|galois|

Using the Software Analysis Workbench (SAW)

Galois, Inc.
421 SW 6th Ave., Ste. 300
Portland, OR 97204

Contents

Overview . . . 2

Invoking SAW **3**

Structure of SAWScript **3**
Syntax . . . 4
Basic Types and Values . . . 4
Basic Expression Forms . . . 5
Other Basic Functions . . . 8

The Term Type **8**

Cryptol and its Role in SAW **9**

Transforming Term Values **12**
Rewriting . . . 12
Folding and Unfolding . . . 15
Other Built-in Transformation and Inspection Functions . . . 16
Loading and Storing Terms . . . 16

Proofs about Terms **17**
Automated Tactics . . . 17
Proof Script Diagnostics . . . 18
Rewriting in Proof Scripts . . . 18
Other Transformations . . . 18
Other External Provers . . . 19
Offline Provers . . . 19
Finishing Proofs without External Solvers . . . 20
Multiple Goals . . . 20
Proof Failure and Satisfying Assignments . . . 20
AIG Values and Proofs . . . 21

Symbolic Execution **21**

Symbolic Termination **22**

Loading Code **23**
Loading LLVM . . . 23
Loading Java . . . 24
Loading MIR . . . 24
Notes on Compiling Code for SAW . . . 24

Compiling LLVM . . . 24
Compiling Java . . . 25
Compiling MIR . . . 25
Notes on C++ Analysis . . . 26

Direct Extraction **26**

Creating Symbolic Variables **27**

Specification-Based Verification **28**
Running a Verification . . . 29
Structure of a Specification . . . 29
Creating Fresh Variables . . . 29
The SetupValue and JVMValue Types . . . 30
Executing . . . 31
Return Values . . . 31
A First Simple Example . . . 31
Compositional Verification . . . 31
Specifying Heap Layout . . . 32
Specifying Heap Values . . . 33
Working with Compound Types . . . 33
Bitfields . . . 34
Global variables . . . 35
Preconditions and Postconditions . . . 37
Assuming specifications . . . 37
A Heap-Based Example . . . 37
Using Ghost State . . . 39
An Extended Example . . . 39
Salsa20 Overview . . . 39
Specifications . . . 40
Verifying Everything . . . 43

Extraction to the Coq theorem prover **43**
Support Library . . . 44
Cryptol module extraction . . . 44
Proofs involving uninterpreted functions . . . 45
Translation limitations and caveats . . . 45
Recursive programs . . . 45
Type coercions . . . 46
Error terms . . . 46

Overview

The Software Analysis Workbench (SAW) is a tool for constructing mathematical models of the computational behavior of software, transforming these models, and proving properties about them.

SAW can currently construct models of a subset of programs written in Cryptol, LLVM (and therefore C), and JVM (and therefore Java). SAW also has experimental, incomplete support for MIR (and therefore Rust). The models take the form of typed functional programs, so in a sense SAW can be considered a translator from imperative programs to their functional equivalents. Various external proof tools, including a variety of SAT and SMT solvers, can be used to prove properties about the functional models. SAW can construct models from arbitrary Cryptol programs, and from C and Java programs that have fixed-size inputs and outputs and that terminate after a fixed number of iterations of any loop (or a fixed number of recursive calls).

One common use case is to verify that an algorithm specification in Cryptol is equivalent to an algorithm implementation in C or Java.

The process of extracting models from programs, manipulating them, forming queries about them, and sending them to external provers is orchestrated using a special purpose language called SAWScript. SAWScript is a typed functional language with support for sequencing of imperative commands.

The rest of this document first describes how to use the SAW tool, `saw`, and outlines the structure of the SAWScript language and its relationship to Cryptol. It then presents the SAWScript commands that transform functional models and prove properties about them. Finally, it describes the specific commands available for constructing models from imperative programs.

Invoking SAW

The primary mechanism for interacting with SAW is through the `saw` executable included as part of the standard binary distribution. With no arguments, `saw` starts a read-evaluate-print loop (REPL) that allows the user to interactively evaluate commands in the SAWScript language. With one file name argument, it executes the specified file as a SAWScript program.

In addition to a file name, the `saw` executable accepts several command-line options:

`-h, -?, --help` Print a help message.

`-V, --version` Show the version of the SAWScript interpreter.

`-c path, --classpath=path` Specify a colon-delimited list of paths to search for Java classes.

`-i path, --import-path=path` Specify a colon-delimited list of paths to search for imports.

`-t, --extra-type-checking` Perform extra type checking of intermediate values.

`-I, --interactive` Run interactively (with a REPL). This is the default if no other arguments are specified.

`-j path, --jars=path` Specify a colon-delimited list of paths to `.jar` files to search for Java classes.

`-b path, --java-bin-dirs` Specify a colon-delimited list of paths to search for a Java executable.

`-d num, --sim-verbose=num` Set the verbosity level of the Java and LLVM simulators.

`-v num, --verbose=num` Set the verbosity level of the SAWScript interpreter.

SAW also uses several environment variables for configuration:

`CRYPTOLPATH` Specify a colon-delimited list of directory paths to search for Cryptol imports (including the Cryptol prelude).

`PATH` If the `--java-bin-dirs` option is not set, then the `PATH` will be searched to find a Java executable.

`SAW_IMPORT_PATH` Specify a colon-delimited list of directory paths to search for imports.

`SAW_JDK_JAR` Specify the path of the `.jar` file containing the core Java libraries. Note that that is not necessary if the `--java-bin-dirs` option or the `PATH` environment variable is used, as SAW can use this information to determine the location of the core Java libraries' `.jar` file.

On Windows, semicolon-delimited lists are used instead of colon-delimited lists.

Structure of SAWScript

A SAWScript program consists, at the top level, of a sequence of commands to be executed in order. Each command is terminated with a semicolon. For example, the `print` command displays a textual representation

of its argument. Suppose the following text is stored in the file `print.saw`:

```
print 3;
```

The command `saw print.saw` will then yield output similar to the following:

```
Loading module Cryptol
Loading file "print.saw"
3
```

The same code can be run from the interactive REPL:

```
sawscript> print 3;
3
```

At the REPL, terminating semicolons can be omitted:

```
sawscript> print 3
3
```

To make common use cases simpler, bare values at the REPL are treated as if they were arguments to `print`:

```
sawscript> 3
3
```

One SAWScript file can be included in another using the `include` command, which takes the name of the file to be included as an argument. For example:

```
sawscript> include "print.saw"
Loading file "print.saw"
3
```

Typically, included files are used to import definitions, not perform side effects like printing. However, as you can see, if any commands with side effects occur at the top level of the imported file, those side effects will occur during import.

Syntax

The syntax of SAWScript is reminiscent of functional languages such as Cryptol, Haskell and ML. In particular, functions are applied by writing them next to their arguments rather than by using parentheses and commas. Rather than writing `f(x, y)`, write `f x y`.

Comments are written as in C, Java, and Rust (among many other languages). All text from `//` until the end of a line is ignored. Additionally, all text between `/*` and `*/` is ignored, regardless of whether the line ends.

Basic Types and Values

All values in SAWScript have types, and these types are determined and checked before a program runs (that is, SAWScript is statically typed). The basic types available are similar to those in many other languages.

- The `Int` type represents unbounded mathematical integers. Integer constants can be written in decimal notation (e.g., `42`), hexadecimal notation (`0x2a`), and binary (`0b00101010`). However, unlike many languages, integers in SAWScript are used primarily as constants. Arithmetic is usually encoded in Cryptol, as discussed in the next section.
- The Boolean type, `Bool`, contains the values `true` and `false`, like in many other languages. As with integers, computations on Boolean values usually occur in Cryptol.

- Values of any type can be aggregated into tuples. For example, the value `(true, 10)` has the type `(Bool, Int)`.
- Values of any type can also be aggregated into records, which are exactly like tuples except that their components have names. For example, the value `{ b = true, n = 10 }` has the type `{ b : Bool, n : Int }`.
- A sequence of values of the same type can be stored in a list. For example, the value `[true, false, true]` has the type `[Bool]`.
- Strings of textual characters can be represented in the `String` type. For example, the value `"example"` has type `String`.
- The "unit" type, written `()`, is essentially a placeholder, similar to `void` in languages like C and Java. It has only one value, also written `()`. Values of type `()` convey no information. We will show in later sections several cases where this is useful.
- Functions are given types that indicate what type they consume and what type they produce. For example, the type `Int -> Bool` indicates a function that takes an `Int` as input and produces a `Bool` as output. Functions with multiple arguments use multiple arrows. For example, the type `Int -> String -> Bool` indicates a function in which the first argument is an `Int`, the second is a `String`, and the result is a `Bool`. It is possible, but not necessary, to group arguments in tuples, as well, so the type `(Int, String)-> Bool` describes a function that takes one argument, a pair of an `Int` and a `String`, and returns a `Bool`.

SAWScript also includes some more specialized types that do not have straightforward counterparts in most other languages. These will appear in later sections.

Basic Expression Forms

One of the key forms of top-level command in SAWScript is a *binding*, introduced with the `let` keyword, which gives a name to a value. For example:

```
sawscript> let x = 5
sawscript> x
5
```

Bindings can have parameters, in which case they define functions. For instance, the following function takes one parameter and constructs a list containing that parameter as its single element.

```
sawscript> let f x = [x]
sawscript> f "text"
["text"]
```

Functions themselves are values and have types. The type of a function that takes an argument of type `a` and returns a result of type `b` is `a -> b`.

Function types are typically inferred, as in the example `f` above. In this case, because `f` only creates a list with the given argument, and because it is possible to create a list of any element type, `f` can be applied to an argument of any type. We say, therefore, that `f` is *polymorphic*. Concretely, we write the type of `f` as `{a} a -> [a]`, meaning it takes a value of any type (denoted `a`) and returns a list containing elements of that same type. This means we can also apply `f` to `10`:

```
sawscript> f 10
[10]
```

However, we may want to specify that a function has a more specific type. In this case, we could restrict `f` to operate only on `Int` parameters.

```
sawscript> let f (x : Int) = [x]
```

This will work identically to the original `f` on an `Int` parameter:

```
sawscript> f 10
[10]
```

However, it will fail for a `String` parameter:

```
sawscript> f "text"

type mismatch: String -> t.0 and Int -> [Int]
 at "_" (REPL)
mismatched type constructors: String and Int
```

Type annotations can be applied to any expression. The notation `(e : t)` indicates that expression `e` is expected to have type `t` and that it is an error for `e` to have a different type. Most types in SAWScript are inferred automatically, but specifying them explicitly can sometimes enhance readability.

Because functions are values, functions can return other functions. We make use of this feature when writing functions of multiple arguments. Consider the function `g`, similar to `f` but with two arguments:

```
sawscript> let g x y = [x, y]
```

Like `f`, `g` is polymorphic. Its type is `{a} a -> a -> [a]`. This means it takes an argument of type `a` and returns a *function* that takes an argument of the same type `a` and returns a list of `a` values. We can therefore apply `g` to any two arguments of the same type:

```
sawscript> g 2 3
[2,3]
sawscript> g true false
[true,false]
```

But type checking will fail if we apply it to two values of different types:

```
sawscript> g 2 false

type mismatch: Bool -> t.0 and Int -> [Int]
 at "_" (REPL)
mismatched type constructors: Bool and Int
```

So far we have used two related terms, *function* and *command*, and we take these to mean slightly different things. A function is any value with a function type (e.g., `Int -> [Int]`). A command is any value with a special command type (e.g. `TopLevel ()`, as shown below). These special types allow us to restrict command usage to specific contexts, and are also *parameterized* (like the list type). Most but not all commands are also functions.

The most important command type is the `TopLevel` type, indicating a command that can run at the top level (directly at the REPL, or as one of the top level commands in a script file). The `print` command has the type `{a} a -> TopLevel ()`, where `TopLevel ()` means that it is a command that runs in the `TopLevel` context and returns a value of type `()` (that is, no useful information). In other words, it has a side effect (printing some text to the screen) but doesn't produce any information to use in the rest of the SAWScript program. This is the primary usage of the `()` type.

It can sometimes be useful to bind a sequence of commands together. This can be accomplished with the `do { ... }` construct. For example:

```
sawscript> let print_two = do { print "first"; print "second"; }
sawscript> print_two
first
second
```

The bound value, `print_two`, has type `TopLevel ()`, since that is the type of its last command.

Note that in the previous example the printing doesn't occur until `print_two` directly appears at the REPL. The `let` expression does not cause those commands to run. The construct that *runs* a command is written using the `<-` operator. This operator works like `let` except that it says to run the command listed on the right hand side and bind the result, rather than binding the variable to the command itself. Using `<-` instead of `let` in the previous example yields:

```
sawscript> print_two <- do { print "first"; print "second"; }
first
second
sawscript> print print_two
()
```

Here, the `print` commands run first, and then `print_two` gets the value returned by the second `print` command, namely `()`. Any command run without using `<-` at either the top level of a script or within a `do` block discards its result. However, the REPL prints the result of any command run without using the `<-` operator.

In some cases it can be useful to have more control over the value returned by a `do` block. The `return` command allows us to do this. For example, say we wanted to write a function that would print a message before and after running some arbitrary command and then return the result of that command. We could write:

```
let run_with_message msg c =
  do {
    print "Starting.";
    print msg;
    res <- c;
    print "Done.";
    return res;
  };

x <- run_with_message "Hello" (return 3);
print x;
```

If we put this script in `run.saw` and run it with `saw`, we get something like:

```
Loading module Cryptol
Loading file "run.saw"
Starting.
Hello
Done.
3
```

Note that it ran the first `print` command, then the caller-specified command, then the second `print` command. The result stored in `x` at the end is the result of the `return` command passed in as an argument.

Other Basic Functions

Aside from the functions we have listed so far, there are a number of other operations for working with basic data structures and interacting with the operating system.

The following functions work on lists:

- `concat : {a} [a] -> [a] -> [a]` takes two lists and returns the concatenation of the two.
- `head : {a} [a] -> a` returns the first element of a list.
- `tail : {a} [a] -> [a]` returns everything except the first element.
- `length : {a} [a] -> Int` counts the number of elements in a list.
- `null : {a} [a] -> Bool` indicates whether a list is empty (has zero elements).
- `nth : {a} [a] -> Int -> a` returns the element at the given position, with `nth l 0` being equivalent to `head l`.
- `for : {m, a, b} [a] -> (a -> m b)-> m [b]` takes a list and a function that runs in some command context. The passed command will be called once for every element of the list, in order. Returns a list of all of the results produced by the command.

For interacting with the operating system, we have:

- `get_opt : Int -> String` returns the command-line argument to `saw` at the given index. Argument 0 is always the name of the `saw` executable itself, and higher indices represent later arguments.
- `exec : String -> [String] -> String -> TopLevel String` runs an external program given, respectively, an executable name, a list of arguments, and a string to send to the standard input of the program. The `exec` command returns the standard output from the program it executes and prints standard error to the screen.
- `exit : Int -> TopLevel ()` stops execution of the current script and returns the given exit code to the operating system.

Finally, there are a few miscellaneous functions and commands:

- `show : {a} a -> String` computes the textual representation of its argument in the same way as `print`, but instead of displaying the value it returns it as a `String` value for later use in the program. This can be useful for constructing more detailed messages later.
- `str_concat : String -> String -> String` concatenates two `String` values, and can also be useful with `show`.
- `time : {a} TopLevel a -> TopLevel a` runs any other `TopLevel` command and prints out the time it took to execute.
- `with_time : {a} TopLevel a -> TopLevel (Int, a)` returns both the original result of the timed command and the time taken to execute it (in milliseconds), without printing anything in the process.

The Term Type

Perhaps the most important type in SAWScript, and the one most unlike the built-in types of most other languages, is the `Term` type. Essentially, a value of type `Term` precisely describes all possible computations performed by some program. In particular, if two `Term` values are *equivalent*, then the programs that they represent will always compute the same results given the same inputs. We will say more later about exactly what it means for two terms to be equivalent, and how to determine whether two terms are equivalent.

Before exploring the `Term` type more deeply, it is important to understand the role of the Cryptol language in SAW.

Cryptol and its Role in SAW

Cryptol is a domain-specific language originally designed for the high-level specification of cryptographic algorithms. It is general enough, however, to describe a wide variety of programs, and is particularly applicable to describing computations that operate on streams of data of some fixed size.

In addition to being integrated into SAW, Cryptol is a standalone language with its own manual:

```
http://cryptol.net/files/ProgrammingCryptol.pdf
```

SAW includes deep support for Cryptol, and in fact requires the use of Cryptol for most non-trivial tasks. To fully understand the rest of this manual and to effectively use SAW, you will need to develop at least a rudimentary understanding of Cryptol.

The primary use of Cryptol within SAWScript is to construct values of type `Term`. Although `Term` values can be constructed from various sources, inline Cryptol expressions are the most direct and convenient way to create them.

Specifically, a Cryptol expression can be placed inside double curly braces (`{{` and `}}`), resulting in a value of type `Term`. As a very simple example, there is no built-in integer addition operation in SAWScript. However, we can use Cryptol's built-in integer addition operator within SAWScript as follows:

```
sawscript> let t = {{ 0x22 + 0x33 }}
sawscript> print t
85
sawscript> :type t
Term
```

Although it printed out in the same way as an `Int`, it is important to note that `t` actually has type `Term`. We can see how this term is represented internally, before being evaluated, with the `print_term` function.

```
sawscript> print_term t
let { x@1 = Prelude.Vec 8 Prelude.Bool
      x@2 = Cryptol.TCNum 8
      x@3 = Cryptol.PLiteralSeqBool x@2
    }
 in Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool x@2)
      (Cryptol.ecNumber (Cryptol.TCNum 34) x@1 x@3)
      (Cryptol.ecNumber (Cryptol.TCNum 51) x@1 x@3)
```

For the moment, it's not important to understand what this output means. We show it only to clarify that `Term` values have their own internal structure that goes beyond what exists in SAWScript. The internal representation of `Term` values is in a language called SAWCore. The full semantics of SAWCore are beyond the scope of this manual.

The text constructed by `print_term` can also be accessed programmatically (instead of printing to the screen) using the `show_term` function, which returns a `String`. The `show_term` function is not a command, so it executes directly and does not need `<-` to bind its result. Therefore, the following will have the same result as the `print_term` command above:

```
sawscript> let s = show_term t
sawscript> :type s
```

```
String
sawscript> print s
<same as above>
```

Numbers are printed in decimal notation by default when printing terms, but the following two commands can change that behavior.

- `set_ascii : Bool -> TopLevel ()`, when passed `true`, makes subsequent `print_term` or `show_term` commands print sequences of bytes as ASCII strings (and doesn't affect printing of anything else).
- `set_base : Int -> TopLevel ()` prints all bit vectors in the given base, which can be between 2 and 36 (inclusive).

A `Term` that represents an integer (any bit vector, as affected by `set_base`) can be translated into a SAWScript `Int` using the `eval_int : Term -> Int` function. This function returns an `Int` if the `Term` can be represented as one, and fails at runtime otherwise.

```
sawscript> print (eval_int t)
85
sawscript> print (eval_int {{ True }})

"eval_int" (<stdin>:1:1):
eval_int: argument is not a finite bitvector
sawscript> print (eval_int {{ [True] }})
1
```

Similarly, values of type `Bit` in Cryptol can be translated into values of type `Bool` in SAWScript using the `eval_bool : Term -> Bool` function:

```
sawscript> let b = {{ True }}
sawscript> print_term b
Prelude.True
sawscript> print (eval_bool b)
true
```

Anything with sequence type in Cryptol can be translated into a list of `Term` values in SAWScript using the `eval_list : Term -> [Term]` function.

```
sawscript> let l = {{ [0x01, 0x02, 0x03] }}
sawscript> print_term l
let { x@1 = Prelude.Vec 8 Prelude.Bool
      x@2 = Cryptol.PLiteralSeqBool (Cryptol.TCNum 8)
    }
 in [Cryptol.ecNumber (Cryptol.TCNum 1) x@1 x@2
    ,Cryptol.ecNumber (Cryptol.TCNum 2) x@1 x@2
    ,Cryptol.ecNumber (Cryptol.TCNum 3) x@1 x@2]
sawscript> print (eval_list l)
[Cryptol.ecNumber (Cryptol.TCNum 1) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))
,Cryptol.ecNumber (Cryptol.TCNum 2) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))
,Cryptol.ecNumber (Cryptol.TCNum 3) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))]
```

Finally, a list of `Term` values in SAWScript can be collapsed into a single `Term` with sequence type using the `list_term : [Term] -> Term` function, which is the inverse of `eval_list`.

```
sawscript> let ts = eval_list l
sawscript> let l = list_term ts
sawscript> print_term l
let { x@1 = Prelude.Vec 8 Prelude.Bool
      x@2 = Cryptol.PLiteralSeqBool (Cryptol.TCNum 8)
    }
 in [Cryptol.ecNumber (Cryptol.TCNum 1) x@1 x@2
    ,Cryptol.ecNumber (Cryptol.TCNum 2) x@1 x@2
    ,Cryptol.ecNumber (Cryptol.TCNum 3) x@1 x@2]
```

In addition to being able to extract integer and Boolean values from Cryptol expressions, `Term` values can be injected into Cryptol expressions. When SAWScript evaluates a Cryptol expression between `{{` and `}}` delimiters, it does so with several extra bindings in scope:

- Any variable in scope that has SAWScript type `Bool` is visible in Cryptol expressions as a value of type `Bit`.
- Any variable in scope that has SAWScript type `Int` is visible in Cryptol expressions as a *type variable*. Type variables can be demoted to numeric bit vector values using the backtick (`` ` ``) operator.
- Any variable in scope that has SAWScript type `Term` is visible in Cryptol expressions as a value with the Cryptol type corresponding to the internal type of the term. The power of this conversion is that the `Term` does not need to have originally been derived from a Cryptol expression.

In addition to these rules, bindings created at the Cryptol level, either from included files or inside Cryptol quoting brackets, are visible only to later Cryptol expressions, and not as SAWScript variables.

To make these rules more concrete, consider the following examples. If we bind a SAWScript `Int`, we can use it as a Cryptol type variable. If we create a `Term` variable that internally has function type, we can apply it to an argument within a Cryptol expression, but not at the SAWScript level:

```
sawscript> let n = 8
sawscript> :type n
Int
sawscript> let {{ f (x : [n]) = x + 1 }}
sawscript> :type {{ f }}
Term
sawscript> :type f

<stdin>:1:1-1:2: unbound variable: "f" (<stdin>:1:1-1:2)
sawscript> print {{ f 2 }}
3
```

If `f` was a binding of a SAWScript variable to a `Term` of function type, we would get a different error:

```
sawscript> let f = {{ \(x : [n]) -> x + 1 }}
sawscript> :type {{ f }}
Term
sawscript> :type f
Term
sawscript> print {{ f 2 }}
3
sawscript> print (f 2)
```

```
type mismatch: Int -> t.0 and Term
 at "_" (REPL)
 mismatched type constructors: (->) and Term
```

One subtlety of dealing with `Term`s constructed from Cryptol is that because the Cryptol expressions themselves are type checked by the Cryptol type checker, and because they may make use of other `Term` values already in scope, they are not type checked until the Cryptol brackets are evaluated. So type errors at the Cryptol level may occur at runtime from the SAWScript perspective (though they occur before the Cryptol expressions are run).

So far, we have talked about using Cryptol *value* expressions. However, SAWScript can also work with Cryptol *types*. The most direct way to refer to a Cryptol type is to use type brackets: `{|` and `|}`. Any Cryptol type written between these brackets becomes a `Type` value in SAWScript. Some types in Cryptol are *numeric* (also known as *size*) types, and correspond to non-negative integers. These can be translated into SAWScript integers with the `eval_size` function. For example:

```
sawscript> let {{ type n = 16 }}
sawscript> eval_size {| n |}
16
sawscript> eval_size {| 16 |}
16
```

For non-numeric types, `eval_size` fails at runtime:

```
sawscript> eval_size {| [16] |}

"eval_size" (<stdin>:1:1):
eval_size: not a numeric type
```

In addition to the use of brackets to write Cryptol expressions inline, several built-in functions can extract `Term` values from Cryptol files in other ways. The `import` command at the top level imports all top-level definitions from a Cryptol file and places them in scope within later bracketed expressions.

The `cryptol_load` command behaves similarly, but returns a `CryptolModule` instead. If any `CryptolModule` is in scope, its contents are available qualified with the name of the `CryptolModule` variable. A specific definition can be explicitly extracted from a `CryptolModule` using the `cryptol_extract` command:

- `cryptol_extract : CryptolModule -> String -> TopLevel Term`

Transforming Term Values

The three primary functions of SAW are *extracting* models (`Term` values) from programs, *transforming* those models, and *proving* properties about models using external provers. So far we've shown how to construct `Term` values from Cryptol programs; later sections will describe how to extract them from other programs. Now we show how to use the various term transformation features available in SAW.

Rewriting

Rewriting a `Term` consists of applying one or more *rewrite rules* to it, resulting in a new `Term`. A rewrite rule in SAW can be specified in multiple ways:

1. as the definition of a function that can be unfolded,
2. as a term of Boolean type (or a function returning a Boolean) that is an equality statement, and

3. as a term of *equality type* with a body that encodes a proof that the equality in the type is valid.

In each case the term logically consists of two sides and describes a way to transform the left side into the right side. Each side may contain variables (bound by enclosing lambda expressions) and is therefore a *pattern* which can match any term in which each variable represents an arbitrary sub-term. The left-hand pattern describes a term to match (which may be a sub-term of the full term being rewritten), and the right-hand pattern describes a term to replace it with. Any variable in the right-hand pattern must also appear in the left-hand pattern and will be instantiated with whatever sub-term matched that variable in the original term.

For example, say we have the following Cryptol function:

```
\(x:[8]) -> (x * 2) + 1
```

We might for some reason want to replace multiplication by a power of two with a shift. We can describe this replacement using an equality statement in Cryptol (a rule of form 2 above):

```
\(y:[8]) -> (y * 2) == (y << 1)
```

Interpreting this as a rewrite rule, it says that for any 8-bit vector (call it `y` for now), we can replace `y * 2` with `y << 1`. Using this rule to rewrite the earlier expression would then yield:

```
\(x:[8]) -> (x << 1) + 1
```

The general philosophy of rewriting is that the left and right patterns, while syntactically different, should be semantically equivalent. Therefore, applying a set of rewrite rules should not change the fundamental meaning of the term being rewritten. SAW is particularly focused on the task of proving that some logical statement expressed as a `Term` is always true. If that is in fact the case, then the entire term can be replaced by the term `True` without changing its meaning. The rewriting process can in some cases, by repeatedly applying rules that themselves are known to be valid, reduce a complex term entirely to `True`, which constitutes a proof of the original statement. In other cases, rewriting can simplify terms before sending them to external automated provers that can then finish the job. Sometimes this simplification can help the automated provers run more quickly, and sometimes it can help them prove things they would otherwise be unable to prove by applying reasoning steps (rewrite rules) that are not available to the automated provers.

In practical use, rewrite rules can be aggregated into `Simpset` values in SAWScript. A few pre-defined `Simpset` values exist:

- `empty_ss : Simpset` is the empty set of rules. Rewriting with it should have no effect, but it is useful as an argument to some of the functions that construct larger `Simpset` values.
- `basic_ss : Simpset` is a collection of rules that are useful in most proof scripts.
- `cryptol_ss : ()-> Simpset` includes a collection of Cryptol-specific rules. Some of these simplify away the abstractions introduced in the translation from Cryptol to SAWCore, which can be useful when proving equivalence between Cryptol and non-Cryptol code. Leaving these abstractions in place is appropriate when comparing only Cryptol code, however, so `cryptol_ss` is not included in `basic_ss`.

The next set of functions can extend or apply a `Simpset`:

- `addsimp' : Term -> Simpset -> Simpset` adds a single `Term` to an existing ‘Simpset.
- `addsimps' : [Term] -> Simpset -> Simpset` adds a list of `Term`s to an existing `Simpset`.
- `rewrite : Simpset -> Term -> Term` applies a `Simpset` to an existing `Term` to produce a new `Term`.

To make this more concrete, we examine how the rewriting example sketched above, to convert multiplication into shift, can work in practice. We simplify everything with `cryptol_ss` as we go along so that the `Term`s don’t get too cluttered. First, we declare the term to be transformed:

```
sawscript> let term = rewrite (cryptol_ss ()) {{ \(x:[8]) -> (x * 2) + 1
    }}
sawscript> print_term term
\(x : Prelude.Vec 8 Prelude.Bool) ->
  Prelude.bvAdd 8 (Prelude.bvMul 8 x (Prelude.bvNat 8 2))
    (Prelude.bvNat 8 1)
```

Next, we declare the rewrite rule:

```
sawscript> let rule = rewrite (cryptol_ss ()) {{ \(y:[8]) -> (y * 2) ==
   (y << 1) }}
sawscript> print_term rule
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in \(y : x@1) ->
      Cryptol.ecEq x@1 (Cryptol.PCmpWord 8)
        (Prelude.bvMul 8 y (Prelude.bvNat 8 2))
        (Prelude.bvShiftL 8 Prelude.Bool 1 Prelude.False y
           (Prelude.bvNat 1 1))
```

Finally, we apply the rule to the target term:

```
sawscript> let result = rewrite (addsimp' rule empty_ss) term
sawscript> print_term result
\(x : Prelude.Vec 8 Prelude.Bool) ->
  Prelude.bvAdd 8
    (Prelude.bvShiftL 8 Prelude.Bool 1 Prelude.False x
       (Prelude.bvNat 1 1))
    (Prelude.bvNat 8 1)
```

Note that `addsimp'` and `addsimps'` take a `Term` or list of `Terms`; these could in principle be anything, and are not necessarily terms representing logically valid equalities. They have `'` suffixes because they are not intended to be the primary interface to rewriting. When using these functions, the soundness of the proof process depends on the correctness of these rules as a side condition.

The primary interface to rewriting uses the `Theorem` type instead of the `Term` type, as shown in the signatures for `addsimp` and `addsimps`.

- `addsimp : Theorem -> Simpset -> Simpset` adds a single `Theorem` to a `Simpset`.
- `addsimps : [Theorem] -> Simpset -> Simpset` adds several `Theorem` values to a `Simpset`.

A `Theorem` is essentially a `Term` that is proven correct in some way. In general, a `Theorem` can be any statement, and may not be useful as a rewrite rule. However, if it has an appropriate shape it can be used for rewriting. In the "Proofs about Terms" section, we'll describe how to construct `Theorem` values from `Term` values.

In the absence of user-constructed `Theorem` values, there are some additional built-in rules that are not included in either `basic_ss` and `cryptol_ss` because they are not always beneficial, but that can sometimes be helpful or essential. The `cryptol_ss` simpset includes rewrite rules to unfold all definitions in the `Cryptol` SAWCore module, but does not include any of the terms of equality type.

- `add_cryptol_defs` :[String] -> Simpset -> Simpset`adds unfolding rules for functions with the given names from the SAWCore`Cryptol`module to the given`Simpset‘.
- `add_cryptol_eqs : [String] -> Simpset -> Simpset` adds the terms of equality type with the given names from the SAWCore `Cryptol` module to the given `Simpset`.

- `add_prelude_defs : [String] -> Simpset -> Simpset` adds unfolding rules from the SAWCore `Prelude` module to a `Simpset`.
- `add_prelude_eqs : [String] -> Simpset -> Simpset` adds equality-typed terms from the SAWCore `Prelude` module to a `Simpset`.

Finally, it's possible to construct a theorem from an arbitrary SAWCore expression (rather than a Cryptol expression), using the `core_axiom` function.

- `core_axiom : String -> Theorem` creates a `Theorem` from a `String` in SAWCore syntax. Any `Theorem` introduced by this function is assumed to be correct, so use it with caution.

Folding and Unfolding

A SAWCore term can be given a name using the `define` function, and is then by default printed as that name alone. A named subterm can be "unfolded" so that the original definition appears again.

- `define : String -> Term -> TopLevel Term`
- `unfold_term : [String] -> Term -> Term`

For example:

```
sawscript> let t = {{ 0x22 }}
sawscript> print_term t
Cryptol.ecNumber (Cryptol.TCNum 34) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))
sawscript> t' <- define "t" t
sawscript> print_term t'
t
sawscript> let t'' = unfold_term ["t"] t'
sawscript> print_term t''
Cryptol.ecNumber (Cryptol.TCNum 34) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))
```

This process of folding and unfolding is useful both to make large terms easier for humans to work with and to make automated proofs more tractable. We'll describe the latter in more detail when we discuss interacting with external provers.

In some cases, folding happens automatically when constructing Cryptol expressions. Consider the following example:

```
sawscript> let t = {{ 0x22 }}
sawscript> print_term t
Cryptol.ecNumber (Cryptol.TCNum 34) (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.PLiteralSeqBool (Cryptol.TCNum 8))
sawscript> let {{ t' = 0x22 }}
sawscript> print_term {{ t' }}
t'
```

This illustrates that a bare expression in Cryptol braces gets translated directly to a SAWCore term. However, a Cryptol *definition* gets translated into a *folded* SAWCore term. In addition, because the second definition of `t` occurs at the Cryptol level, rather than the SAWScript level, it is visible only inside Cryptol braces. Definitions imported from Cryptol source files are also initially folded and can be unfolded as needed.

Other Built-in Transformation and Inspection Functions

In addition to the `Term` transformation functions described so far, a variety of others also exist.

- `beta_reduce_term : Term -> Term` takes any sub-expression of the form `(\x -> t)v` in the given `Term` and replaces it with a transformed version of `t` in which all instances of `x` are replaced by `v`.
- `replace : Term -> Term -> Term -> TopLevel Term` replaces arbitrary subterms. A call to `replace x y t` replaces any instance of `x` inside `t` with `y`.

Assessing the size of a term can be particularly useful during benchmarking. SAWScript provides two mechanisms for this.

- `term_size : Term -> Int` calculates the number of nodes in the Directed Acyclic Graph (DAG) representation of a `Term` used internally by SAW. This is the most appropriate way of determining the resource use of a particular term.
- `term_tree_size : Term -> Int` calculates how large a `Term` would be if it were represented by a tree instead of a DAG. This can, in general, be much, much larger than the number returned by `term_size`, and serves primarily as a way of assessing, for a specific term, how much benefit there is to the term sharing used by the DAG representation.

Finally, there are a few commands related to the internal SAWCore type of a `Term`.

- `check_term : Term -> TopLevel ()` checks that the internal structure of a `Term` is well-formed and that it passes all of the rules of the SAWCore type checker.
- `type : Term -> Type` returns the type of a particular `Term`, which can then be used to, for example, construct a new fresh variable with `fresh_symbolic`.

Loading and Storing Terms

Most frequently, `Term` values in SAWScript come from Cryptol, JVM, or LLVM programs, or some transformation thereof. However, it is also possible to obtain them from various other sources.

- `parse_core : String -> Term` parses a `String` containing a term in SAWCore syntax, returning a `Term`.
- `read_core : String -> TopLevel Term` is like `parse_core`, but obtains the text from the given file and expects it to be in the simpler SAWCore external representation format, rather than the human-readable syntax shown so far.
- `read_aig : String -> TopLevel Term` returns a `Term` representation of an And-Inverter-Graph (AIG) file in AIGER format.
- `read_bytes : String -> TopLevel Term` reads a constant sequence of bytes from a file and represents it as a `Term`. Its result will always have Cryptol type `[n][8]` for some `n`.

It is also possible to write `Term` values into files in various formats, including: AIGER (`write_aig`), CNF (`write_cnf`), SAWCore external representation (`write_core`), and SMT-Lib version 2 (`write_smtlib2`).

- `write_aig : String -> Term -> TopLevel ()`
- `write_cnf : String -> Term -> TopLevel ()`
- `write_core : String -> Term -> TopLevel ()`
- `write_smtlib2 : String -> Term -> TopLevel ()`

Proofs about Terms

The goal of SAW is to facilitate proofs about the behavior of programs. It may be useful to prove some small fact to use as a rewrite rule in later proofs, but ultimately these rewrite rules come together into a proof of some higher-level property about a software system.

Whether proving small lemmas (in the form of rewrite rules) or a top-level theorem, the process builds on the idea of a *proof script* that is run by one of the top level proof commands.

- `prove_print : ProofScript ()-> Term -> TopLevel Theorem` takes a proof script (which we'll describe next) and a `Term`. The `Term` should be of function type with a return value of `Bool` (`Bit` at the Cryptol level). It will then use the proof script to attempt to show that the `Term` returns `True` for all possible inputs. If it is successful, it will print `Valid` and return a `Theorem`. If not, it will abort.
- `sat_print : ProofScript ()-> Term -> TopLevel ()` is similar except that it looks for a *single* value for which the `Term` evaluates to `True` and prints out that value, returning nothing.
- `prove_core : ProofScript ()-> String -> TopLevel Theorem` proves and returns a `Theorem` from a string in SAWCore syntax.

Automated Tactics

The simplest proof scripts just specify the automated prover to use. The `ProofScript` values `abc` and `z3` select the ABC and Z3 theorem provers, respectively, and are typically good choices.

For example, combining `prove_print` with `abc`:

```
sawscript> t <- prove_print abc {{ \(x:[8]) -> x+x == x*2 }}
Valid
sawscript> t
Theorem (let { x@1 = Prelude.Vec 8 Prelude.Bool
      x@2 = Cryptol.TCNum 8
      x@3 = Cryptol.PArithSeqBool x@2
    }
 in (x : x@1)
-> Prelude.EqTrue
     (Cryptol.ecEq x@1 (Cryptol.PCmpSeqBool x@2)
        (Cryptol.ecPlus x@1 x@3 x x)
        (Cryptol.ecMul x@1 x@3 x
           (Cryptol.ecNumber (Cryptol.TCNum 2) x@1
              (Cryptol.PLiteralSeqBool x@2)))))
```

Similarly, `sat_print` will show that the function returns `True` for one specific input (which it should, since we already know it returns `True` for all inputs):

```
sawscript> sat_print abc {{ \(x:[8]) -> x+x == x*2 }}
Sat: [x = 0]
```

In addition to these, the `boolector`, `cvc4`, `cvc5`, `mathsat`, and `yices` provers are available. The internal decision procedure `rme`, short for Reed-Muller Expansion, is an automated prover that works particularly well on the Galois field operations that show up, for example, in AES.

In more complex cases, some pre-processing can be helpful or necessary before handing the problem off to an automated prover. The pre-processing can involve rewriting, beta reduction, unfolding, the use of provers that require slightly more configuration, or the use of provers that do very little real work.

Proof Script Diagnostics

During development of a proof, it can be useful to print various information about the current goal. The following tactics are useful in that context.

- `print_goal : ProofScript ()` prints the entire goal in SAWCore syntax.
- `print_goal_consts : ProofScript ()` prints a list of unfoldable constants in the current goal.
- `print_goal_depth : Int -> ProofScript ()` takes an integer argument, `n`, and prints the goal up to depth `n`. Any elided subterms are printed with a `...` notation.
- `print_goal_size : ProofScript ()` prints the number of nodes in the DAG representation of the goal.

Rewriting in Proof Scripts

One of the key techniques available for completing proofs in SAWScript is the use of rewriting or transformation. The following commands support this approach.

- `simplify : Simpset -> ProofScript ()` works just like `rewrite`, except that it works in a `ProofScript` context and implicitly transforms the current (unnamed) goal rather than taking a `Term` as a parameter.
- `goal_eval : ProofScript ()` will evaluate the current proof goal to a first-order combination of primitives.
- `goal_eval_unint : [String] -> ProofScript ()` works like `goal_eval` but avoids expanding or simplifying the given names.

Other Transformations

Some useful transformations are not easily specified using equality statements, and instead have special tactics.

- `beta_reduce_goal : ProofScript ()` works like `beta_reduce_term` but on the current goal. It takes any sub-expression of the form `(\x -> t)v` and replaces it with a transformed version of `t` in which all instances of `x` are replaced by `v`.
- `unfolding : [String] -> ProofScript ()` works like `unfold_term` but on the current goal.

Using `unfolding` is mostly valuable for proofs based entirely on rewriting, since the default behavior for automated provers is to unfold everything before sending a goal to a prover. However, with some provers it is possible to indicate that specific named subterms should be represented as uninterpreted functions.

- `unint_cvc4 : [String] -> ProofScript ()`
- `unint_cvc5 : [String] -> ProofScript ()`
- `unint_yices : [String] -> ProofScript ()`
- `unint_z3 : [String] -> ProofScript ()`

The list of `String` arguments in these cases indicates the names of the subterms to leave folded, and therefore present as uninterpreted functions to the prover. To determine which folded constants appear in a goal, use the `print_goal_consts` function described above.

Ultimately, we plan to implement a more generic tactic that leaves certain constants uninterpreted in whatever prover is ultimately used (provided that uninterpreted functions are expressible in the prover).

Note that each of the `unint_*` tactics have variants that are prefixed with `sbv_` and `w4_`. The `sbv_`-prefixed tactics make use of the SBV library to represent and solve SMT queries:

- `sbv_unint_cvc4 : [String] -> ProofScript ()`
- `sbv_unint_cvc5 : [String] -> ProofScript ()`
- `sbv_unint_yices : [String] -> ProofScript ()`
- `sbv_unint_z3 : [String] -> ProofScript ()`

The `w4_`-prefixed tactics make use of the What4 library instead of SBV:

- `w4_unint_cvc4 : [String] -> ProofScript ()`
- `w4_unint_cvc5 : [String] -> ProofScript ()`
- `w4_unint_yices : [String] -> ProofScript ()`
- `w4_unint_z3 : [String] -> ProofScript ()`

In most specifications, the choice of SBV versus What4 is not important, as both libraries are broadly compatible in terms of functionality. There are some situations where one library may outpeform the other, however, due to differences in how each library represents certain SMT queries. There are also some experimental features that are only supported with What4 at the moment, such as `enable_lax_loads_and_stores`.

Other External Provers

In addition to the built-in automated provers already discussed, SAW supports more generic interfaces to other arbitrary theorem provers supporting specific interfaces.

- `external_aig_solver : String -> [String] -> ProofScript ()` supports theorem provers that can take input as a single-output AIGER file. The first argument is the name of the executable to run. The second argument is the list of command-line parameters to pass to that executable. Any element of this list equal to `"%f"` will be replaced with the name of the temporary AIGER file generated for the proof goal. The output from the solver is expected to be in DIMACS solution format.
- `external_cnf_solver : String -> [String] -> ProofScript ()` works similarly but for SAT solvers that take input in DIMACS CNF format and produce output in DIMACS solution format.

Offline Provers

For provers that must be invoked in more complex ways, or to defer proof until a later time, there are functions to write the current goal to a file in various formats, and then assume that the goal is valid through the rest of the script.

- `offline_aig : String -> ProofScript ()`
- `offline_cnf : String -> ProofScript ()`
- `offline_extcore : String -> ProofScript ()`
- `offline_smtlib2 : String -> ProofScript ()`
- `offline_unint_smtlib2 : [String] -> String -> ProofScript ()`

These support the AIGER, DIMACS CNF, shared SAWCore, and SMT-Lib v2 formats, respectively. The shared representation for SAWCore is described in the `saw-script` repository. The `offline_unint_smtlib2` command represents the folded subterms listed in its first argument as uninterpreted functions.

Finishing Proofs without External Solvers

Some proofs can be completed using unsound placeholders, or using techniques that do not require significant computation.

- `assume_unsat : ProofScript ()` indicates that the current goal should be assumed to be unsatisfiable. This is an alias for `assume_valid`. Users should prefer to use `admit` instead.
- `assume_valid : ProofScript ()` indicates that the current goal should be assumed to be valid. Users should prefer to use `admit` instead
- `admit : String -> ProofScript ()` indicates that the current goal should be assumed to be valid without proof. The given string should be used to record why the user has decided to assume this proof goal.
- `quickcheck : Int -> ProofScript ()` runs the goal on the given number of random inputs, and succeeds if the result of evaluation is always `True`. This is unsound, but can be helpful during proof development, or as a way to provide some evidence for the validity of a specification believed to be true but difficult or infeasible to prove.
- `trivial : ProofScript ()` states that the current goal should be trivially true. This tactic recognizes instances of equality that can be demonstrated by conversion alone. In particular it is able to prove `EqTrue x` goals where `x` reduces to the constant value `True`. It fails if this is not the case.

Multiple Goals

The proof scripts shown so far all have a single implicit goal. As in many other interactive provers, however, SAWScript proofs can have multiple goals. The following commands can introduce or work with multiple goals. These are experimental and can be used only after `enable_experimental` has been called.

- `goal_apply : Theorem -> ProofScript ()` will apply a given introduction rule to the current goal. This will result in zero or more new subgoals.
- `goal_assume : ProofScript Theorem` will convert the first hypothesis in the current proof goal into a local `Theorem`
- `goal_insert : Theorem -> ProofScript ()` will insert a given `Theorem` as a new hypothesis in the current proof goal.
- `goal_intro : String -> ProofScript Term` will introduce a quantified variable in the current proof goal, returning the variable as a `Term`.
- `goal_when : String -> ProofScript ()-> ProofScript ()` will run the given proof script only when the goal name contains the given string.
- `goal_exact : Term -> ProofScript ()` will attempt to use the given term as an exact proof for the current goal. This tactic will succeed whever the type of the given term exactly matches the current goal, and will fail otherwise.
- `split_goal : ProofScript ()` will split a goal of the form `Prelude.and prop1 prop2` into two separate goals `prop1` and `prop2`.

Proof Failure and Satisfying Assignments

The `prove_print` and `sat_print` commands print out their essential results (potentially returning a `Theorem` in the case of `prove_print`). In some cases, though, one may want to act programmatically on the result of a proof rather than displaying it.

The `prove` and `sat` commands allow this sort of programmatic analysis of proof results. To allow this, they use two types we haven't mentioned yet: `ProofResult` and `SatResult`. These are different from the other types in SAWScript because they encode the possibility of two outcomes. In the case of `ProofResult`, a statement may be valid or there may be a counter-example. In the case of `SatResult`, there may be a satisfying assignment or the statement may be unsatisfiable.

- `prove : ProofScript SatResult -> Term -> TopLevel ProofResult`
- `sat : ProofScript SatResult -> Term -> TopLevel SatResult`

To operate on these new types, SAWScript includes a pair of functions:

- `caseProofResult : {b} ProofResult -> b -> (Term -> b)-> b` takes a `ProofResult`, a value to return in the case that the statement is valid, and a function to run on the counter-example, if there is one.
- `caseSatResult : {b} SatResult -> b -> (Term -> b)-> b` has the same shape: it returns its first argument if the result represents an unsatisfiable statement, or its second argument applied to a satisfying assignment if it finds one.

AIG Values and Proofs

Most SAWScript programs operate on `Term` values, and in most cases this is the appropriate representation. It is possible, however, to represent the same function that a `Term` may represent using a different data structure: an And-Inverter-Graph (AIG). An AIG is a representation of a Boolean function as a circuit composed entirely of AND gates and inverters. Hardware synthesis and verification tools, including the ABC tool that SAW has built in, can do efficient verification and particularly equivalence checking on AIGs.

To take advantage of this capability, a handful of built-in commands can operate on AIGs.

- `bitblast : Term -> TopLevel AIG` represents a `Term` as an `AIG` by "blasting" all of its primitive operations (things like bit-vector addition) down to the level of individual bits.
- `load_aig : String -> TopLevel AIG` loads an `AIG` from an external AIGER file.
- `save_aig : String -> AIG -> TopLevel ()` saves an `AIG` to an external AIGER file.
- `save_aig_as_cnf : String -> AIG -> TopLevel ()` writes an `AIG` out in CNF format for input into a standard SAT solver.

Symbolic Execution

Analysis of Java and LLVM within SAWScript relies heavily on *symbolic execution*, so some background on how this process works can help with understanding the behavior of the available built-in functions.

At the most abstract level, symbolic execution works like normal program execution except that the values of all variables within the program can be arbitrary *expressions*, potentially containing free variables, rather than concrete values. Therefore, each symbolic execution corresponds to some set of possible concrete executions.

As a concrete example, consider the following C program that returns the maximum of two values:

```
unsigned int max(unsigned int x, unsigned int y) {
    if (y > x) {
        return y;
    } else {
        return x;
    }
}
```

If you call this function with two concrete inputs, like this:

```
int r = max(5, 4);
```

then it will assign the value `5` to `r`. However, we can also consider what it will do for *arbitrary* inputs. Consider the following example:

```
int r = max(a, b);
```

where `a` and `b` are variables with unknown values. It is still possible to describe the result of the `max` function in terms of `a` and `b`. The following expression describes the value of `r`:

```
ite (b > a) b a
```

where `ite` is the "if-then-else" mathematical function, which based on the value of the first argument returns either the second or third. One subtlety of constructing this expression, however, is the treatment of conditionals in the original program. For any concrete values of `a` and `b`, only one branch of the `if` statement will execute. During symbolic execution, on the other hand, it is necessary to execute *both* branches, track two different program states (each composed of symbolic values), and then *merge* those states after executing the `if` statement. This merging process takes into account the original branch condition and introduces the `ite` expression.

A symbolic execution system, then, is very similar to an interpreter that has a different notion of what constitutes a value and executes *all* paths through the program instead of just one. Therefore, the execution process is similar to that of a normal interpreter, and the process of generating a model for a piece of code is similar to building a test harness for that same code.

More specifically, the setup process for a test harness typically takes the following form:

- Initialize or allocate any resources needed by the code. For Java and LLVM code, this typically means allocating memory and setting the initial values of variables.
- Execute the code.
- Check the desired properties of the system state after the code completes.

Accordingly, three pieces of information are particularly relevant to the symbolic execution process, and are therefore needed as input to the symbolic execution system:

- The initial (potentially symbolic) state of the system.
- The code to execute.
- The final state of the system, and which parts of it are relevant to the properties being tested.

In the following sections, we describe how the Java and LLVM analysis primitives work in the context of these key concepts. We start with the simplest situation, in which the structure of the initial and final states can be directly inferred, and move on to more complex cases that require more information from the user.

Symbolic Termination

Above we described the process of executing multiple branches and merging the results when encountering a conditional statement in the program. When a program contains loops, the branch that chooses to continue or terminate a loop could go either way. Therefore, without a bit more information, the most obvious implementation of symbolic execution would never terminate when executing programs that contain loops.

The solution to this problem is to analyze the branch condition whenever considering multiple branches. If the condition for one branch can never be true in the context of the current symbolic state, there is no reason to execute that branch, and skipping it can make it possible for symbolic execution to terminate.

Directly comparing the branch condition to a constant can sometimes be enough to ensure termination. For example, in simple, bounded loops like the following, comparison with a constant is sufficient.

```
for (int i = 0; i < 10; i++) {
    // do something
}
```

In this case, the value of `i` is always concrete, and will eventually reach the value `10`, at which point the branch corresponding to continuing the loop will be infeasible.

As a more complex example, consider the following function:

```
uint8_t f(uint8_t i) {
  int done = 0;
  while (!done) {
    if (i % 8 == 0) done = 1;
    i += 5;
  }
  return i;
}
```

The loop in this function can only be determined to symbolically terminate if the analysis takes into account algebraic rules about common multiples. Similarly, it can be difficult to prove that a base case is eventually reached for all inputs to a recursive program.

In this particular case, however, the code *is* guaranteed to terminate after a fixed number of iterations (where the number of possible iterations is a function of the number of bits in the integers being used). To show that the last iteration is in fact the last possible one, it's necessary to do more than just compare the branch condition with a constant. Instead, we can use the same proof tools that we use to ultimately analyze the generated models to, early in the process, prove that certain branch conditions can never be true (i.e., are *unsatisfiable*).

Normally, most of the Java and LLVM analysis commands simply compare branch conditions to the constant `True` or `False` to determine whether a branch may be feasible. However, each form of analysis allows branch satisfiability checking to be turned on if needed, in which case functions like `f` above will terminate.

Next, we examine the details of the specific commands available to analyze JVM and LLVM programs.

Loading Code

The first step in analyzing any code is to load it into the system.

Loading LLVM

To load LLVM code, simply provide the location of a valid bitcode file to the `llvm_load_module` function.

- `llvm_load_module : String -> TopLevel LLVMModule`

The resulting `LLVMModule` can be passed into the various functions described below to perform analysis of specific LLVM functions.

The LLVM bitcode parser should generally work with LLVM versions between 3.5 and 16.0, though it may be incomplete for some versions. Debug metadata has changed somewhat throughout that version range, so is the most likely case of incompleteness. We aim to support every version after 3.5, however, so report any parsing failures as on GitHub.

Loading Java

Loading Java code is slightly more complex, because of the more structured nature of Java packages. First, when running `saw`, three flags control where to look for classes:

- The `-b` flag takes the path where the `java` executable lives, which is used to locate the Java standard library classes and add them to the class database. Alternatively, one can put the directory where `java` lives on the `PATH`, which SAW will search if `-b` is not set.
- The `-j` flag takes the name of a JAR file as an argument and adds the contents of that file to the class database.
- The `-c` flag takes the name of a directory as an argument and adds all class files found in that directory (and its subdirectories) to the class database. By default, the current directory is included in the class path.

Most Java programs will only require setting the `-b` flag (or the `PATH`), as that is enough to bring in the standard Java libraries. Note that when searching the `PATH`, SAW makes assumptions about where the standard library classes live. These assumptions are likely to hold on JDK 7 or later, but they may not hold on older JDKs on certain operating systems. If you are using an old version of the JDK and SAW is unable to find a standard Java class, you may need to specify the location of the standard classes' JAR file with the `-j` flag (or, alternatively, with the `SAW_JDK_JAR` environment variable).

Once the class path is configured, you can pass the name of a class to the `java_load_class` function.

- `java_load_class : String -> TopLevel JavaClass`

The resulting `JavaClass` can be passed into the various functions described below to perform analysis of specific Java methods.

Java class files from any JDK newer than version 6 should work. However, support for JDK 9 and later is experimental. Verifying code that only uses primitive data types is known to work well, but there are some as-of-yet unresolved issues in verifying code involving classes such as `String`. For more information on these issues, refer to this GitHub issue.

Loading MIR

To load a piece of Rust code, first compile it to a MIR JSON file, as described in this section, and then provide the location of the JSON file to the `mir_load_module` function:

- `mir_load_module : String -> TopLevel MIRModule`

SAW currently supports Rust code that can be built with a March 22, 2020 Rust nightly. If you encounter a Rust feature that SAW does not support, please report it on GitHub.

Notes on Compiling Code for SAW

SAW will generally be able to load arbitrary LLVM bitcode, JVM bytecode, and MIR JSON files, but several guidelines can help make verification easier or more likely to succeed.

Compiling LLVM

For generating LLVM with `clang`, it can be helpful to:

- Turn on debugging symbols with `-g` so that SAW can find source locations of functions, names of variables, etc.
- Optimize with `-O1` so that the generated bitcode more closely matches the C/C++ source, making the results more comprehensible.

- Use `-fno-threadsafe-statics` to prevent `clang` from emitting unnecessary pthread code.
- Link all relevant bitcode with `llvm-link` (including, *e.g.*, the C++ standard library when analyzing C++ code).

All SAW proofs include side conditions to rule out undefined behavior, and proofs will only succeed if all of these side conditions have been discharged. However the default SAW notion of undefined behavior is with respect to the semantics of LLVM, rather than C or C++. If you want to rule out undefined behavior according to the C or C++ standards, consider compiling your code with `-fsanitize=undefined` or one of the related flags[1] to `clang`.

Generally, you'll also want to use `-fsanitize-trap=undefined`, or one of the related flags, to cause the compiled code to use `llvm.trap` to indicate the presence of undefined behavior. Otherwise, the compiled code will call a separate function, such as `__ubsan_handle_shift_out_of_bounds`, for each type of undefined behavior, and SAW currently does not have built in support for these functions (though you could manually create overrides for them in a verification script).

Compiling Java

For Java, the only compilation flag that tends to be valuable is `-g` to retain information about the names of function arguments and local variables.

Compiling MIR

In order to verify Rust code, SAW analyzes Rust's MIR (mid-level intermediate representation) language. In particular, SAW analyzes a particular form of MIR that the `mir-json` tool produces. You will need to intall `mir-json` and run it on Rust code in order to produce MIR JSON files that SAW can load (see this section).

For `cargo`-based projects, `mir-json` provides a `cargo` subcommand called `cargo saw-build` that builds a JSON file suitable for use with SAW. `cargo saw-build` integrates directly with `cargo`, so you can pass flags to it like any other `cargo` subcommand. For example:

```
$ export SAW_RUST_LIBRARY_PATH=<...>
$ cargo saw-build <other cargo flags>
<snip>
linking 11 mir files into <...>/example-364cf2df365c7055.linked-mir.json
<snip>
```

Note that:

- The full output of `cargo saw-build` here is omitted. The important part is the `.linked-mir.json` file that appears after `linking X mir files into`, as that is the JSON file that must be loaded with SAW.
- `SAW_RUST_LIBRARY_PATH` should point to the the MIR JSON files for the Rust standard library.

`mir-json` also supports compiling individual `.rs` files through `mir-json`'s `saw-rustc` command. As the name suggests, it accepts all of the flags that `rustc` accepts. For example:

```
$ export SAW_RUST_LIBRARY_PATH=<...>
$ saw-rustc example.rs <other rustc flags>
<snip>
linking 11 mir files into <...>/example.linked-mir.json
<snip>
```

[1] https://clang.llvm.org/docs/UsersManual.html#controlling-code-generation

Notes on C++ Analysis

The distance between C++ code and LLVM is greater than between C and LLVM, so some additional considerations come into play when analyzing C++ code with SAW.

The first key issue is that the C++ standard library is large and complex, and tends to be widely used by C++ applications. To analyze most C++ code, it will be necessary to link your code with a version of the `libc++` library[2] compiled to LLVM bitcode. The `wllvm` program can[3] be useful for this.

The C++ standard library includes a number of key global variables, and any code that touches them will require that they be initialized using `llvm_alloc_global`.

Many C++ names are slightly awkward to deal with in SAW. They may be mangled relative to the text that appears in the C++ source code. SAW currently only understands the mangled names. The `llvm-nm` program can be used to show the list of symbols in an LLVM bitcode file, and the `c++filt` program can be used to demangle them, which can help in identifying the symbol you want to refer to. In addition, C++ names from namespaces can sometimes include quote marks in their LLVM encoding. For example:

```
%"class.quux::Foo" = type { i32, i32 }
```

This can be mentioned in SAW by saying:

```
llvm_type "%\"class.quux::Foo\""
```

Finally, there is no support for calling constructors in specifications, so you will need to construct objects piece-by-piece using, *e.g.*, `llvm_alloc` and `llvm_points_to`.

Direct Extraction

In the case of the `max` function described earlier, the relevant inputs and outputs are immediately apparent. The function takes two integer arguments, always uses both of them, and returns a single integer value, making no other changes to the program state.

In cases like this, a direct translation is possible, given only an identification of which code to execute. Two functions exist to handle such simple code. The first, for LLVM is the more stable of the two:

- `llvm_extract : LLVMModule -> String -> TopLevel Term`

A similar function exists for Java, but is more experimental.

- `jvm_extract : JavaClass -> String -> TopLevel Term`

Because of its lack of maturity, it (and later Java-related commands) must be enabled by running the `enable_experimental` command beforehand.

- `enable_experimental : TopLevel ()`

The structure of these two extraction functions is essentially identical. The first argument describes where to look for code (in either a Java class or an LLVM module, loaded as described in the previous section). The second argument is the name of the method or function to extract.

When the extraction functions complete, they return a `Term` corresponding to the value returned by the function or method as a function of its arguments.

These functions currently work only for code that takes some fixed number of integral parameters, returns an integral result, and does not access any dynamically-allocated memory (although temporary memory allocated during execution is allowed).

[2] https://libcxx.llvm.org/docs/BuildingLibcxx.html
[3] https://github.com/travitch/whole-program-llvm

Creating Symbolic Variables

The direct extraction process just discussed automatically introduces symbolic variables and then abstracts over them, yielding a SAWScript `Term` that reflects the semantics of the original Java or LLVM code. For simple functions, this is often the most convenient interface. For more complex code, however, it can be necessary (or more natural) to specifically introduce fresh variables and indicate what portions of the program state they correspond to.

- `fresh_symbolic : String -> Type -> TopLevel Term` is responsible for creating new variables in this context. The first argument is a name used for pretty-printing of terms and counter-examples. In many cases it makes sense for this to be the same as the name used within SAWScript, as in the following:

```
x <- fresh_symbolic "x" ty;
```

However, using the same name is not required.

The second argument to `fresh_symbolic` is the type of the fresh variable. Ultimately, this will be a SAWCore type; however, it is usually convenient to specify it using Cryptol syntax with the type quoting brackets `{|` and `|}`. For example, creating a 32-bit integer, as might be used to represent a Java `int` or an LLVM `i32`, can be done as follows:

```
x <- fresh_symbolic "x" {| [32] |};
```

Although symbolic execution works best on symbolic variables, which are "unbound" or "free", most of the proof infrastructure within SAW uses variables that are *bound* by an enclosing lambda expression. Given a `Term` with free symbolic variables, we can construct a lambda term that binds them in several ways.

- `abstract_symbolic : Term -> Term` finds all symbolic variables in the `Term` and constructs a lambda expression binding each one, in some order. The result is a function of some number of arguments, one for each symbolic variable. It is the simplest but least flexible way to bind symbolic variables.

```
sawscript> x <- fresh_symbolic "x" {| [8] |}
sawscript> let t = {{ x + x }}
sawscript> print_term t
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8))
      x
      x
sawscript> let f = abstract_symbolic t
sawscript> print_term f
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in \(x : x@1) ->
      Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8)) x x
```

If there are multiple symbolic variables in the `Term` passed to `abstract_symbolic`, the ordering of parameters can be hard to predict. In some cases (such as when a proof is the immediate next step, and it's expected to succeed) the order isn't important. In others, it's nice to have more control over the order.

- `lambda : Term -> Term -> Term` is the building block for controlled binding. It takes two terms: the one to transform, and the portion of the term to abstract over. Generally, the first `Term` is one obtained from `fresh_symbolic` and the second is a `Term` that would be passed to `abstract_symbolic`.

```
sawscript> let f = lambda x t
sawscript> print_term f
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in \(x : x@1) ->
      Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8)) x x
```

- `lambdas : [Term] -> Term -> Term` allows you to list the order in which symbolic variables should be bound. Consider, for example, a `Term` which adds two symbolic variables:

```
sawscript> x1 <- fresh_symbolic "x1" {| [8] |}
sawscript> x2 <- fresh_symbolic "x2" {| [8] |}
sawscript> let t = {{ x1 + x2 }}
sawscript> print_term t
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8))
      x1
      x2
```

We can turn `t` into a function that takes `x1` followed by `x2`:

```
sawscript> let f1 = lambdas [x1, x2] t
sawscript> print_term f1
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in \(x1 : x@1) ->
      \(x2 : x@1) ->
        Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8)) x1
          x2
```

Or we can turn `t` into a function that takes `x2` followed by `x1`:

```
sawscript> let f1 = lambdas [x2, x1] t
sawscript> print_term f1
let { x@1 = Prelude.Vec 8 Prelude.Bool
    }
 in \(x2 : x@1) ->
      \(x1 : x@1) ->
        Cryptol.ecPlus x@1 (Cryptol.PArithSeqBool (Cryptol.TCNum 8)) x1
          x2
```

Specification-Based Verification

The built-in functions described so far work by extracting models of code that can then be used for a variety of purposes, including proofs about the properties of the code.

When the goal is to prove equivalence between some LLVM or Java code and a specification, however, a more declarative approach is sometimes convenient. The following sections describe an approach that combines model extraction and verification with respect to a specification. A verified specification can then be used as input to future verifications, allowing the proof process to be decomposed.

Running a Verification

Verification of LLVM is controlled by the `llvm_verify` command.

```
llvm_verify :
  LLVMModule ->
  String ->
  [CrucibleMethodSpec] ->
  Bool ->
  LLVMSetup () ->
  ProofScript SatResult ->
  TopLevel CrucibleMethodSpec
```

The first two arguments specify the module and function name to verify, as with `llvm_verify`. The third argument specifies the list of already-verified specifications to use for compositional verification (described later; use `[]` for now). The fourth argument specifies whether to do path satisfiability checking, and the fifth gives the specification of the function to be verified. Finally, the last argument gives the proof script to use for verification. The result is a proved specification that can be used to simplify verification of functions that call this one.

A similar command for JVM programs is available if `enable_experimental` has been run.

```
jvm_verify :
  JavaClass ->
  String ->
  [JVMMethodSpec] ->
  Bool ->
  JVMSetup () ->
  ProofScript SatResult ->
  TopLevel JVMMethodSpec
```

Now we describe how to construct a value of type `LLVMSetup ()` (or `JVMSetup ()`).

Structure of a Specification

A specifications for Crucible consists of three logical components:

- A specification of the initial state before execution of the function.
- A description of how to call the function within that state.
- A specification of the expected final value of the program state.

These three portions of the specification are written in sequence within a `do` block of `LLVMSetup` (or `JVMSetup`) type. The command `llvm_execute_func` (or `jvm_execute_func`) separates the specification of the initial state from the specification of the final state, and specifies the arguments to the function in terms of the initial state. Most of the commands available for state description will work either before or after `llvm_execute_func`, though with slightly different meaning, as described below.

Creating Fresh Variables

In any case where you want to prove a property of a function for an entire class of inputs (perhaps all inputs) rather than concrete values, the initial values of at least some elements of the program state must contain fresh variables. These are created in a specification with the `llvm_fresh_var` and `jvm_fresh_var` commands rather than `fresh_symbolic`.

- `llvm_fresh_var : String -> LLVMType -> LLVMSetup Term`

- `jvm_fresh_var : String -> JavaType -> JVMSetup Term`

The first parameter to both functions is a name, used only for presentation. It's possible (though not recommended) to create multiple variables with the same name, but SAW will distinguish between them internally. The second parameter is the LLVM (or Java) type of the variable. The resulting `Term` can be used in various subsequent commands.

LLVM types are built with this set of functions:

- `llvm_int : Int -> LLVMType`
- `llvm_alias : String -> LLVMType`
- `llvm_array : Int -> LLVMType -> LLVMType`
- `llvm_float : LLVMType`
- `llvm_double : LLVMType`
- `llvm_packed_struct : [LLVMType] -> LLVMType`
- `llvm_struct : [LLVMType] -> LLVMType`

Java types are built up using the following functions:

- `java_bool : JavaType`
- `java_byte : JavaType`
- `java_char : JavaType`
- `java_short : JavaType`
- `java_int : JavaType`
- `java_long : JavaType`
- `java_float : JavaType`
- `java_double : JavaType`
- `java_class : String -> JavaType`
- `java_array : Int -> JavaType -> JavaType`

Most of these types are straightforward mappings to the standard LLVM and Java types. The one key difference is that arrays must have a fixed, concrete size. Therefore, all analysis results are valid only under the assumption that any arrays have the specific size indicated, and may not hold for other sizes. The `llvm_int` function also takes an `Int` parameter indicating the variable's bit width.

LLVM types can also be specified in LLVM syntax directly by using the `llvm_type` function.

- `llvm_type : String -> LLVMType`

For example, `llvm_type "i32"` yields the same result as `llvm_int 32`.

The most common use for creating fresh variables is to state that a particular function should have the specified behaviour for arbitrary initial values of the variables in question. Sometimes, however, it can be useful to specify that a function returns (or stores, more about this later) an arbitrary value, without specifying what that value should be. To express such a pattern, you can also run `llvm_fresh_var` from the post state (i.e., after `llvm_execute_func`).

The SetupValue and JVMValue Types

Many specifications require reasoning about both pure values and about the configuration of the heap. The `SetupValue` type corresponds to values that can occur during symbolic execution, which includes both `Term` values, pointers, and composite types consisting of either of these (both structures and arrays).

The `llvm_term` and `jvm_term` functions create a `SetupValue` or `JVMValue` from a `Term`:

- `llvm_term : Term -> SetupValue`
- `jvm_term : Term -> JVMValue`

Executing

Once the initial state has been configured, the `llvm_execute_func` command specifies the parameters of the function being analyzed in terms of the state elements already configured.

- `llvm_execute_func : [SetupValue] -> LLVMSetup ()`

Return Values

To specify the value that should be returned by the function being verified use the `llvm_return` or `jvm_return` command.

- `llvm_return : SetupValue -> LLVMSetup ()`
- `jvm_return : JVMValue -> JVMSetup ()`

A First Simple Example

The commands introuduced so far are sufficient to verify simple programs that do not use pointers (or that use them only internally). Consider, for instance the C program that adds its two arguments together:

```
#include <stdint.h>
uint32_t add(uint32_t x, uint32_t y) {
    return x + y;
}
```

We can specify this function's expected behavior as follows:

```
let add_setup = do {
    x <- llvm_fresh_var "x" (llvm_int 32);
    y <- llvm_fresh_var "y" (llvm_int 32);
    llvm_execute_func [llvm_term x, llvm_term y];
    llvm_return (llvm_term {{ x + y : [32] }});
};
```

We can then compile the C file `add.c` into the bitcode file `add.bc` and verify it with ABC:

```
m <- llvm_load_module "add.bc";
add_ms <- llvm_verify m "add" [] false add_setup abc;
```

Compositional Verification

The primary advantage of the specification-based approach to verification is that it allows for compositional reasoning. That is, when proving properties of a given method or function, we can make use of properties we have already proved about its callees rather than analyzing them anew. This enables us to reason about much larger and more complex systems than otherwise possible.

The `llvm_verify` and `jvm_verify` functions return values of type `CrucibleMethodSpec` and `JVMMethodSpec`, respectively. These values are opaque objects that internally contain both the information provided in the associated `JVMSetup` or `LLVMSetup` blocks and the results of the verification process.

Any of these `MethodSpec` objects can be passed in via the third argument of the `..._verify` functions. For any function or method specified by one of these parameters, the simulator will not follow calls to the associated target. Instead, it will perform the following steps:

- Check that all `llvm_points_to` and `llvm_precond` statements (or the corresponding JVM statements) in the specification are satisfied.

- Update the simulator state and optionally construct a return value as described in the specification.

More concretely, building on the previous example, say we have a doubling function written in terms of `add`:

```
uint32_t dbl(uint32_t x) {
    return add(x, x);
}
```

It has a similar specification to `add`:

```
let dbl_setup = do {
    x <- llvm_fresh_var "x" (llvm_int 32);
    llvm_execute_func [llvm_term x];
    llvm_return (llvm_term {{ x + x : [32] }});
};
```

And we can verify it using what we've already proved about `add`:

```
llvm_verify m "dbl" [add_ms] false dbl_setup abc;
```

In this case, doing the verification compositionally doesn't save computational effort, since the functions are so simple, but it illustrates the approach.

Specifying Heap Layout

Most functions that operate on pointers expect that certain pointers point to allocated memory before they are called. The `llvm_alloc` command allows you to specify that a function expects a particular pointer to refer to an allocated region appropriate for a specific type.

- `llvm_alloc : LLVMType -> LLVMSetup SetupValue`

This command returns a `SetupValue` consisting of a pointer to the allocated space, which can be used wherever a pointer-valued `SetupValue` can be used.

In the initial state, `llvm_alloc` specifies that the function expects a pointer to allocated space to exist. In the final state, it specifies that the function itself performs an allocation.

When using the experimental Java implementation, separate functions exist for specifying that arrays or objects are allocated:

- `jvm_alloc_array : Int -> JavaType -> JVMSetup JVMValue` specifies an array of the given concrete size, with elements of the given type.
- `jvm_alloc_object : String -> JVMSetup JVMValue` specifies an object of the given class name.

In LLVM, it's also possible to construct fresh pointers that do not point to allocated memory (which can be useful for functions that manipulate pointers but not the values they point to):

- `llvm_fresh_pointer : LLVMType -> LLVMSetup SetupValue`

The NULL pointer is called `llvm_null` in LLVM and `jvm_null` in JVM:

- `llvm_null : SetupValue`
- `jvm_null : JVMValue`

One final, slightly more obscure command is the following:

- `llvm_alloc_readonly : LLVMType -> LLVMSetup SetupValue`

This works like `llvm_alloc` except that writes to the space allocated are forbidden. This can be useful for specifying that a function should take as an argument a pointer to allocated space that it will not modify. Unlike `llvm_alloc`, regions allocated with `llvm_alloc_readonly` are allowed to alias other read-only regions.

Specifying Heap Values

Pointers returned by `llvm_alloc` don't, initially, point to anything. So if you pass such a pointer directly into a function that tried to dereference it, symbolic execution will fail with a message about an invalid load. For some functions, such as those that are intended to initialize data structures (writing to the memory pointed to, but never reading from it), this sort of uninitialized memory is appropriate. In most cases, however, it's more useful to state that a pointer points to some specific (usually symbolic) value, which you can do with the `llvm_points_to` command.

- `llvm_points_to : SetupValue -> SetupValue -> LLVMSetup ()` takes two `SetupValue` arguments, the first of which must be a pointer, and states that the memory specified by that pointer should contain the value given in the second argument (which may be any type of `SetupValue`).

When used in the final state, `llvm_points_to` specifies that the given pointer *should* point to the given value when the function finishes.

Occasionally, because C programs frequently reinterpret memory of one type as another through casts, it can be useful to specify that a pointer points to a value that does not agree with its static type.

- `llvm_points_to_untyped : SetupValue -> SetupValue -> LLVMSetup ()` works like `llvm_points_to` but omits type checking. Rather than omitting type checking across the board, we introduced this additional function to make it clear when a type reinterpretation is intentional. As an alternative, one may instead use `llvm_cast_pointer` to line up the static types.

Working with Compound Types

The commands mentioned so far give us no way to specify the values of compound types (arrays or `struct`s). Compound values can be dealt with either piecewise or in their entirety.

- `llvm_elem : SetupValue -> Int -> SetupValue` yields a pointer to an internal element of a compound value. For arrays, the `Int` parameter is the array index. For `struct` values, it is the field index.
- `llvm_field : SetupValue -> String -> SetupValue` yields a pointer to a particular named `struct` field, if debugging information is available in the bitcode.

Either of these functions can be used with `llvm_points_to` to specify the value of a particular array element or `struct` field. Sometimes, however, it is more convenient to specify all array elements or field values at once. The `llvm_array_value` and `llvm_struct_value` functions construct compound values from lists of element values.

- `llvm_array_value : [SetupValue] -> SetupValue`
- `llvm_struct_value : [SetupValue] -> SetupValue`

To specify an array or struct in which each element or field is symbolic, it would be possible, but tedious, to use a large combination of `llvm_fresh_var` and `llvm_elem` or `llvm_field` commands. However, the following function can simplify the common case where you want every element or field to have a fresh value.

- `llvm_fresh_expanded_val : LLVMType -> LLVMSetup SetupValue`

The `llvm_struct_value` function normally creates a `struct` whose layout obeys the alignment rules of the platform specified in the LLVM file being analyzed. Structs in LLVM can explicitly be "packed", however, so

that every field immediately follows the previous in memory. The following command will create values of such types:

- `llvm_packed_struct_value : [SetupValue] -> SetupValue`

C programs will sometimes make use of pointer casting to implement various kinds of polymorphic behaviors, either via direct pointer casts, or by using `union` types to codify the pattern. To reason about such cases, the following operation is useful.

- `llvm_cast_pointer : SetupValue -> LLVMType -> SetupValue`

This function function casts the type of the input value (which must be a pointer) so that it points to values of the given type. This mainly affects the results of subsequent `llvm_field` and `llvm_elem` calls, and any eventual `points_to` statements that the resulting pointer flows into. This is especially useful for dealing with C `union` types, as the type information provided by LLVM is imprecise in these cases.

We can automate the process of applying pointer casts if we have debug information avaliable:

- `llvm_union : SetupValue -> String -> SetupValue`

Given a pointer setup value, this attempts to select the named union branch and cast the type of the pointer. For this to work, debug symbols must be included; moreover, the process of correlating LLVM type information with information contained in debug symbols is a bit heuristic. If `llvm_union` cannot figure out how to cast a pointer, one can fall back on the more manual `llvm_cast_pointer` instead.

In the experimental Java verification implementation, the following functions can be used to state the equivalent of a combination of `llvm_points_to` and either `llvm_elem` or `llvm_field`.

- `jvm_elem_is : JVMValue -> Int -> JVMValue -> JVMSetup ()` specifies the value of an array element.
- `jvm_field_is : JVMValue -> String -> JVMValue -> JVMSetup ()` specifies the name of an object field.

Bitfields

SAW has experimental support for specifying `struct`s with bitfields, such as in the following example:

```
struct s {
  uint8_t x:1;
  uint8_t y:1;
};
```

Normally, a `struct` with two `uint8_t` fields would have an overall size of two bytes. However, because the `x` and `y` fields are declared with bitfield syntax, they are instead packed together into a single byte.

Because bitfields have somewhat unusual memory representations in LLVM, some special care is required to write SAW specifications involving bitfields. For this reason, there is a dedicated `llvm_points_to_bitfield` function for this purpose:

- `llvm_points_to_bitfield : SetupValue -> String -> SetupValue -> LLVMSetup ()`

The type of `llvm_points_to_bitfield` is similar that of `llvm_points_to`, except that it takes the name of a field within a bitfield as an additional argument. For example, here is how to assert that the `y` field in the `struct` example above should be `0`:

```
ss <- llvm_alloc (llvm_alias "struct.s");
llvm_points_to_bitfield ss "y" (llvm_term {{ 0 : [1] }});
```

Note that the type of the right-hand side value (`0`, in this example) must be a bitvector whose length is equal to the size of the field within the bitfield. In this example, the `y` field was declared as `y:1`, so `y`'s value must be of type `[1]`.

Note that the following specification is *not* equivalent to the one above:

```
ss <- llvm_alloc (llvm_alias "struct.s");
llvm_points_to (llvm_field ss "y") (llvm_term {{ 0 : [1] }});
```

`llvm_points_to` works quite differently from `llvm_points_to_bitfield` under the hood, so using `llvm_points_to` on bitfields will almost certainly not work as expected.

In order to use `llvm_points_to_bitfield`, one must also use the `enable_lax_loads_and_stores` command:

- `enable_lax_loads_and_stores: TopLevel ()`

Both `llvm_points_to_bitfield` and `enable_lax_loads_and_stores` are experimental commands, so these also require using `enable_experimental` before they can be used.

The `enable_lax_loads_and_stores` command relaxes some of SAW's assumptions about uninitialized memory, which is necessary to make `llvm_points_to_bitfield` work under the hood. For example, reading from uninitialized memory normally results in an error in SAW, but with `enable_lax_loads_and_stores`, such a read will instead return a symbolic value. At present, `enable_lax_loads_and_stores` only works with What4-based tactics (e.g., `w4_unint_z3`); using it with SBV-based tactics (e.g., `sbv_unint_z3`) will result in an error.

Note that SAW relies on LLVM debug metadata in order to determine which struct fields reside within a bitfield. As a result, you must pass `-g` to `clang` when compiling code involving bitfields in order for SAW to be able to reason about them.

Global variables

Mutable global variables that are accessed in a function must first be allocated by calling `llvm_alloc_global` on the name of the global.

- `llvm_alloc_global : String -> LLVMSetup ()`

This ensures that all global variables that might influence the function are accounted for explicitly in the specification: if `llvm_alloc_global` is used in the precondition, there must be a corresponding `llvm_points_to` in the postcondition describing the new state of that global. Otherwise, a specification might not fully capture the behavior of the function, potentially leading to unsoundness in the presence of compositional verification.

Immutable (i.e. `const`) global variables are allocated implicitly, and do not require a call to `llvm_alloc_global`.

Pointers to global variables or functions can be accessed with `llvm_global`:

- `llvm_global : String -> SetupValue`

Like the pointers returned by `llvm_alloc`, however, these aren't initialized at the beginning of symbolic – setting global variables may be unsound in the presence of compositional verification.

To understand the issues surrounding global variables, consider the following C code:

```
int x = 0;

int f(int y) {
  x = x + 1;
  return x + y;
```

```
}

int g(int z) {
  x = x + 2;
  return x + z;
}
```

One might initially write the following specifications for `f` and `g`:

```
m <- llvm_load_module "./test.bc";

f_spec <- llvm_verify m "f" [] true (do {
    y <- llvm_fresh_var "y" (llvm_int 32);
    llvm_execute_func [llvm_term y];
    llvm_return (llvm_term {{ 1 + y : [32] }});
}) abc;

g_spec <- llvm_llvm_verify m "g" [] true (do {
    z <- llvm_fresh_var "z" (llvm_int 32);
    llvm_execute_func [llvm_term z];
    llvm_return (llvm_term {{ 2 + z : [32] }});
}) abc;
```

If globals were always initialized at the beginning of verification, both of these specs would be provable. However, the results wouldn't truly be compositional. For instance, it's not the case that `f(g(z))== z + 3` for all `z`, because both `f` and `g` modify the global variable `x` in a way that crosses function boundaries.

To deal with this, we can use the following function:

- `llvm_global_initializer : String -> SetupValue` returns the value of the constant global initializer for the named global variable.

Given this function, the specifications for `f` and `g` can make this reliance on the initial value of `x` explicit:

```
m <- llvm_load_module "./test.bc";


let init_global name = do {
  llvm_alloc_global name;
  llvm_points_to (llvm_global name)
                 (llvm_global_initializer name);
};

f_spec <- llvm_verify m "f" [] true (do {
    y <- llvm_fresh_var "y" (llvm_int 32);
    init_global "x";
    llvm_precond {{ y < 2^^31 - 1 }};
    llvm_execute_func [llvm_term y];
    llvm_return (llvm_term {{ 1 + y : [32] }});
}) abc;
```

which initializes `x` to whatever it is initialized to in the C code at the beginning of verification. This specification is now safe for compositional verification: SAW won't use the specification `f_spec` unless it can determine that `x` still has its initial value at the point of a call to `f`. This specification also constrains `y` to prevent integer overflow resulting from the `x + y` expression in `f`.

Preconditions and Postconditions

Sometimes a function is only well-defined under certain conditions, or sometimes you may be interested in certain initial conditions that give rise to specific final conditions. For these cases, you can specify an arbitrary predicate as a precondition or post-condition, using any values in scope at the time.

- `llvm_precond : Term -> LLVMSetup ()`
- `llvm_postcond : Term -> LLVMSetup ()`
- `llvm_assert : Term -> LLVMSetup ()`
- `jvm_precond : Term -> JVMSetup ()`
- `jvm_postcond : Term -> JVMSetup ()`
- `jvm_assert : Term -> JVMSetup ()`

These commands take `Term` arguments, and therefore cannot describe the values of pointers. The "assert" variants will work in either pre- or post-conditions, and are useful when defining helper functions that, e.g., state datastructure invariants that make sense in both phases. The `llvm_equal` command states that two `SetupValue`s should be equal, and can be used in either the initial or the final state.

- `llvm_equal : SetupValue -> SetupValue -> LLVMSetup ()`

The use of `llvm_equal` can also sometimes lead to more efficient symbolic execution when the predicate of interest is an equality.

Assuming specifications

Normally, a `MethodSpec` is the result of both simulation and proof of the target code. However, in some cases, it can be useful to use a `MethodSpec` to specify some code that either doesn't exist or is hard to prove. The previously-mentioned `assume_unsat` tactic omits proof but does not prevent simulation of the function. To skip simulation altogether, one can use:

```
llvm_unsafe_assume_spec :
  LLVMModule -> String -> LLVMSetup () -> TopLevel CrucibleMethodSpec
```

Or, in the experimental JVM implementation:

```
jvm_unsafe_assume_spec :
  JavaClass -> String -> JVMSetup () -> TopLevel JVMMethodSpec
```

A Heap-Based Example

To tie all of the command descriptions from the previous sections together, consider the case of verifying the correctness of a C program that computes the dot product of two vectors, where the length and value of each vector are encapsulated together in a `struct`.

The dot product can be concisely specified in Cryptol as follows:

```
dotprod : {n, a} (fin n, fin a) => [n][a] -> [n][a] -> [a]
dotprod xs ys = sum (zip (*) xs ys)
```

To implement this in C, let's first consider the type of vectors:

```
typedef struct {
    uint32_t *elts;
    uint32_t size;
} vec_t;
```

This struct contains a pointer to an array of 32-bit elements, and a 32-bit value indicating how many elements that array has.

We can compute the dot product of two of these vectors with the following C code (which uses the size of the shorter vector if they differ in size).

```
uint32_t dotprod_struct(vec_t *x, vec_t *y) {
    uint32_t size = MIN(x->size, y->size);
    uint32_t res = 0;
    for(size_t i = 0; i < size; i++) {
        res += x->elts[i] * y->elts[i];
    }
    return res;
}
```

The entirety of this implementation can be found in the `examples/llvm/dotprod_struct.c` file in the `saw-script` repository.

To verify this program in SAW, it will be convenient to define a couple of utility functions (which are generally useful for many heap-manipulating programs). First, combining allocation and initialization to a specific value can make many scripts more concise:

```
let alloc_init ty v = do {
    p <- llvm_alloc ty;
    llvm_points_to p v;
    return p;
};
```

This creates a pointer `p` pointing to enough space to store type `ty`, and then indicates that the pointer points to value `v` (which should be of that same type).

A common case for allocation and initialization together is when the initial value should be entirely symbolic.

```
let ptr_to_fresh n ty = do {
    x <- llvm_fresh_var n ty;
    p <- alloc_init ty (llvm_term x);
    return (x, p);
};
```

This function returns the pointer just allocated along with the fresh symbolic value it points to.

Given these two utility functions, the `dotprod_struct` function can be specified as follows:

```
let dotprod_spec n = do {
    let nt = llvm_term {{ `n : [32] }};
    (xs, xsp) <- ptr_to_fresh "xs" (llvm_array n (llvm_int 32));
    (ys, ysp) <- ptr_to_fresh "ys" (llvm_array n (llvm_int 32));
    let xval = llvm_struct_value [ xsp, nt ];
    let yval = llvm_struct_value [ ysp, nt ];
    xp <- alloc_init (llvm_alias "struct.vec_t") xval;
    yp <- alloc_init (llvm_alias "struct.vec_t") yval;
    llvm_execute_func [xp, yp];
    llvm_return (llvm_term {{ dotprod xs ys }});
};
```

Any instantiation of this specification is for a specific vector length `n`, and assumes that both input vectors have that length. That length `n` automatically becomes a type variable in the subsequent Cryptol expressions, and the backtick operator is used to reify that type as a bit vector of length 32.

The entire script can be found in the `dotprod_struct-crucible.saw` file alongside `dotprod_struct.c`.

Running this script results in the following:

```
Loading file "dotprod_struct.saw"
Proof succeeded! dotprod_struct
Registering override for `dotprod_struct`
  variant `dotprod_struct`
Symbolic simulation completed with side conditions.
Proof succeeded! dotprod_wrap
```

Using Ghost State

In some cases, information relevant to verification is not directly present in the concrete state of the program being verified. This can happen for at least two reasons:

- When providing specifications for external functions, for which source code is not present. The external code may read and write global state that is not directly accessible from the code being verified.
- When the abstract specification of the program naturally uses a different representation for some data than the concrete implementation in the code being verified does.

One solution to these problems is the use of *ghost* state. This can be thought of as additional global state that is visible only to the verifier. Ghost state with a given name can be declared at the top level with the following function:

- `llvm_declare_ghost_state : String -> TopLevel Ghost`

Ghost state variables do not initially have any particluar type, and can store data of any type. Given an existing ghost variable the following function can be used to specify its value:

- `llvm_ghost_value : Ghost -> Term -> LLVMSetup ()`

Currently, this function can only be used for LLVM verification, though that will likely be generalized in the future. It can be used in either the pre state or the post state, to specify the value of ghost state either before or after the execution of the function, respectively.

An Extended Example

To tie together many of the concepts in this manual, we now present a non-trivial verification task in its entirety. All of the code for this example can be found in the `examples/salsa20` directory of the SAWScript repository.

Salsa20 Overview

Salsa20 is a stream cipher developed in 2005 by Daniel J. Bernstein, built on a pseudorandom function utilizing add-rotate-XOR (ARX) operations on 32-bit words[4]. Bernstein himself has provided several public domain implementations of the cipher, optimized for common machine architectures. For the mathematically inclined, his specification for the cipher can be found here.

The repository referenced above contains three implementations of the Salsa20 cipher: A reference Cryptol implementation (which we take as correct in this example), and two C implementations, one of which is

[4] https://en.wikipedia.org/wiki/Salsa20

from Bernstein himself. For this example, we focus on the second of these C implementations, which more closely matches the Cryptol implementation. Full verification of Bernstein's implementation is available in `examples/salsa20/djb`, for the interested. The code for this verification task can be found in the files named according to the pattern `examples/salsa20/(s|S)alsa20.*`.

Specifications

We now take on the actual verification task. This will be done in two stages: We first define some useful utility functions for constructing common patterns in the specifications for this type of program (i.e. one where the arguments to functions are modified in-place.) We then demonstrate how one might construct a specification for each of the functions in the Salsa20 implementation described above.

Utility Functions

We first define the function `alloc_init : LLVMType -> Term -> LLVMSetup SetupValue`.

`alloc_init ty v` returns a `SetupValue` consisting of a pointer to memory allocated and initialized to a value `v` of type `ty`. `alloc_init_readonly` does the same, except the memory allocated cannot be written to.

```
import "Salsa20.cry";

let alloc_init ty v = do {
    p <- llvm_alloc ty;
    llvm_points_to p (llvm_term v);
    return p;
};

let alloc_init_readonly ty v = do {
    p <- llvm_alloc_readonly ty;
    llvm_points_to p (llvm_term v);
    return p;
};
```

We now define `ptr_to_fresh : String -> LLVMType -> LLVMSetup (Term, SetupValue)`.

`ptr_to_fresh n ty` returns a pair `(x, p)` consisting of a fresh symbolic variable `x` of type `ty` and a pointer `p` to it. `n` specifies the name that SAW should use when printing `x`. `ptr_to_fresh_readonly` does the same, but returns a pointer to space that cannot be written to.

```
let ptr_to_fresh n ty = do {
    x <- llvm_fresh_var n ty;
    p <- alloc_init ty x;
    return (x, p);
};

let ptr_to_fresh_readonly n ty = do {
    x <- llvm_fresh_var n ty;
    p <- alloc_init_readonly ty x;
    return (x, p);
};
```

Finally, we define `oneptr_update_func : String -> LLVMType -> Term -> LLVMSetup ()`.

`oneptr_update_func n ty f` specifies the behavior of a function that takes a single pointer (with a printable name given by `n`) to memory containing a value of type `ty` and mutates the contents of that memory. The

specification asserts that the contents of this memory after execution are equal to the value given by the application of `f` to the value in that memory before execution.

```
let oneptr_update_func n ty f = do {
    (x, p) <- ptr_to_fresh n ty;
    llvm_execute_func [p];
    llvm_points_to p (llvm_term {{ f x }});
};
```

The `quarterround` operation

The C function we wish to verify has type `void s20_quarterround(uint32_t *y0, uint32_t *y1, uint32_t *y2, uint32_t *y3)`.

The function's specification generates four symbolic variables and pointers to them in the precondition/setup stage. The pointers are passed to the function during symbolic execution via `llvm_execute_func`. Finally, in the postcondition/return stage, the expected values are computed using the trusted Cryptol implementation and it is asserted that the pointers do in fact point to these expected values.

```
let quarterround_setup : LLVMSetup () = do {
    (y0, p0) <- ptr_to_fresh "y0" (llvm_int 32);
    (y1, p1) <- ptr_to_fresh "y1" (llvm_int 32);
    (y2, p2) <- ptr_to_fresh "y2" (llvm_int 32);
    (y3, p3) <- ptr_to_fresh "y3" (llvm_int 32);

    llvm_execute_func [p0, p1, p2, p3];

    let zs = {{ quarterround [y0,y1,y2,y3] }}; // from Salsa20.cry
    llvm_points_to p0 (llvm_term {{ zs@0 }});
    llvm_points_to p1 (llvm_term {{ zs@1 }});
    llvm_points_to p2 (llvm_term {{ zs@2 }});
    llvm_points_to p3 (llvm_term {{ zs@3 }});
};
```

Simple Updating Functions

The following functions can all have their specifications given by the utility function `oneptr_update_func` implemented above, so there isn't much to say about them.

```
let rowround_setup =
    oneptr_update_func "y" (llvm_array 16 (llvm_int 32)) {{ rowround }};

let columnround_setup =
    oneptr_update_func "x" (llvm_array 16 (llvm_int 32)) {{ columnround
       }};

let doubleround_setup =
    oneptr_update_func "x" (llvm_array 16 (llvm_int 32)) {{ doubleround
       }};

let salsa20_setup =
    oneptr_update_func "seq" (llvm_array 64 (llvm_int 8)) {{ Salsa20 }};
```

32-Bit Key Expansion

The next function of substantial behavior that we wish to verify has the following prototype:

```
void s20_expand32( uint8_t *k
                 , uint8_t n[static 16]
                 , uint8_t keystream[static 64]
                 )
```

This function's specification follows a similar pattern to that of `s20_quarterround`, though for extra assurance we can make sure that the function does not write to the memory pointed to by `k` or `n` using the utility `ptr_to_fresh_readonly`, as this function should only modify `keystream`. Besides this, we see the call to the trusted Cryptol implementation specialized to `a=2`, which does 32-bit key expansion (since the Cryptol implementation can also specialize to `a=1` for 16-bit keys). This specification can easily be changed to work with 16-bit keys.

```
let salsa20_expansion_32 = do {
    (k, pk) <- ptr_to_fresh_readonly "k" (llvm_array 32 (llvm_int 8));
    (n, pn) <- ptr_to_fresh_readonly "n" (llvm_array 16 (llvm_int 8));

    pks <- llvm_alloc (llvm_array 64 (llvm_int 8));

    llvm_execute_func [pk, pn, pks];

    let rks = {{ Salsa20_expansion`{a=2}(k, n) }};
    llvm_points_to pks (llvm_term rks);
};
```

32-bit Key Encryption

Finally, we write a specification for the encryption function itself, which has type

```
enum s20_status_t s20_crypt32( uint8_t *key
                             , uint8_t nonce[static 8]
                             , uint32_t si
                             , uint8_t *buf
                             , uint32_t buflen
                             )
```

As before, we can ensure this function does not modify the memory pointed to by `key` or `nonce`. We take `si`, the stream index, to be 0. The specification is parameterized on a number `n`, which corresponds to `buflen`. Finally, to deal with the fact that this function returns a status code, we simply specify that we expect a success (status code 0) as the return value in the postcondition stage of the specification.

```
let s20_encrypt32 n = do {
    (key, pkey) <- ptr_to_fresh_readonly "key" (llvm_array 32 (llvm_int
       8));
    (v, pv) <- ptr_to_fresh_readonly "nonce" (llvm_array 8 (llvm_int 8))
       ;
    (m, pm) <- ptr_to_fresh "buf" (llvm_array n (llvm_int 8));

    llvm_execute_func [ pkey
                      , pv
                      , llvm_term {{ 0 : [32] }}
                      , pm
                      , llvm_term {{ `n : [32] }}
                      ];
```

```
    llvm_points_to pm (llvm_term {{ Salsa20_encrypt (key, v, m) }});
    llvm_return (llvm_term {{ 0 : [32] }});
};
```

Verifying Everything

Finally, we can verify all of the functions. Notice the use of compositional verification and that path satisfiability checking is enabled for those functions with loops not bounded by explicit constants. Notice that we prove the top-level function for several sizes; this is due to the limitation that SAW can only operate on finite programs (while Salsa20 can operate on any input size.)

```
let main : TopLevel () = do {
    m      <- llvm_load_module "salsa20.bc";
    qr     <- llvm_verify m "s20_quarterround" []      false
       quarterround_setup   abc;
    rr     <- llvm_verify m "s20_rowround"     [qr]    false
       rowround_setup       abc;
    cr     <- llvm_verify m "s20_columnround"  [qr]    false
       columnround_setup    abc;
    dr     <- llvm_verify m "s20_doubleround"  [cr,rr] false
       doubleround_setup    abc;
    s20    <- llvm_verify m "s20_hash"         [dr]    false
       salsa20_setup        abc;
    s20e32 <- llvm_verify m "s20_expand32"     [s20]   true
       salsa20_expansion_32 abc;
    s20encrypt_63 <- llvm_verify m "s20_crypt32" [s20e32] true (
       s20_encrypt32 63) abc;
    s20encrypt_64 <- llvm_verify m "s20_crypt32" [s20e32] true (
       s20_encrypt32 64) abc;
    s20encrypt_65 <- llvm_verify m "s20_crypt32" [s20e32] true (
       s20_encrypt32 65) abc;

    print "Done!";
};
```

Extraction to the Coq theorem prover

In addition to the (semi-)automatic and compositional proof modes already discussed above, SAW has experimental support for exporting Cryptol and SAWCore values as terms to the Coq proof assistant[5]. This is intended to support more manual proof efforts for properties that go beyond what SAW can support (for example, proofs requiring induction) or for connecting to preexisting formalizations in Coq of useful algorithms (e.g. the fiat crypto library[6]).

This support consists of two related pieces. The first piece is a library of formalizations of the primitives underlying Cryptol and SAWCore and various supporting concepts that help bridge the conceptual gap between SAW and Coq. The second piece is a term translation that maps the syntactic forms of SAWCore onto corresponding concepts in Coq syntax, designed to dovetail with the concepts defined in the support library. SAWCore is a quite similar language to the core calculus underlying Coq, so much of this translation is quite straightforward; however, the languages are not exactly equivalent, and there are some tricky cases

[5] https://coq.inria.fr
[6] https://github.com/mit-plv/fiat-crypto

that mostly arise from Cryptol code that can only be partially supported. We will note these restrictions later in the manual.

We expect this extraction process to work with a fairly wide range of Coq versions, as we are not using bleeding-edge Coq features. It has been most fully tested with Coq version 8.13.2.

Support Library

In order to make use of SAW's extraction capabilities, one must first compile the support library using Coq so that the included definitions and theorems can be referenced by the extracted code. From the top directory of the SAW source tree, the source code for this support library can be found in the `saw-core-coq/coq` subdirectory. In this subdirectory you will find a `_CoqProject` and a `Makefile`. A simple `make` invocation should be enough to compile all the necessary files, assuming Coq is installed and `coqc` is available in the user's `PATH`. HTML documentation for the support library can also be generated by `make html` from the same directory.

Once the library is compiled, the recommended way to import it into your subsequent development is by adding the following lines to your `_CoqProject` file:

```
-Q <SAWDIR>/saw-core-coq/coq/generated/CryptolToCoq CryptolToCoq
-Q <SAWDIR>/saw-core-coq/coq/handwritten/CryptolToCoq CryptolToCoq
```

Here `<SAWDIR>` refers to the location on your system where the SAWScript source tree is checked out. This will add the relevant library files to the `CryptolToCoq` namespace, where the extraction process will expect to find them.

The support library for extraction is broken into two parts: those files which are handwritten, versus those that are automatically generated. The handwritten files are generally fairly readable and are reasonable for human inspection; they define most of the interesting pipe-fitting that allows Cryptol and SAWCore definitions to connect to corresponding Coq concepts. In particular the file `SAWCoreScaffolding.v` file defines most of the bindings of base types to Coq types, and the `SAWCoreVectorsAsCoqVectors.v` defines the core bitvector operations. The automatically generated files are direct translations of the SAWCore source files (`saw-core/prelude/Prelude.sawcore` and `cryptol-saw-core/saw/Cryptol.sawcore`) that correspond to the standard libraries for SAWCore and Cryptol, respectively.

The autogenerated files are intended to be kept up-to-date with changes in the corresponding `sawcore` files, and end users should not need to generate them. Nonetheless, if they are out of sync for some reason, these files may be regenerated using the `saw` executable by running `(cd saw-core-coq; saw saw/generate_scaffolding.saw)` from the top-level of the SAW source directory before compiling them with Coq as described above.

You may also note some additional files and concepts in the standard library, such as `CompM.v`, and a variety of lemmas and definitions related to it. These definitions are related to the "heapster" system, which form a separate use-case for the SAWCore to Coq translation. These definitions will not be used for code extracted from Cryptol.

Cryptol module extraction

There are several modes of use for the SAW to Coq term extraction facility, but the easiest to use is whole Cryptol module extraction. This will extract all the definitions in the given Cryptol module, together with it's transitive dependencies, into a single Coq module which can then be compiled and pulled into subsequence developments.

Suppose we have a Cryptol source file named `source.cry` and we want to generate a Coq file named `output.v`. We can accomplish this by running the following commands in saw (either directly from the `saw` command prompt, or via a script file)

```
enable_experimental;
write_coq_cryptol_module "source.cry" "output.v" [] [];
```

In this default mode, identifiers in the Cryptol source will be directly translated into identifiers in Coq. This may occasionally cause problems if source identifiers clash with Coq keywords or preexisting definitions. The third argument to `write_coq_cryptol_module` can be used to remap such names if necessary by giving a list of `(in,out)` pairs of names. The fourth argument is a list of source identifiers to skip translating, if desired. Authoritative online documentation for this command can be obtained directly from the `saw` executable via `:help write_coq_cryptol_module` after `enable_experimental`.

The resulting "output.v" file will have some of the usual hallmarks of computer-generated code; it will have poor formatting and, explicit parenthesis and fully-qualified names. Thankfully, once loaded into Coq, the Coq pretty-printer will do a much better job of rendering these terms in a somewhat human-readable way.

Proofs involving uninterpreted functions

It is possible to write a Cryptol module that references uninterpreted functions by using the `primitive` keyword to declare them in your Cryptol source. Primitive Cryptol declarations will be translated into Coq section variables; as usual in Coq, uses of these section variables will be translated into additional parameters to the definitions from outside the section. In this way, consumers of the translated module can instantiate the declared Cryptol functions with corresponding terms in subsequent Coq developments.

Although the Cryptol interpreter itself will not be able to compute with declared but undefined functions of this sort, they can be used both to provide specifications for functions to be verified with `llvm_verify` or `jvm_verify` and also for Coq extraction.

For example, if I write the following Cryptol source file:

```
primitive f : Integer -> Integer

g : Integer -> Bool
g x = f (f x) > 0
```

After extraction, the generated term `g` will have Coq type:

```
(Integer -> Integer) -> Integer -> Bool
```

Translation limitations and caveats

Translation from Cryptol to Coq has a number of fundamental limitations that must be addressed. The most severe of these is that Cryptol is a fully general-recursive language, and may exhibit runtime errors directly via calls to the `error` primitive, or via partial operations (such as indexing a sequence out-of-bounds). The internal language of Coq, by contrast, is a strongly-normalizing language of total functions. As such, our translation is unable to extract all Cryptol programs.

Recursive programs

The most severe of the currently limitations for our system is that the translation is unable to translate any recursive Cryptol program. Doing this would require us to attempt to find some termination argument for the recursive function sufficient to satisfy Coq; for now, no attempt is made to do so. if you attempt to extract a recursive function, SAW will produce an error about a "malformed term" with `Prelude.fix` as the head symbol.

Certain limited kinds of recursion are available via the `foldl` Cryptol primitive operation, which is translated directly into a fold operation in Coq. This is sufficient for many basic iterative algorithms.

Type coercions

Another limitation of the translation system is that Cryptol uses SMT solvers during its typechecking process and uses the results of solver proofs to justify some of its typing judgments. When extracting these terms to Coq, type equality coercions must be generated. Currently, we do not have a good way to transport the reasoning done inside Cryptol's typechecker into Coq, so we just supply a very simple `Ltac` tactic to discharge these coercions (see `solveUnsafeAssert` in `CryptolPrimitivesForSAWCoreExtra.v`). This tactic is able to discover simple coercions, but for anything nontrivial it may fail. The result will be a typechecking failure when compiling the generated code in Coq when the tactic fails. If you encounter this problem, it may be possible to enhance the `solveUnsafeAssert` tactic to cover your use case.

Error terms

A final caveat that is worth mentioning is that Cryptol can sometimes produce runtime errors. These can arise from explicit calls to the `error` primitive, or from partially defined operations (e.g., division by zero or sequence access out of bounds). Such instances are translated to occurrences of an unrealized Coq axiom named `error`. In order to avoid introducing an inconsistent environment, the `error` axiom is restricted to apply only to inhabited types. All the types arising from Cryptol programs are inhabited, so this is no problem in principle. However, collecting and passing this information around on the Coq side is a little tricky.

The main technical device we use here is the `Inhabited` type class; it simply asserts that a type has some distinguished inhabitant. We provide instances for the base types and type constructors arising from Cryptol, so the necessary instances ought to be automatically constructed when needed. However, polymorphic Cryptol functions add some complications, as type arguments must also come together with evidence that they are inhabited. The translation process takes care to add the necessary `Inhabited` arguments, so everything ought to work out. However, if Coq typechecking of generated code fails with errors about `Inhabited` class instances, it likely represents some problem with this aspect of the translation.