let foo_spec = do {
  let a_init = mir_static_initializer a;
  mir_points_to (mir_static a) a_init;
  mir_execute_func [];

  // This is new
  mir_return (mir_term {{ 0 : [32] }});
};
```

Be warned that if your program declares any mutable static items, then any compositional override *must* state what the value of each mutable static item is in its postconditions. This applies *even if the override does not directly use the mutable static items*. For example, if we had declared a second

mutable static item alongside `A`:

```
static mut A: u32 = 42;
static mut B: u32 = 27;
```

Then `side_effect_spec` would need an additional `mir_points_to` statement involving `B` to satisfy this requirement. This requirement is somewhat heavy-handed, but it is necessary in general to avoid unsoundness. Think carefully before you use mutable static items!

9 Case study: Salsa20

If you've made it this far into the tutorial, congrats! You've now been exposed to all of the SAW fundamentals that you need to verify Rust code found in the wild. Of course, talking about verifying real-world code is one thing, but actually *doing* the verification is another thing entirely. Making the jump from the small examples to "industrial-strength" code can be intimidating.

To make this jump somewhat less frightening, the last part of this tutorial will consist of a case study in using SAW to verify a non-trivial piece of Rust code. In particular, we will be looking at a Rust implementation of the Salsa20 (https://en.wikipedia.org/wiki/Salsa20) stream cipher. We do not assume any prior expertise in cryptography or stream ciphers in this tutorial, so don't worry if you are not familiar with Salsa20.

More than anything, this case study is meant to emphasize that verification is an iterative process. It's not uncommon to try something with SAW and encounter an error message. That's OK! We will explain what can go wrong when verifying Salsa20 and how to recover from these mistakes. Later, if you encounter similar issues when verifying your own code with SAW, the experience you have developed when developing these proofs can inform you of possible ways to fix the issues.

9.1 The `salsa20` crate

The code for this Salsa20 implementation we will be verifying can be found under the `code/salsa20` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20) subdirectory. This code is adapted from version 0.3.0 of the `salsa20` crate, which is a part of the `stream-ciphers` (https://github.com/RustCrypto/stream-ciphers) project. The code implements Salsa20 as well as variants such as HSalsa20 and XSalsa20, but we will only be focusing on the original Salsa20 cipher in this tutorial.

The parts of the crate that are relevant for our needs are mostly contained in the `src/core.rs` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/src/core.rs) file, as well as some auxiliary definitions in the `src/rounds.rs` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/src/rounds.rs) and `src/lib.rs` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/src/lib.rs) files. You can take a look at these files if you'd like, but you don't need to understand everything in them just yet. We will introduce the relevant parts of the code in the tutorial as they come up.

9.2 Salsa20 preliminaries

Salsa20 is a stream cipher, which is a cryptographic technique for encrypting and decrypting messages. A stream cipher encrypts a message by combining it with a *keystream* to produce a ciphertext (the encrypted message). Moreover, the same keystream can then be combined with the ciphertext to decrypt it back into the original message.

The original author of Salsa20 has published a specification for Salsa20 here (https://cr.yp.to/snuffle/spec.pdf). This is a great starting point for a formal verification project, as this gives us a high-level description of Salsa20's behavior that will guide us in proving the functional correctness of the `salsa20` crate. When we say that `salsa20` is functionally correct, we really mean "proven correct with respect to the Salsa20 specification".

The first step in our project would be to port the Salsa20 spec to Cryptol code, as we will need to use this code when writing SAW proofs. The process of transcribing an English-language specification to executable Cryptol code is interesting in its own right, but it is not the primary focus of this tutorial. As such, we will save you some time by providing a pre-baked Cryptol implementation of the Salsa20 spec here (https://github.com/GaloisInc/saw-script/blob/master/doc/rust-tutorial/code/salsa20/Salsa20.cry). (This implementation is adapted (https://github.com/GaloisInc/cryptol-specs/blob/1366ccf71db9dca58b16ff04ca7d960a4fe20e34/Primitive/Symmetric/Cipher/Stream/Salsa20.cry) from the `cryptol-specs` (https://github.com/GaloisInc/cryptol-specs) repo.)

Writing the Cryptol version of the spec is only half the battle, however. We still have to prove that the Rust implementation in the `salsa20` crate adheres to the behavior prescribed by the spec, which is where SAW enters the picture. As we will see shortly, the code in `salsa20` is not a direct port of the pseudocode shown in the Salsa20 spec, as it is somewhat more low-level. SAW's role is to provide us assurance that the behavior of the low-level Rust code and the high-level Cryptol code coincide.

9.3 A note about cryptographic security

As noted in the previous section, our goal is to prove that the behavior of `salsa20` functions is functionally correct. This property should *not* be confused with cryptographic security. While functional correctness is an important aspect of cryptographic security, a full cryptographic security audit would encompass additional properties such as whether the code runs in constant time on modern CPUs. As such, the SAW proofs we will write would not constitute a full security audit (and indeed, the `salsa20 README` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/README.md) states that the crate has never received such an audit).

9.4 An overview of the `salsa20` code

Before diving into proofs, it will be helpful to have a basic understanding of the functions and data types used in the `salsa20` crate. Most of the interesting code lives in `src/core.rs` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/src/core.rs). At the top of this file, we have the `Core` struct:

```rust
pub struct Core<R: Rounds> {
    /// Internal state of the core function
    state: [u32; STATE_WORDS],

    /// Number of rounds to perform
    rounds: PhantomData<R>,
}
```

Let's walk through this:

- The `state` field is an array that is `STATE_WORDS` elements long, where `STATE_WORDS` is a commonly used alias for `16`:

  ```rust
  /// Number of 32-bit words in the Salsa20 state
  const STATE_WORDS: usize = 16;
  ```

- The `rounds` field is of type `PhantomData<R>`. If you haven't seen it before, `PhantomData<R>` (https://doc.rust-lang.org/std/marker/struct.PhantomData.html) is a special type that tells the Rust compiler to pretend as though the struct is storing something of type `R`, even though a `PhantomData` value will not take up any space at runtime.

The reason that `Core` needs a `PhantomData<R>` field is because `R` implements the `Rounds` trait:

```rust
//! Numbers of rounds allowed to be used with a Salsa20 family stream␣
↪cipher

pub trait Rounds: Copy {
    const COUNT: usize;
}
```

A core operation in Salsa20 is hashing its input through a series of *rounds*. The `COUNT` constant indicates how many rounds should be performed. The Salsa20 spec assumes 20 rounds:

```rust
/// 20-rounds (Salsa20/20)
#[derive(Copy, Clone)]
pub struct R20;

impl Rounds for R20 {
    const COUNT: usize = 20;
}
```

However, there are also reduced-round variants that perform 8 and 12 rounds, respectively:

```rust
/// 8-rounds (Salsa20/8)
#[derive(Copy, Clone)]
```

(continues on next page)

(continued from previous page)

```
pub struct R8;

impl Rounds for R8 {
    const COUNT: usize = 8;
}

/// 12-rounds (Salsa20/12)
#[derive(Copy, Clone)]
pub struct R12;

impl Rounds for R12 {
    const COUNT: usize = 12;
}
```

Each number of rounds has a corresponding struct whose names begins with the letter `R`. For instance, a `Core<R20>` value represents a 20-round Salsa20 cipher. Here is the typical use case for a `Core` value:

- A `Core` value is created using the `new` function:

  ```
      pub fn new(key: &Key, iv: &Nonce) -> Self {
  ```

  We'll omit the implementation for now. This function takes a secret `Key` value and a unique `Nonce` value and uses them to produce the initial `state` in the `Core` value.

- After creating a `Core` value, the `counter_setup` and `rounds` functions are used to produce the Salsa20 keystream:

  ```
      pub(crate) fn counter_setup(&mut self, counter: u64) {
  ```

  ```
      fn rounds(&mut self, state: &mut [u32; STATE_WORDS]) {
  ```

  We'll have more to say about these functions later.

- The *pièce de résistance* is the `apply_keystream` function. This takes a newly created `Core` value, produces its keystream, and applies it to a message to produce the `output`:

  ```
      pub fn apply_keystream(&mut self, counter: u64, output: &mut␣
  ↪[u8]) {
  ```

Our ultimate goal is to verify the `apply_keystream` function, which is the Rust equivalent of the Salsa20 encryption function described in the spec.

9.5 Building `salsa20`

The next step is to build the `salsa20` crate. Unlike the examples we have seen up to this point, which have been self-contained Rust files, `salsa20` is a `cargo`-based project. As such, we will need to build it using `cargo saw-build`, an extension to the `cargo` package manager that integrates with `mir-json`. Before you proceed, make sure that you have defined the `SAW_RUST_LIBRARY_PATH` environment variable as described in *this section*.

To build the `salsa20` crate, perform the following steps:

```
$ cd code/salsa20/
$ cargo saw-build
```

Near the end of the build output, you will see a line that looks like this:

```
linking 9 mir files into <...>/saw-script/doc/rust-tutorial/code/
↪salsa20/target/x86_64-unknown-linux-gnu/debug/deps/salsa20-
↪dd0d90f28492b9cb.linked-mir.json
```

This is the location of the MIR JSON file that we will need to provide to SAW. (When we tried it, the hash in the file name was `dd0d90f28492b9cb`, but it could very well be different on your machine.) Due to how `cargo` works, the location of this file is in a rather verbose, hard-to-remember location. For this reason, we recommend copying this file to a different path, e.g.,

```
$ cp <...>/saw-script/doc/rust-tutorial/code/salsa20/target/x86_64-
↪unknown-linux-gnu/debug/deps/salsa20-dd0d90f28492b9cb.linked-mir.json␣
↪code/salsa20/salsa20.linked-mir.json
```

As a safeguard, we have also checked in a compressed version of this MIR JSON file as `code/salsa20/salsa/salsa20.linked-mir.json.tar.gz` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/salsa20.linked-mir.json.tar.gz). In a pinch, you can extract this archive to obtain a copy of the MIR JSON file, which is approximately 4.6 megabytes when uncompressed.

9.6 Getting started with SAW

Now that we've built the `salsa20` crate, it's time to start writing some proofs! Let's start a new `code/salsa20/salsa20.saw` file as fill it in with the usual preamble:

```
enable_experimental;

m <- mir_load_module "salsa20.linked-mir.json";
```

We are also going to need to make use of the Cryptol implementation of the Salsa20 spec, which is defined in `code/salsa20/Salsa20.cry` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/Salsa20.cry). SAW allows you to import standalone Cryptol `.cry` files by using the `import` command:

```
import "Salsa20.cry";
```

As an aside, note that we have also checked in a `code/salsa20/salsa20-reference.saw` (https://github.com/GaloisInc/saw-script/tree/master/doc/rust-tutorial/code/salsa20/salsa20-reference.saw), which contains a complete SAW file. We encourage you *not* to look at this file for now, since following along with the tutorial is meant to illustrate the "a-ha moments" that one would have in the process of writing the proofs. In you become stuck while following along and absolutely need a hint, however, then this file can help you become unstuck.

9.7 Verifying our first `salsa20` function

Now it's time to start verifying some `salsa20` code. But where do we start? It's tempting to start with `apply_keystream`, which is our end goal. This is likely going to be counter-productive, however, as `apply_keystream` is a large function with several moving parts. Throwing SAW at it immediately is likely to cause it to spin forever without making any discernible progress.

For this reason, we will instead take the approach of working from the bottom-up. That is, we will first verify the functions that `apply_keystream` transitively invokes, and then leverage compositional verification to verify a proof of `apply_keystream` using overrides. This approach naturally breaks up the problem into smaller pieces that are easier to understand in isolation.

If we look at the implementation of `apply_keystream`, we see that it invokes the `round` function, which in turn invokes the `quarter_round` function:

```
pub(crate) fn quarter_round(
    a: usize,
    b: usize,
    c: usize,
    d: usize,
    state: &mut [u32; STATE_WORDS],
) {
    let mut t: u32;

    t = state[a].wrapping_add(state[d]);
    state[b] ^= t.rotate_left(7) as u32;

    t = state[b].wrapping_add(state[a]);
    state[c] ^= t.rotate_left(9) as u32;

    t = state[c].wrapping_add(state[b]);
    state[d] ^= t.rotate_left(13) as u32;

    t = state[d].wrapping_add(state[c]);
    state[a] ^= t.rotate_left(18) as u32;
}
```

`quarter_round` is built on top of the standard library functions `wrapping_add` (https://doc.rust-lang.org/std/primitive.usize.html#method.wrapping_add) and `rotate_left` (https://doc.rust-lang.org/std/primitive.usize.html#method.rotate_left), so we have finally reached the bottom of the call stack. This makes `quarter_round` a good choice for the first function to verify.

The implementation of the Rust `quarter_round` function is quite similar to the Cryptol `quarterround` function in `Salsa20.cry`:

```
quarterround : [4][32] -> [4][32]
quarterround [y0, y1, y2, y3] = [z0, z1, z2, z3]
  where
    z1 = y1 ^ ((y0 + y3) <<< 0x7)
    z2 = y2 ^ ((z1 + y0) <<< 0x9)
    z3 = y3 ^ ((z2 + z1) <<< 0xd)
    z0 = y0 ^ ((z3 + z2) <<< 0x12)
```

The Cryptol `quarterround` function doesn't have anything like the `state` argument in the Rust `quarter_round` function, but let's not fret about that too much yet. Our SAW spec is going to involve `quarterround` *somehow*—we just have to figure out how to make it fit.

Let's start filling out the SAW spec for `quarter_round`:

```
let quarter_round_spec = do {
```

We are going to need some fresh variables for the `a`, `b`, `c`, and `d` arguments:

```
  a <- mir_fresh_var "a" mir_usize;
  b <- mir_fresh_var "b" mir_usize;
  c <- mir_fresh_var "c" mir_usize;
  d <- mir_fresh_var "d" mir_usize;
```

We will also need to allocate a reference for the `state` argument. The reference's underlying type is `STATE_WORDS` (`16`) elements long:

```
  state_ref <- mir_alloc_mut (mir_array STATE_WORDS mir_u32);
  state_arr <- mir_fresh_var "state" (mir_array STATE_WORDS mir_u32);
  mir_points_to state_ref (mir_term state_arr);
```

Finally, we will need to pass these arguments to the function:

```
  mir_execute_func [ mir_term a
                   , mir_term b
                   , mir_term c
                   , mir_term d
                   , state_ref
                   ];
```

(continues on next page)

(continued from previous page)

```
};
```

With that, we have a spec for `quarter_round`! It's not very interesting just yet, as we don't specify what `state_ref` should point to after the function has returned. But that's fine for now. When developing a SAW proof, it can be helpful to first write out the "skeleton" of a function spec that only contains the call to `mir_execute_func`, without any additional preconditions or postconditions. We can add those later after ensuring that the skeleton works as expected.

Let's check our progress thus far by running this through SAW:

```
$ saw salsa20.saw
...
[23:16:05.080] Type errors:
  salsa20.saw:12:39-12:68: Unbound variable: "STATE_WORDS" (salsa20.
↪saw:12:49-12:60)
Note that some built-in commands are available only after running
either `enable_deprecated` or `enable_experimental`.

  salsa20/salsa20.saw:11:31-11:60: Unbound variable: "STATE_WORDS"␣
↪(salsa20.saw:11:41-11:52)
Note that some built-in commands are available only after running
either `enable_deprecated` or `enable_experimental`.
```

We've already run into some type errors. Not too surprising, considering this was our first attempt. The error message contains that `STATE_WORDS` is unbound. This makes sense if you think about it, as `STATE_WORDS` is defined in the Rust code, but not in the SAW file itself. Let's fix that by adding this line to `salsa20.saw`:

```
let STATE_WORDS = 16;
```

That change fixes the type errors in `quarter_round_spec`. Hooray! Let's press on.

Next, we need to add a call to `mir_verify`. In order to do this, we need to know what the full identifier for the `quarter_round` function is. Because it is defined in the `salsa20` crate and in the `core.rs` file, so we would expect the identifier to be named `salsa20::core::quarter_round`:

```
quarter_round_ov <-
  mir_verify m "salsa20::core::quarter_round" [] false quarter_round_
↪spec z3;
```

However, SAW disagrees:

```
[00:22:56.970] Stack trace:
"mir_verify" (salsa20.saw:26:3-26:13)
Couldn't find MIR function named: salsa20::core::quarter_round
```

Ugh. This is a consequence of how `mir-json` disambiguates identifiers. Because there is a separate `core` crate in the Rust standard libraries, `mir-json` uses "`core#1`", a distinct name, to refer to the `core.rs` file. You can see this for yourself by digging around in the MIR JSON file, if you'd like. (In a future version of SAW, one will be able to look this name up (https://github.com/GaloisInc/saw-script/issues/1980) more easily.)

Once we change the identifier:

```
quarter_round_ov <-
  mir_verify m "salsa20::core#1::quarter_round" [] false quarter_round_
↪spec z3;
```

We can run SAW once more. This time, SAW complains about a different thing:

```
[01:00:19.697] Verifying salsa20/10e438b3::core#1[0]::quarter_round[0] .
↪..
[01:00:19.714] Simulating salsa20/10e438b3::core#1[0]::quarter_round[0]␣
↪...
[01:00:19.717] Checking proof obligations salsa20/10e438b3::core
↪#1[0]::quarter_round[0] ...
[01:00:19.739] Subgoal failed: salsa20/10e438b3::core#1[0]::quarter_
↪round[0] index out of bounds: the length is move _10 but the index is␣
↪_9
[01:00:19.739] SolverStats {solverStatsSolvers = fromList ["SBV->Z3"],␣
↪solverStatsGoalSize = 53}
[01:00:19.739] ----------Counterexample----------
[01:00:19.739]   a: 2147483648
```

Here, SAW complains that we have an `index out of bounds`. Recall that we are indexing into the `state` array, which is of length 16, using the `a`/`b`/`c`/`d` arguments. Each of these arguments are of type `usize`, and because we are declaring these to be symbolic, it is quite possible for each argument to be 16 or greater, which would cause the index into `state` to be out of bounds.

In practice, however, the only values of `a`/`b`/`c`/`d` that we will use are less than 16. We can express this fact as a precondition:

```
  mir_precond {{ a < `STATE_WORDS /\
                 b < `STATE_WORDS /\
                 c < `STATE_WORDS /\
                 d < `STATE_WORDS }};
```

That is enough to finally get SAW to verify this very stripped-down version of `quarter_round_spec`. Some good progress! But we aren't done yet, as we don't yet say what happens to the value that `state` points to after the function returns. This will be a requirement if we want to use `quarter_round_spec` in compositional verification (and we do want this), so we should address this shortly.

Recall that unlike the Rust `quarter_round` function, the Cryptol `quarterround` function doesn't have a `state` argument. This is because the Rust function does slightly more than what the Cryptol function does. The Rust function will look up elements of the `state` array, use them to perform the computations that the Cryptol function does, and then insert the new values back into the `state` array. To put it another way: the Rust function can be thought of as a wrapper around the Cryptol function that also performs an in-place bulk update of the `state` array.

In Cryptol, one can look up elements of an array using the `(@@)` function, and one can perform in-place array updates using the `updates` function. This translates into a postcondition that looks like this:

```
  let indices = {{ [a, b, c, d] }};
  let state_arr' = {{ updates state_arr indices (quarterround (state_
↪arr @@ indices)) }};
  mir_points_to state_ref (mir_term state_arr');
```

What does SAW think of this? Someone surprisingly, SAW finds a counterexample:

```
[01:43:30.065] Verifying salsa20/10e438b3::core#1[0]::quarter_round[0] .
↪..
[01:43:30.078] Simulating salsa20/10e438b3::core#1[0]::quarter_round[0]␣
↪...
[01:43:30.084] Checking proof obligations salsa20/10e438b3::core
↪#1[0]::quarter_round[0] ...
[01:43:30.801] Subgoal failed: salsa20/10e438b3::core#1[0]::quarter_
↪round[0] Literal equality postcondition

[01:43:30.801] SolverStats {solverStatsSolvers = fromList ["SBV->Z3"],␣
↪solverStatsGoalSize = 1999}
[01:43:30.802] ----------Counterexample----------
[01:43:30.802]   a: 13
[01:43:30.802]   b: 3
[01:43:30.802]   c: 0
[01:43:30.802]   d: 0
[01:43:30.802]   state: [3788509705, 0, 0, 3223325776, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 1074561051, 0, 0]
```

Note that in this counterexample, the values of `c` and `d` are the same. In the Rust version of the function, each element in `state` is updated sequentially, so if two of the array indices are the same, then the value that was updated with the first index will later be overwritten by the value at the later index. In the Cryptol version of the function, however, all of the positions in the array are updated simultaneously. This implicitly assumes that all of the array indices are disjoint from each other, an assumption that we are not encoding into `quarter_round_spec`'s preconditions.

At this point, it can be helpful to observe *how* the `quarter_round` function is used in practice. The call sites are found in the `rounds` function:

```
            // column rounds
            quarter_round(0, 4, 8, 12, state);
            quarter_round(5, 9, 13, 1, state);
            quarter_round(10, 14, 2, 6, state);
            quarter_round(15, 3, 7, 11, state);

            // diagonal rounds
            quarter_round(0, 1, 2, 3, state);
            quarter_round(5, 6, 7, 4, state);
            quarter_round(10, 11, 8, 9, state);
            quarter_round(15, 12, 13, 14, state);
```

Here, we can see that the values of `a`/`b`/`c`/`d` will only ever be chosen from a set of eight possible options. We can take advantage of this fact to constrain the possible set of values for `a`/`b`/`c`/`d`. The latest iteration of the `quarter_round_spec` is now:

```
let quarter_round_spec = do {
  a <- mir_fresh_var "a" mir_usize;
  b <- mir_fresh_var "b" mir_usize;
  c <- mir_fresh_var "c" mir_usize;
  d <- mir_fresh_var "d" mir_usize;
  let indices = {{ [a, b, c, d] }};
  mir_precond {{    indices == [0, 1, 2, 3]
                 \/ indices == [5, 6, 7, 4]
                 \/ indices == [10, 11, 8, 9]
                 \/ indices == [15, 12, 13, 14]
                 \/ indices == [0, 4, 8, 12]
                 \/ indices == [5, 9, 13, 1]
                 \/ indices == [10, 14, 2, 6]
                 \/ indices == [15, 3, 7, 11]
               }};
  state_ref <- mir_alloc_mut (mir_array STATE_WORDS mir_u32);
  state_arr <- mir_fresh_var "state" (mir_array STATE_WORDS mir_u32);
  mir_points_to state_ref (mir_term state_arr);
```

Note that:

- The `indices` value is constrained (via a precondition) to be one of the set of values that is chosen in the `rounds` function. (Note that `\/` is the logical-or function in Cryptol.) Each of these are concrete values that are less than `STATE_WORDS` (`16`), so we no longer need a precondition stating `a < STATE_WORDS /\ ...`.
- Because we now reference `indices` in the preconditions, we have moved its definition up. (Previously, it was defined in the postconditions section.)

With this in place, will SAW verify `quarter_round_spec` now?

```
[02:14:02.037] Verifying salsa20/10e438b3::core#1[0]::quarter_round[0] .
↪..
[02:14:02.051] Simulating salsa20/10e438b3::core#1[0]::quarter_round[0]␣
↪...
[02:14:02.057] Checking proof obligations salsa20/10e438b3::core
↪#1[0]::quarter_round[0] ...
[02:14:18.616] Proof succeeded! salsa20/10e438b3::core#1[0]::quarter_
↪round[0]
```

At long last, it succeeds. Hooray! SAW does have to think for a while, however, as this proof takes about 17 seconds to complete. It would be unfortunate to have to wait 17 seconds on every subsequent invocation of SAW, and since we still have other functions to verify, this is a very real possibility. For this reason, it can be helpful to temporarily turn this use of `mir_verify` into a `mir_unsafe_assume_spec`:

```
quarter_round_ov <-
  mir_unsafe_assume_spec m "salsa20::core#1::quarter_round" quarter_
↪round_spec;
  // Temporarily commented out to save time:
  //
  // mir_verify m "salsa20::core#1::quarter_round" [] false quarter_
↪round_spec z3;
```

Once we are done with the entire proof, we can come back and remove this use of `mir_unsafe_assume_spec`, as we're only using it as a time-saving measure.

9.8 Verifying the `rounds` function

Now that we've warmed up, let's try verifying the `rounds` function, which is where `quarter_round` is invoked. Here is the full definition of `rounds`:

```
    fn rounds(&mut self, state: &mut [u32; STATE_WORDS]) {
        for _ in 0..(R::COUNT / 2) {
            // column rounds
            quarter_round(0, 4, 8, 12, state);
            quarter_round(5, 9, 13, 1, state);
            quarter_round(10, 14, 2, 6, state);
            quarter_round(15, 3, 7, 11, state);

            // diagonal rounds
            quarter_round(0, 1, 2, 3, state);
            quarter_round(5, 6, 7, 4, state);
            quarter_round(10, 11, 8, 9, state);
            quarter_round(15, 12, 13, 14, state);
```

(continues on next page)

(continued from previous page)

```
        }

        for (s1, s0) in state.iter_mut().zip(&self.state) {
            *s1 = s1.wrapping_add(*s0);
        }
    }
```

First, `rounds` performs `COUNT` rounds on the `state` argument. After this, it takes each element of `self.state` and adds it to the corresponding element in `state`.

Linking back at the Salsa20 spec, we can see that the `rounds` function is *almost* an implementation of the Salsa20(x) hash function. The only notable difference is that while the Salsa20(x) hash function converts the results to little-endian form, the `rounds` function does not. `Salsa20.cry` implements this part of the spec here:

```
Salsa20 : [32] -> [64][8] -> [64][8]
Salsa20 count xs = littleendian_state_inverse (Salsa20_rounds count xw␣
↪xw)
  where
    xw = littleendian_state xs

Salsa20_rounds : [32] -> [16][32] -> [16][32] -> [16][32]
Salsa20_rounds count xw xw' = xw + zs@(count/2)
  where
    zs = [xw'] # [ doubleround zi | zi <- zs ]
```

Where `Salsa20` is the hash function, and `Salsa20_rounds` is the part of the hash function that excludes the little-endian conversions. In other words, `Salsa20_rounds` precisely captures the behavior of the Rust `rounds` function.

One aspect of the `rounds` function that will make verifying it slightly different from verifying `quarter_rounds` is that `rounds` is defined in an `impl` block for the `Core` struct. This means that the `&mut self` argument in `rounds` has the type `&mut Core<R>`. As such, we will have to look up the `Core` ADT in SAW using `mir_find_adt`.

This raises another question, however: when looking up `Core<R>`, what type should we use to instantiate `R`? As noted above, our choices are `R8`, `R12`, and `R20`, depending on how many rounds you want. For now, we'll simply hard-code it so that `R` is instantiated to be `R8`, but we will generalize this a bit later.

Alright, enough chatter—time to start writing a proof. First, let's look up the `R8` ADT. This is defined in the `salsa20` crate in the `rounds.rs` file, so its identifier becomes `salsa20::rounds::R8`:

```
let r_adt = mir_find_adt m "salsa20::rounds::R8" [];
```

Next, we need to look up the `PhantomData<R8>` ADT, which is used in the `rounds` field of the

`Core<R8>` struct. This is defined in `core::marker`:

```
let phantom_data_adt = mir_find_adt m "core::marker::PhantomData" [mir_
↪adt r_adt];
```

Finally, we must look up `Core<R8>` itself. Like `quarter_round`, the `Core` struct is defined in `salsa20::core#1`:

```
let core_adt = mir_find_adt m "salsa20::core#1::Core" [mir_adt r_adt];
```

Now that we have the necessary prerequisites, let's write a spec for the `rounds` function. First, we need to allocate a reference for the `self` argument:

```
let rounds_spec = do {
  self_ref <- mir_alloc_mut (mir_adt core_adt);
```

Next, we need to create symbolic values for the fields of the `Core` struct, which `self_ref` will point to. The `self.state` field will be a fresh array, and the `self.rounds` field will be a simple, empty struct value:

```
  self_state <- mir_fresh_var "self_state" (mir_array STATE_WORDS mir_
↪u32);
  let self_rounds = mir_struct_value phantom_data_adt [];
```

Finally, putting all of the `self` values together:

```
  let self_val = mir_struct_value core_adt [mir_term self_state, self_
↪rounds];
  mir_points_to self_ref self_val;
```

Next, we need a `state` argument (not to be confused with the `self.state` field in `Core`). This is handled much the same as it was in `quarter_round_spec`:

```
  state_ref <- mir_alloc_mut (mir_array STATE_WORDS mir_u32);
  state_arr <- mir_fresh_var "state" (mir_array STATE_WORDS mir_u32);
  mir_points_to state_ref (mir_term state_arr);
```

Lastly, we cap it off with a call to `mir_execute_func`:

```
  mir_execute_func [self_ref, state_ref];
};
```

(Again, we're missing some postconditions describing what `self_ref` and `state_ref` point to after the function returns, but we'll return to that in a bit.)

If we run SAW at this point, we see that everything in `rounds_spec` typechecks, so we're off to a good start. Let's keep going and add a `mir_verify` call.

Here, we are faced with an interesting question: what is the identifier for `rounds::<R8>`? The `rounds` function is defined using generics, so we can't verify it directly—we must instead verify a particular instantiation of `rounds`. At present, there isn't a good way to look up what the identifiers for instantiations of generic functions are (there will be in the future (https://github.com/GaloisInc/saw-script/issues/1980)), but it turns out that the identifier for `rounds::<R8>` is this:

```
rounds_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::rounds::_inst6e4a2d7250998ef7
↪" [quarter_round_ov] false rounds_spec z3;
```

Note that we are using `quarter_round_ov` as a compositional override. Once again, SAW is happy with our work thus far:

```
[03:12:35.990] Proof succeeded! salsa20/10e438b3::core#1[0]::{impl#0}
↪[0]::rounds[0]::_inst6e4a2d7250998ef7[0]
```

Nice. Now let's go back and fill in the missing postconditions in `rounds_spec`. In particular, we must declare what happens to both `self_ref` and `state_ref`. A closer examination of the code in the Rust `rounds` function reveals that the `self` argument is never modified at all, so that part is easy:

```
  mir_points_to self_ref self_val;
```

The `state` argument, on the other hand, is modified in-place. This time, our job is made easier by the fact that `Salsa20_rounds` implements *exactly* what we need. Because we are instantiating `rounds` at type `R8`, we must explicitly state that we are using 8 rounds:

```
  mir_points_to state_ref (mir_term {{ Salsa20_rounds 8 self_state␣
↪state_arr }});
```

Once again, SAW is happy with our work. We're on a roll!

Now let's address the fact that we are hard-coding everything to `R8`, which is somewhat uncomfortable. We can make things better by allowing the user to specify the number of rounds. The first thing that we will need to change is the `r_adt` definition, which is response for looking up `R8`. We want to turn this into a function that, depending on the user input, will look up `R8`, `R12`, or `R20`:

```
let r_adt num_rounds = mir_find_adt m (str_concat "salsa20::rounds::R"␣
↪(show num_rounds)) [];
```

Where `str_concat` is a SAW function for concatenating strings together:

```
sawscript> :type str_concat
String -> String -> String
```

We also want to parameterize `phantom_data_adt` and `core_adt`:

```
let phantom_data_adt r = mir_find_adt m "core::marker::PhantomData"␣
↪[mir_adt r];
let core_adt r = mir_find_adt m "salsa20::core#1::Core" [mir_adt r];
```

Next, we need to parameterize `rounds_spec` by the number of rounds. This will require changes in both the preconditions and postconditions. On the preconditions side, we must pass the number of rounds to the relevant functions:

```
let rounds_spec num_rounds = do {
  let r = r_adt num_rounds;
  let core_adt_inst = core_adt r;
  self_ref <- mir_alloc_mut (mir_adt core_adt_inst);
  self_state <- mir_fresh_var "self_state" (mir_array STATE_WORDS mir_
↪u32);
  let self_rounds = mir_struct_value (phantom_data_adt r) [];
  let self_val = mir_struct_value core_adt_inst [mir_term self_state,␣
↪self_rounds];
```

And on the postconditions side, we must pass the number of rounds to the Cryptol `Salsa20_rounds` function:

```
  mir_points_to state_ref (mir_term {{ Salsa20_rounds `num_rounds self_
↪state state_arr }});
};
```

Finally, we must adjust the call to `rounds_spec` in the context of `mir_verify` so that we pick `8` as the number of rounds:

```
rounds_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::rounds::_inst6e4a2d7250998ef7
↪" [quarter_round_ov] false (rounds_spec 8) z3;
```

SAW is happy with this generalization. To demonstrate that we have generalized things correctly, we can also verify the same function at `R20` instead of `R8`:

```
rounds_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::rounds::_instfa33e77d840484a0
↪" [quarter_round_ov] false (rounds_spec 20) z3;
```

The only things that we had to change were the identifier and the argument to `rounds_spec`. Not bad!

9.9 Verifying the `counter_setup` function

We're very nearly at the point of being able to verify `apply_keystream`. Before we do, however, there is one more function that `apply_keystream` calls, which we ought to verify first: `counter_setup`. Thankfully, the implementation of `counter_setup` is short and sweet:

```
    pub(crate) fn counter_setup(&mut self, counter: u64) {
        self.state[8] = (counter & 0xffff_ffff) as u32;
        self.state[9] = ((counter >> 32) & 0xffff_ffff) as u32;
    }
```

This updates the elements of the `state` array at indices `8` and `9` with the lower 32 bits and the upper 32 bits of the `counter` argument, respecitvely. At a first glance, there doesn't appear to be any function in `Salsa20.cry` that directly corresponds to what `counter_setup` does. This is a bit of a head-scratcher, but the answer to this mystery will become more apparent as we get further along in the proof.

For now, we should take matters into our own hands and write our own Cryptol spec for `counter_setup`. To do this, we will create a new Cryptol file named `Salsa20Extras.cry`, which imports from `Salsa20.cry`:

```
module Salsa20Extras where

import Salsa20
```

The Cryptol implementation of `counter_setup` will need arrays of length `STATE_WORDS`, so we shall define `STATE_WORDS` first:

```
type STATE_WORDS = 16
```

Note that we preceded this definition with the `type` keyword. In Cryptol, sequence lengths are encoded at the type level, so if we want to use `STATE_WORDS` at the type level, we must declare it as a `type`.

Finally, we can write a Cryptol version of `counter_setup` using our old friend `updates` to perform a bulk sequence update:

```
counter_setup : [STATE_WORDS][32] -> [64] -> [STATE_WORDS][32]
counter_setup state counter =
  updates state [8, 9] [drop counter, drop (counter >> 32)]
```

Note that `counter` is a 64-bit word, but the elements of the `state` sequence are 32-bit words. As a result, we have to explicitly truncate `counter` and `counter >> 32` to 32-bit words by using the `drop` function, which drops the first 32 bits from each word.

Returning to `salsa20.saw`, we must now make use of our new Cryptol file by `import`ing it at the top:

```
import "Salsa20Extras.cry";
```

With the `counter_setup` Cryptol implementation in scope, we can now write a spec for the Rust `counter_setup` function. There's not too much to remark on here, as the spec proves relatively straightforward to write:

```
let counter_setup_spec num_rounds = do {
  let r = r_adt num_rounds;
  let core_adt_inst = core_adt r;
  self_ref <- mir_alloc_mut (mir_adt core_adt_inst);
  self_state <- mir_fresh_var "self_state" (mir_array STATE_WORDS mir_
↪u32);
  let self_rounds = mir_struct_value (phantom_data_adt r) [];
  let self_val = mir_struct_value core_adt_inst [mir_term self_state,␣
↪self_rounds];
  mir_points_to self_ref self_val;

  counter <- mir_fresh_var "counter" mir_u64;

  mir_execute_func [self_ref, mir_term counter];

  let self_state' = {{ counter_setup self_state counter }};
  let self_val' = mir_struct_value core_adt_inst [mir_term self_state',␣
↪self_rounds];
  mir_points_to self_ref self_val';
};
```

We can now verify `counter_setup` against `counter_setup_spec` at lengths `8` and `20`:

```
counter_setup_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::counter_setup::_
↪inst6e4a2d7250998ef7" [] false (counter_setup_spec 8) z3;
counter_setup_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::counter_setup::_
↪instfa33e77d840484a0" [] false (counter_setup_spec 20) z3;
```

That wasn't so bad. It's a bit unsatisfying that we had to resort to writing a Cryptol function not found in `Salsa20.cry`, but go along with this for now—it will become apparent later why this needed to be done.

9.10 Verifying the `apply_keystream` function (first attempt)

It's time. Now that we've verified `rounds` and `counter_setup`, it's time to tackle the topmost function in the call stack: `apply_keystream`:

```
    pub fn apply_keystream(&mut self, counter: u64, output: &mut [u8]) {
        debug_assert_eq!(output.len(), BLOCK_SIZE);
        self.counter_setup(counter);

        let mut state = self.state;
        self.rounds(&mut state);

        for (i, chunk) in output.chunks_mut(4).enumerate() {
            for (a, b) in chunk.iter_mut().zip(&state[i].to_le_bytes())
↪{
                *a ^= *b;
            }
        }
    }
```

There aren't *that* many lines of code in this function, but there is still quite a bit going on. Let's walk through `apply_keystream` in more detail:

1. The `output` argument represents the message to encrypt (or decrypt). `output` is a slice of bytes, so in principle, `output` can have an arbitrary length. That being said, the first line of `apply_keystream`'s implementation checks that `output`'s length is equal to `BLOCK_SIZE`:

   ```
           debug_assert_eq!(output.len(), BLOCK_SIZE);
   ```

   Where `BLOCK_SIZE` is defined here:

   ```
   /// Size of a Salsa20 block in bytes
   pub const BLOCK_SIZE: usize = 64;
   ```

   So in practice, `output` must have exactly 64 elements.

2. Next, `apply_keystream` invokes the `counter_setup` and `rounds` functions to set up the keystream (the local `state` variable).

3. Finally, `apply_keystream` combines the keystream with `output`. It does so by chunking `output` into a sequence of 4 bytes, and then it XOR's the value of each byte in-place with the corresponding byte from the keystream. This performs little-endian conversions as necessary.

The fact that we are XOR'ing bytes strongly suggests that this is an implementation of the Salsa20 encryption function from the spec. There is an important difference between how the Salsa20 spec defines the encryption function versus how `apply_keystream` defines it, however. In the Salsa20 spec, encryption is a function of a key, nonce, and a message. `apply_keystream`, on the other

hand, is a function of `self`'s internal state, a counter, and a message. The two aren't *quite* the same, which is makes it somewhat tricky to describe one in terms of the other.

`Salsa20.cry` defines a straightforward Cryptol port of the Salsa20 encryption function from the spec, named `Salsa20_encrypt`. Because it takes a key and a nonce as an argument, it's not immediately clear how we'd tie this back to `apply_keystream`. But no matter: we can do what we did before and define our own Cryptol version of `apply_keystream` in `Salsa20Extras.cry`:

```
apply_keystream : [32] -> [STATE_WORDS][32] -> [64] -> [BLOCK_SIZE][8] -
↪> [BLOCK_SIZE][8]
apply_keystream count state0 counter output =
    output ^ littleendian_state_inverse state2
  where
    state1 = counter_setup state0 counter
    state2 = Salsa20_rounds count state1 state1
```

This implementation builds on top of the Cryptol `counter_setup` and `Salsa20_rounds` functions, which we used as the reference implementations for the Rust `counter_setup` and `rounds` functions, respectively. We also make sure to define a `BLOCK_SIZE` type alias at the top of the file:

```
type BLOCK_SIZE = 64
```

Now let's write a SAW spec for `apply_keystream`. Once again, we will need to reference `BLOCK_SIZE` when talking about the `output`-related parts of the spec, so make sure to define `BLOCK_SIZE` at the top of the `.saw` file:

```
let BLOCK_SIZE = 64;
```

First, we need to declare all of our arguments, which proceeds as you would expect:

```
let apply_keystream_spec num_rounds = do {
  let r = r_adt num_rounds;
  let core_adt_inst = core_adt r;
  self_ref <- mir_alloc_mut (mir_adt core_adt_inst);
  self_state <- mir_fresh_var "self_state" (mir_array STATE_WORDS mir_
↪u32);
  let self_rounds = mir_struct_value (phantom_data_adt r) [];
  let self_val = mir_struct_value core_adt_inst [mir_term self_state,␣
↪self_rounds];
  mir_points_to self_ref self_val;

  counter <- mir_fresh_var "counter" mir_u64;

  output_arr <- mir_fresh_var "output_arr" (mir_array BLOCK_SIZE mir_
↪u8);
```

(continues on next page)

(continued from previous page)

```
  output_ref <- mir_alloc_mut (mir_array BLOCK_SIZE mir_u8);
  mir_points_to output_ref (mir_term output_arr);
  let output = mir_slice_value output_ref;

  mir_execute_func [self_ref, mir_term counter, output];
```

What about the postconditions? We have two mutable references to contend with: `self_ref` and `output_ref`. The postcondition for `self_ref` is fairly straightforward: the only time it is ever modified is when `counter_setup` is called. This means that after the `apply_keystream` function has returned, `self_ref` will point to the results of calling the `counter_setup` Cryptol function:

```
  let self_state' = {{ counter_setup self_state counter }};
  let self_val' = mir_struct_value core_adt_inst [mir_term self_state',␣
↪self_rounds];
  mir_points_to self_ref self_val';
```

`output_ref` is where the interesting work happenings. After the Rust `apply_keystream` function has returned, it will point to the results of calling the Cryptol `apply_keystream` function that we just defined:

```
  mir_points_to output_ref (mir_term {{ apply_keystream `num_rounds␣
↪self_state counter output_arr }});
};
```

Finally, we can put this all together and verify `apply_keystream` against `apply_keystream_spec` at lengths `8` and `20`:

```
apply_keystream_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪inst6e4a2d7250998ef7" [counter_setup_8_spec_ov, rounds_8_spec_ov]␣
↪false (apply_keystream_spec 8) z3;
apply_keystream_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪instfa33e77d840484a0" [counter_setup_20_spec_ov, rounds_20_spec_ov]␣
↪false (apply_keystream_spec 20) z3;
```

SAW will successfully verify these. We've achieved victory… or have we? Recall that we had to tailor the Cryptol `apply_keystream` function to specifically match the behavior of the corresponding Rust code. This makes the proof somewhat underwhelming, since the low-level implementation is nearly identical to the high-level spec.

A more impressive proof would require linking `apply_keystream` to a Cryptol function in the `Salsa20.cry` file, which was developed independently of the Rust code. As we mentioned before, however, doing so will force us to reconcile the differences in the sorts of arguments that each function takes, as `apply_keystream` doesn't take a key or nonce argument. Time to think for a bit.

9.11 Verifying the `new_raw` function

At this point, we should ask ourselves: *why* doesn't `apply_keystream` take a key or nonce argument? The reason lies in the fact that the `salsa20` crate implements Salsa20 in a stateful way. Specifically, the `Core` struct stores internal state that is used to compute the keystream to apply when hashing. In order to use this internal state, however, we must first initialize it. The `new` function that is responsible for this initialization:

```
    /// Initialize core function with the given key and IV
    pub fn new(key: &Key, iv: &Nonce) -> Self {
        Self::new_raw(key.as_ref(), iv.as_ref())
    }
```

Sure enough, this function takes a key and a nonce as an argument! This is a critical point that we overlooked. When using the `salsa20` crate, you wouldn't use the `apply_keystream` function in isolation. Instead, you would create an initial `Core` value using `new`, and *then* you would invoke `apply_keystream`. The Salsa20 spec effectively combines both of these operations in is encryption function, whereas the `salsa20` splits these two operations into separate functions altogether.

Strictly speaking, we don't need to verify `new` in order to verify `apply_keystream`, as the latter never invokes the former. Still, it will be a useful exercise to verify `new`, as the insights we gain when doing so will help us write a better version of `apply_keystream_spec`.

All that being said, we probably to verify `new_raw` (a lower-level helper function) rather than `new` itself. This is because the definitions of `Key` and `Nonce` are somewhat involved. For instance, `Key` is defined as:

```
pub type Key = cipher::generic_array::GenericArray<u8, U32>;
```

`GenericArray` (https://docs.rs/generic-array/latest/generic_array/struct.GenericArray.html) is a somewhat complicated abstraction. Luckily, we don't really *need* to deal with it, since `new_raw` deals with simple array references rather than `GenericArray`s:

```
    /// Initialize core function with the given key and IV
    pub fn new_raw(key: &[u8; 32], iv: &[u8; 8]) -> Self {
```

The full implementation of `new_raw` is rather long, so we won't inline the whole thing here. At a high level, it initializes the `state` array of a `Core` value by populating each element of the array with various things. Some elements of the array are populated with `key`, some parts are populated with `iv` (i.e., the nonce), and other parts are populated with an array named `CONSTANTS`:

```
/// State initialization constant ("expand 32-byte k")
const CONSTANTS: [u32; 4] = [0x6170_7865, 0x3320_646e, 0x7962_2d32,␣
↪0x6b20_6574];
```

The comment about `"expand 32-byte k"` is a strong hint that `new_raw` is implementing a portion of the Salsa20 expansion function from the spec. (No really, the spec literally says to use the exact

string `"expand 32-byte k"`—look it up!) The `Salsa20.cry` Cryptol file has an implementation of this portion of the expansion function, which is named `Salsa20_init`:

```
Salsa20_init : {a} (a >= 1, 2 >= a) => ([16*a][8], [16][8]) -> [64][8]
Salsa20_init(k, n) = x
  where
    [s0, s1, s2, s3] = split "expand 32-byte k" : [4][4][8]
    [t0, t1, t2, t3] = split "expand 16-byte k" : [4][4][8]
    x = if(`a == 2) then s0 # k0 # s1 # n # s2 # k1 # s3
                    else t0 # k0 # t1 # n # t2 # k0 # t3
    [k0, k1] = (split(k#zero)):[2][16][8]
```

Note that we were careful to say a *portion* of the Salsa20 expansion function. There is also a Cryptol implementation of the full expansion function, named `Salsa20_expansion`:

```
Salsa20_expansion : {a} (a >= 1, 2 >= a) => ([32], [16*a][8], [16][8]) -
↪> [64][8]
Salsa20_expansion(count, k, n) = Salsa20 count (Salsa20_init(k, n))
```

This calls `Salsa20_init` followed by `Salsa20`, the latter of which performs hashing. Importantly, `new_raw` does *not* do any hashing on its own, just initialization. For this reason, we want to use `Salsa20_init` as the reference implementation of `new_raw`, not `Salsa20_expansion`.

Alright, time to write a SAW spec. The first part of the spec is straightforward:

```
let new_raw_spec num_rounds = do {
  key_ref <- mir_alloc (mir_array 32 mir_u8);
  key_arr <- mir_fresh_var "key_arr" (mir_array 32 mir_u8);
  mir_points_to key_ref (mir_term key_arr);

  nonce_ref <- mir_alloc (mir_array 8 mir_u8);
  nonce_arr <- mir_fresh_var "nonce" (mir_array 8 mir_u8);
  mir_points_to nonce_ref (mir_term nonce_arr);

  mir_execute_func [key_ref, nonce_ref];
```

As is usually the case, the postconditions are the tricky part. We know that the behavior of `new_raw` will roughly coincide with the `Salsa20_init` function, so let's try that first:

```
  let r = r_adt num_rounds;
  let self_state = {{ Salsa20_init(key_arr, nonce_arr) }};
  let self_rounds = mir_struct_value (phantom_data_adt r) [];
  let self_val = mir_struct_value (core_adt r) [mir_term self_state,␣
↪self_rounds];
  mir_return self_val;
```

If we attempt to verify this using `mir_verify`:

```
new_raw_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::new_raw::_
↪inst6e4a2d7250998ef7" [] false (new_raw_spec 8) z3;
new_raw_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::new_raw::_
↪instfa33e77d840484a0" [] false (new_raw_spec 20) z3;
```

SAW complains thusly:

```
Cryptol error:
[error] at salsa20.saw:109:45--109:54:
  Type mismatch:
    Expected type: 16
    Inferred type: 8
    Context: [ERROR] _
    When checking type of 2nd tuple field
```

Here, the 2nd tuple field is the `nonce_arr` in `Salsa20_init(key_arr, nonce_arr)`. And sure enough, `Salsa20_init` expects the 2nd tuple field to be a sequence of 16 elements, but `nonce_arr` only has 8 elements. Where do we get the remaining 8 elements from?

The answer to this question can be found by looking at the implementation of `new_raw` more closely. Let's start at this code:

```
        for (i, chunk) in iv.chunks(4).enumerate() {
            state[6 + i] = u32::from_le_bytes(chunk.try_into().
↪unwrap());
```

This will chunk up `iv` (the nonce) into two 4-byte chunks and copies them over to the elements of `state` array at indices `6` and `7`. This is immediately followed by two updates at indices `8` and `9`, which are updated to be `0`:

```
        state[8] = 0;
        state[9] = 0;
```

If you take the two 4-bytes chunks of `iv` and put two 4-byte `0` values after them, then you would have a total of 16 bytes. This suggests that the nonce value that `Salsa20_init` expects is actually this:

```
nonce_arr # zero : [16][8]
```

Where `zero : [8][8]` is a Cryptol expression that returns all zeroes, and `(#)` is the Cryptol operator for concatenating two sequences together. Let's update `new_raw_spec` to reflect this:

```
  let self_state = {{ Salsa20_init(key_arr, nonce_arr # zero) }};
```

This is closer to what we want, but not quite. SAW still complains:

```
could not match specified value with actual value:
  ...
  type of actual value:    [u32; 16]
  type of specified value: [u8; 64]
```

This is because `Salsa20_init` returns something of type `[64][8]`, which corresponds to the Rust type `[u8; 64]`. `self.state`, on the other hand, is of type `[u32; 16]`. These types are very close, as they both contain the same number of bytes, but they are chunked up differently. Recall the code that copies the nonce value over to `self.state`:

```
        for (i, chunk) in iv.chunks(4).enumerate() {
            state[6 + i] = u32::from_le_bytes(chunk.try_into().
↪unwrap());
```

In order to resolve the type differences between `iv` and `state`, this code needed to explicitly convert `iv` to little-endian form using the `u32::from_le_bytes` (https://doc.rust-lang.org/std/primitive.u32.html#method.from_le_bytes) function. There is a similar Cryptol function in `Salsa20.cry` named `littleendian_state`:

```
littleendian_state : [64][8] -> [16][32]
littleendian_state b = [littleendian xi | xi <- split b]
```

Note that `[64][8]` is the Cryptol equivalent of `[u8; 64]`, and `[16][32]` is the Cryptol equivalent of `[u32; 16]`. As such, this is exactly the function that we need to resolve the differences in types:

```
  let self_state = {{ littleendian_state (Salsa20_init(key_arr, nonce_
↪arr # zero)) }};
```

With that change, SAW is finally happy with `new_raw_spec` and successfully verifies it.

There is an interesting connection between the `new_raw` and `counter_setup` functions. Both functions perform in-place updates on `state` at indices `8` and `9`. Whereas `new_raw` always sets these elements of `state` to `0`, `counter_setup` will set them to the bits of the `counter` argument (after converting `counter` to little-endian form). This means that if you invoke `counter_setup` right after `new_raw`, then `counter_setup` would overwrite the `0` values with the `counter` argument. In order words, it would be tantamount to initializing `state` like so:

```
littleendian_state (Salsa20_init(key, nonce # littleendian_inverse␣
↪counter))
```

Where `littleendian_inverse` (a sibling of `littleendian_state`) converts a `[64]` value to a `[8][8]` one. This pattern is a curious one…

9.12 Verifying the `apply_keystream` function (second attempt)

Let's now return to the problem of linking `apply_keystream` up to `Salsa20_encrypt`. In particular, let's take a closer look at the definition of `Salsa20_encrypt` itself:

```
Salsa20_encrypt : {a, l} (a >= 1, 2 >= a, l <= 2^^70) => ([32],␣
↪[16*a][8], [8][8], [l][8]) -> [l][8]
Salsa20_encrypt(count, k, v, m) = c
  where
    salsa = take (join [ Salsa20_expansion(count, k, v#(littleendian_
↪inverse i)) | i <- [0, 1 ... ] ])
```

Does anything about this definition strike you as interesting? Take a look at the `v#(littleendian_inverse i)` part—we *just* saw a use of `littleendian_inverse` earlier in our discussion about initializing the `state`! Moreover, `v` is the nonce argument, so it is becoming clearer that `Sals20_encrypt` is creating an initial state is much the same way that `new_raw` is.

A related question: what is the `i` value? The answer is somewhat technical: the Salsa20 encryption function is designed to work with messages with differing numbers of bytes (up to `2^^70` bytes, to be exact). Each 8-byte chunk in the message will be encrypted with a slightly difference nonce. For instance, the first 8-byte chunk's nonce will have its lower 32 bits set to `0`, the second 8-byte chunk's nonce will have its lower 32 bits set to `1`, and so on. In general, the `i`th 8-byte chunk's nonce will have its lower 32 bits set to `i`, and this corresponds exactly to the `i` in the expression `littleendian_inverse i`.

Note, however, that `apply_keystream` only ever uses a message that consists of exactly eight 8-byte chunks. This means that `Salsa20_encrypt` will only ever invoke `Salsa20_expansion` once with a nonce value where the lower 32 bits are set to `0`. That is, it will perform encryption with an initial state derived from:

```
Salsa20_init(k, v#(littleendian_inverse zero))
```

Which can be further simplified to `Salsa20_init(k, v # zero)`. This is very nearly what we want, as this gives us the behavior of the Rust `new_raw` function. There's just one problem though: it doesn't take the behavior of `counter_setup` into account. How do we go from `zero` to `littleendian_inverse counter`?

While `Salsa20_encrypt` doesn't take counters into account at all, it is not too difficult to generalize `Salsa20_encrypt` in this way. There is a variant of `Salsa20_encrypt` in the same file named `Salsa20_encrypt_with_offset`, where the offset argument `o` serves the same role that `counter` does in `counter_setup`:

```
Salsa20_encrypt_with_offset : {a, l} (a >= 1, 2 >= a, l <= 2^^70) =>
    ([32], [16*a][8], [8][8], [64], [l][8]) -> [l][8]
Salsa20_encrypt_with_offset(count, k, v, o, m) = c
  where
```

(continues on next page)

(continued from previous page)

```
    salsa = take (join [ Salsa20_expansion(count, k, v#(littleendian_
↪inverse (o + i))) | i <- [0, 1 ... ] ])
    c = m ^ salsa
```

(Observe that `Salsa20_encrypt(count, k, v, m)` is equivalent to `Salsa20_encrypt_with_offset(count, k, v, 0, m)`.)

At long last, we have discovered the connection between `apply_keystream` and the Salsa20 spec. If you assume that you invoke `new_raw` beforehand, then the behavior of `apply_keystream` corresponds exactly to that of `Salsa20_encrypt_with_offset`. This insight will inform us how to write an alternative SAW spec for `apply_keystream`:

```
let apply_keystream_alt_spec num_rounds = do {
  key <- mir_fresh_var "key" (mir_array 32 mir_u8);
  nonce <- mir_fresh_var "nonce" (mir_array 8 mir_u8);
  counter <- mir_fresh_var "counter" mir_u64;

  let r = r_adt num_rounds;
  let core_adt_inst = core_adt r;
  self_ref <- mir_alloc_mut (mir_adt core_adt_inst);
  let self_state = {{ littleendian_state (Salsa20_init(key, nonce #␣
↪littleendian_inverse counter)) }};
  let self_rounds = mir_struct_value (phantom_data_adt r) [];
  let self_val = mir_struct_value core_adt_inst [mir_term self_state,␣
↪self_rounds];
  mir_points_to self_ref self_val;
```

Observe the following differences between `apply_keystream_alt_spec` and our earlier `apply_keystream_spec`:

1. In `apply_keystream_alt_spec`, we declare fresh `key` and `nonce` values, which weren't present at all in `apply_keystream_spec`.
2. In `apply_keystream_alt_spec`, we no longer make `self_state` a fresh, unconstrained value. Instead, we declare that it must be the result of calling `Salsa20_init` on the `key`, `nonce`, and `counter` values. This is the part that encodes the assumption that `new_raw` was invoked beforehand.

The parts of the spec relating to `output` remain unchanged:

```
  output_arr <- mir_fresh_var "output_arr" (mir_array BLOCK_SIZE mir_
↪u8);
  output_ref <- mir_alloc_mut (mir_array BLOCK_SIZE mir_u8);
  mir_points_to output_ref (mir_term output_arr);
  let output = mir_slice_value output_ref;
```

(continues on next page)

(continued from previous page)

```
  mir_execute_func [self_ref, mir_term counter, output];
```

The postconditions are slightly different in `apply_keystream_alt_spec`. While the parts relating to `self_ref` remain unchanged, we now have `output_ref` point to the results of calling `Salsa20_encrypt_with_offset`:

```
  let self_state' = {{ counter_setup self_state counter }};
  let self_val' = mir_struct_value core_adt_inst [mir_term self_state',␣
↪self_rounds];
  mir_points_to self_ref self_val';
  mir_points_to output_ref (mir_term {{ Salsa20_encrypt_with_
↪offset(`num_rounds, key, nonce, counter, output_arr) }});
```

Tying this all together, we call `mir_verify`, making sure to use compositional overrides involving `counter_setup` and `rounds`:

```
apply_keystream_alt_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪inst6e4a2d7250998ef7" [counter_setup_8_spec_ov, rounds_8_spec_ov]␣
↪false (apply_keystream_alt_spec 8) z3;
apply_keystream_alt_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪instfa33e77d840484a0" [counter_setup_20_spec_ov, rounds_20_spec_ov]␣
↪false (apply_keystream_alt_spec 20) z3;
```

At long last, it is time to run SAW on this. When we do, we see this:

```
[15:11:44.576] Checking proof obligations salsa20/10e438b3::core#1[0]::
↪{impl#0}[0]::apply_keystream[0]::_inst6e4a2d7250998ef7[0] ...
```

After this, SAW loops forever. Oh no! While somewhat disheartening, this is a reality of SMT-based verification that we must content with. SMT solvers are extremely powerful, but their performance can sometimes be unpredictable. The task of verifying `apply_keystream_alt_spec` is *just* complicated enough that Z3 cannot immediately figure out that the proof is valid, so it resorts to much slower algorithms to solve proof goals.

We could try waiting for Z3 to complete, but we'd be waiting for a long time. It's not unheard of for SMT solvers to take many hours on especially hard problems, but we don't have that many hours to spare. We should try a slightly different approach instead.

When confronted with an infinite loop in SAW, there isn't a one-size-fits-all solution that will cure the problem. Sometimes, it is worth stating your SAW spec in a slightly different way such that the SMT solver can spot patterns that it couldn't before. Other times, it can be useful to try and break the problem up into smaller functions and use compositional verification to handle the more complicated

subfunctions. As we mentioned before, the performance of SMT solvers in unpredictable, and it's not always obvious what the best solution is.

In this example, however, the problem lies with Z3 itself. As it turns out, Yices (a different SMT solver) *can* spot the patterns needed to prove `apply_keystream_alt_spec` immediately. Fortunately, SAW includes support for both Z3 and Yices. In order to switch from Z3 to Yices, swap out the `z3` proof script with `yices`:

```
apply_keystream_alt_8_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪inst6e4a2d7250998ef7" [counter_setup_8_spec_ov, rounds_8_spec_ov]␣
↪false (apply_keystream_alt_spec 8) yices;
apply_keystream_alt_20_spec_ov <-
  mir_verify m "salsa20::core#1::{impl#0}::apply_keystream::_
↪instfa33e77d840484a0" [counter_setup_20_spec_ov, rounds_20_spec_ov]␣
↪false (apply_keystream_alt_spec 20) yices;
```

After doing this, SAW is leverage Yices to solve the proof goals almost immediately:

```
[15:22:00.745] Proof succeeded! salsa20/10e438b3::core#1[0]::{impl#0}
↪[0]::apply_keystream[0]::_instfa33e77d840484a0[0]
```

And with that, we're finally done! You've successfully completed a non-trivial SAW exercise in writing some interesting proofs. Give yourself a well-deserved pat on the back.

The process of developing these proofs was bumpy at times, but that is to be expected. You very rarely get a proof correct on the very first try, and when SAW doesn't accept your proof, it is important to be able to figure out what went wrong and how to fix it. This is a skill that takes some time to grow, but with enough time and experience, you will be able to recognize common pitfalls. This case study showed off some of these pitfalls, but there are likely others.

10 A final word

Like everything else in the `saw-script` repo, this tutorial is being maintained and developed. If you see something in the tutorial that is wrong, misleading, or confusing, please file an issue about it here (https://github.com/GaloisInc/saw-script/issues).