|galois|

Using the Software Analysis Workbench (SAW)

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

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). 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 SAW functional models. Models can be constructed from arbitrary Cryptol programs, and can typically be constructed 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 commmands.

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 in a variety of languages.

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.

`-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).

`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.

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
```

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 and Java (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. 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.

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 a function where the result type is one of a specific set of special types. These special types are *parameterized* (like the list type), and allow us to restrict command usage to specific contexts.

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 in a unit. 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_two
()
```

Here, the `print` commands run first, and then `print_two` gets the value returned by the second `print` command, namely `()`. Any command listed alone at the REPL, the top level in a script, or inside a `do` block is treated as implicitly having a `<-` that binds its result to an unnamed variable (that is, discards it).

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 c =
  do {
    print "Starting.";
    res <- c;
    print "Done.";
    return res;
  };

x <- run_with_message (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.
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]

head : {a} [a] -> a

tail : {a} [a] -> [a]

length : {a} [a] -> Int

null : {a} [a] -> Bool

nth : {a} [a] -> Int -> a

for : {m, a, b} [a] -> (a -> m b) -> m [b]
```

The `concat` function takes two lists and returns the concatenation of the two. The `head` function returns the first element of a list, and the `tail` function returns everything except the first element. The `length` function counts the number of elements in a list, and the `null` function indicates whether a list is empty (has zero elements). The `nth` function returns the element at the given position, with `nth l 0` being equivalent to `head l`. The `for` command 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, and `for` will ultimately return a list of all of the results produced by the command.

For interacting with the operating system, we have:

```
get_opt : Int -> String

exec : String -> [String] -> String -> TopLevel String

exit : Int -> TopLevel ()
```

The `get_opt` function 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. The `exec` command 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. Finally, the `exit` command 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. The `show` function 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. The `str_concat` function, which concatenates two `String` values, can also be useful in this case.

The `time` command runs any other `TopLevel` command and prints out the time it took to execute. If you want to use the time value later in the program, the `with_time` function returns both the original result of the timed command and the time taken to execute it (in milliseconds), without printing anything in the process.

```
show : {a} a -> String

str_concat : String -> String -> String

time : {a} TopLevel a -> TopLevel a

with_time : {a} TopLevel a -> TopLevel (Int, a)
```


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

Cyptol 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
```

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
Cryptol.ecPlus
  (Prelude.Vec 8 Prelude.Bool)
  (Cryptol.OpsSeq
     (Cryptol.TCNum 8)
     Prelude.Bool
     Cryptol.OpsBit)
  (Prelude.bvNat 8 34)
  (Prelude.bvNat 8 51)
```

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> print s
```

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

```
set_ascii : Bool -> TopLevel ()

set_base : Int -> TopLevel ()
```

The `set_ascii` command, 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). The `set_base` command, which supports any base from 2 through 36 (inclusive), prints all bit vectors in the given base.

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
```

Finally, 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))]
```

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 value in scope of SAWScript type `Bool` is visible in Cryptol expressions as a value of type `Bit`.
- Any value in scope of 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 value in scope of 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> let {{ f (x : [n]) = x + 1 }}
sawscript> print {{ f 2 }}
3
sawscript> print (f 2)

unbound variable: "f" (<stdin>:1:8)
```

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> 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 *size* types, and isomorphic to 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-size 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. To see how this works, consider the `cryptol_prims` function, of type `() -> CryptolModule`. This function returns a built-in module containing a collection of useful Cryptol definitions that are not available in the standard Cryptol Prelude.

The definitions in this module include (in Cryptol syntax):

```
trunc : {m, n} (fin m, fin n) => [m + n] -> [n]

uext : {m, n} (fin m, fin n) => [n] -> [m + n]

sgt : {n} (fin n) => [n] -> [n] -> Bit

sge : {n} (fin n) => [n] -> [n] -> Bit

slt : {n} (fin n) => [n] -> [n] -> Bit

sle : {n} (fin n) => [n] -> [n] -> Bit
```

These perform bit-vector operations of truncation (`trunc`), unsigned extension (`uext`), and signed comparison (`sgt`, `sge`, `slt`, and `sle`). These definitions are typically accessed through binding `cryptol_prims` to a local variable:

```
sawscript> set_base 16
sawscript> let m = cryptol_prims ()
sawscript> let x = {{ (m::trunc 0x23) : [4] }}
sawscript> print x
0x3
```

The 8-bit value `0x23` was truncated to a 4-bit value `0x3`.

Finally, a specific definition can be extracted from a `CryptolModule` more explicitly 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:

- as the definition of a function that can be unfolded,
- as a term of Boolean type (or a function returning a Boolean) that is an equality statement, and
- as a term of *equality type* with a body that encodes a proof that the equality in the type is valid.

Each of these forms is a `Term` of a different shape. In each case the term logically consists of two parts, each of which may contain variables (bound by enclosing lambda expressions). By thinking of the variables as holes that may match any sub-term, the two parts of each term can both be seen as *patterns*. 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:

```
\(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`. Applying this rule to 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
basic_ss : Simpset
cryptol_ss : () -> Simpset
```

The first 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. The `basic_ss` constant is a collection of rules that are useful in most proof scripts. The `cryptol_ss` value 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 add either a single rule or a list of rules to an existing `Simpset`.

```
addsimp' : Term -> Simpset -> Simpset

addsimps' : [Term] -> Simpset -> Simpset
```

Given a `Simpset`, the `rewrite` command applies it to an existing `Term` to produce a new `Term`.

```
rewrite : Simpset -> Term -> 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 `Terms` 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 { x0 = Prelude.Vec 8 Prelude.Bool;
    }
 in \(y::x0) ->
       Prelude.eq x0
         (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 `Term`s; 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

addsimps : [Theorem] -> Simpset -> 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 the shape described earlier, 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.

```
add_cryptol_eqs : [String] -> Simpset -> Simpset

add_prelude_defs : [String] -> Simpset -> Simpset

add_prelude_eqs : [String] -> Simpset -> Simpset
```

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. The `add_cryptol_eqs` function adds the terms of equality type with the given names to the given `Simpset`. The `add_prelude_defs` and `add_prelude_eqs` functions add definition unfolding rules and equality-typed terms, respectively, from the SAWCore `Prelude` module.

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
```

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
```

```
Prelude.bvNat 8 34
sawscript> t' <- define "t" t
sawscript> print_term t'
t
sawscript> let t'' = unfold_term ["t"] t'
sawscript> print_term t''
Prelude.bvNat 8 34
```

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
Prelude.bvNat 8 34
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

replace : Term -> Term -> Term -> TopLevel Term
```

The `beta_reduce_term` function 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`.

The `replace` function 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

term_tree_size : Term -> Int
```

The first, `term_size`, 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. The second, `term_tree_size`, 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 ()
```

```
type : Term -> Type
```

The `check_term` command checks that the internal structure of a `Term` is well-formed and that it passes all of the rules of the SAWCore type checker. The `type` function 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
```

```
read_aig : String -> TopLevel Term
```

```
read_bytes : String -> TopLevel Term
```

```
read_core : String -> TopLevel Term
```

The `parse_core` function parses a `String` containing a term in SAWCore syntax, returning a `Term`. The `read_core` command is similar, 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. The `read_aig` command returns a `Term` representation of an And-Inverter-Graph (AIG) file in AIGER format. The `read_bytes` command 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 SatResult -> Term -> TopLevel Theorem
```

```
sat_print : ProofScript SatResult -> Term -> TopLevel ()
```

The `prove_print` command 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.

The `sat_print` command is similar except that it looks for a *single* value for which the `Term` evaluates to `True` and prints out that value, returning nothing.

A similar command to `prove_print`, `prove_core`, can produce a `Theorem` from a string containing a SAWCore term.

```
prove_core : ProofScript SatResult -> String -> TopLevel Theorem
```

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 { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
      x1 = Cryptol.ePArith x0;
    }
 in \(x::Prelude.Vec 8 Prelude.Bool) ->
      Cryptol.ecEq x0
        (Cryptol.ePCmp x0)
        (Cryptol.ecPlus x0 x1 x x)
        (Cryptol.ecMul x0 x1 x
           (Prelude.bvNat 8 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`, `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 ()

print_goal_consts : ProofScript ()

print_goal_depth : Int -> ProofScript ()
```

```
print_goal_size : ProofScript ()
```

The `print_goal` tactic prints the entire goal in SAWCore syntax. The `print_goal_depth` is intended for especially large goals. It takes an integer argument, `n`, and prints the goal up to depth `n`. Any elided subterms are printed with a `...` notation. The `print_goal_consts` tactic prints a list of the names of subterms that are folded in the current goal, and `print_goal_size` prints the number of nodes in the DAG representation of the goal.

Rewriting in Proof Scripts

The `simplify` command works just like the `rewrite` command, except that it works in a `ProofScript` context and implicitly transforms the current (unnamed) goal rather than taking a `Term` as a parameter.

```
simplify : Simpset -> ProofScript ()
```

Other Transformations

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

```
beta_reduce_goal : ProofScript ()

unfolding : [String] -> ProofScript ()
```

The `beta_reduce_goal` tactic 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`.

The `unfolding` tactic works like `unfold_term` but on the current goal. Using `unfolding` is mostly valuable for proofs based entirely on rewriting, since default behavior for automated provers is to unfold everything before sending a goal to a prover. However, with Z3 and CVC4, it is possible to indicate that specific named subterms should be represented as uninterpreted functions.

```
unint_cvc4 : [String] -> ProofScript SatResult

unint_yices : [String] -> ProofScript SatResult

unint_z3 : [String] -> ProofScript SatResult
```

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).

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 SatResult

external_cnf_solver : String -> [String] -> ProofScript SatResult
```

The `external_aig_solver` function 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. Within this list, any element that consists of `%f` on its own is 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.

The `external_cnf_solver` function 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 SatResult

offline_cnf : String -> ProofScript SatResult

offline_extcore : String -> ProofScript SatResult

offline_smtlib2 : String -> ProofScript SatResult

offline_unint_smtlib2 : [String] -> String -> ProofScript SatResult
```

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.

Miscellaneous Tactics

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

```
assume_unsat : ProofScript SatResult

assume_valid : ProofScript ProofResult

quickcheck : Int -> ProofScript SatResult

trivial : ProofScript SatResult
```

The `assume_unsat` and `assume_valid` tactics indicate that the current goal should be considered unsatisfiable or valid, depending on whether the proof script is checking satisfiability or validity. At the moment, `java_verify` and `llvm_verify` run their proofs in the a satisfiability-checking context, so `assume_unsat` is currently the appropriate tactic. This is likely to change in the future.

The `quickcheck` tactic 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.

The `trivial` tactic states that the current goal should be trivially true (i.e., the constant `True` or a function that immediately returns `True`). It fails if that is not the case.

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

caseSatResult : {b} SatResult -> b -> (Term -> b) -> b
```

The `caseProofResult` function 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. The `caseSatResult` function 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

cec : AIG -> AIG -> TopLevel ProofResult

load_aig : String -> TopLevel AIG

save_aig : String -> AIG -> TopLevel ()

save_aig_as_cnf : String -> AIG -> TopLevel ()
```

The `bitblast` command 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. The `cec` command, for Combinational Equivalence Check, will compare two AIGs, returning a `ProofResult` representing whether the two are equivalent. The `load_aig` and `save_aig` commands work with external representations of AIG data structures in the AIGER format. Finally, `save_aig_as_cnf` will write an AIG out in CNF format for input into a standard SAT solver.

More Advanced Proof Scripts

TODO

`goal_apply` is missing `goal_assume` is missing `goal_eval` is missing `goal_eval_unint` is missing `goal_insert` is missing `goal_intro` is missing `goal_when` is missing `hoist_ifs` is missing `split_goal` is missing

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

In the previous section 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.

Now, 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.

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.

Loading Java code is slightly more complex, because of the more structured nature of Java packages. First, when running `saw`, two flags control where to look for classes. 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. However, the Java runtime and standard library (usually called `rt.jar`) is generally required for any non-trivial Java code, and can be installed in a wide variety of different locations. Therefore, for most Java analysis, you must provide a `-j` argument specifying where to find this file.

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.

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:

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

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

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

Because of its lack of maturity, it 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).

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.

The function `fresh_symbolic` is responsible for creating new variables in this context.

```
fresh_symbolic : String -> Type -> TopLevel Term
```

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

lambda : Term -> Term -> Term

lambdas : [Term] -> Term -> Term
```

The `abstract_symbolic` function is the simplest, but gives you the least control. It 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.

```
sawscript> x <- fresh_symbolic "x" {| [8] |}
sawscript> let t = {{ x + x }}
sawscript> print_term t
```

```
let { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
    }
 in Cryptol.ecPlus x0
      (Cryptol.ePArith x0)
      x
      x
sawscript> let f = abstract_symbolic t
sawscript> print_term f
let { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
    }
 in \(x::Prelude.Vec 8 Prelude.Bool) ->
      Cryptol.ecPlus x0
        (Cryptol.ePArith x0)
        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.

The building block for controlled binding is `lambda`. 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 { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
    }
 in \(x::Prelude.Vec 8 Prelude.Bool) ->
      Cryptol.ecPlus x0
        (Cryptol.ePArith x0)
        x
        x
```

For `Terms` with more than one symbolic variable, `lambdas` allows you to list the order in which they 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 { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
      x1 = Prelude.Vec 8 Prelude.Bool;
    }
 in Cryptol.ecPlus x0
      (Cryptol.ePArith x0)
      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 { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
```

```
        x1 = Prelude.Vec 8 Prelude.Bool;
      }
  in \(x1::x1) ->
        \(x2::x1) ->
          Cryptol.ecPlus x0
            (Cryptol.ePArith x0)
            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 { x0 = Cryptol.TCSeq (Cryptol.TCNum 8) Cryptol.TCBit;
        x1 = Prelude.Vec 8 Prelude.Bool;
      }
  in \(x2::x1) ->
        \(x1::x1) ->
          Cryptol.ecPlus x0
            (Cryptol.ePArith x0)
            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 with Crucible is controlled by the `crucible_llvm_verify` command.

```
crucible_llvm_verify :
  LLVMModule ->
  String ->
  [CrucibleMethodSpec] ->
  Bool ->
  CrucibleSetup () ->
  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 (which is separated from the specification itself, unlike `llvm_verify`). 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.

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

Now we describe how to construct a value of type `CrucibleSetup ()` (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 `CrucibleSetup` (or `JVMSetup`) type. The command `crucible_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 `crucible_execute_func`, though with slightly different meaning.

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 `crucible_fresh_var` (or `jvm_fresh_var`) command.

```
crucible_fresh_var : String -> LLVMType -> CrucibleSetup 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_array : Int -> LLVMType -> LLVMType
llvm_struct : String -> LLVMType
llvm_float : LLVMType
llvm_double : 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 `crucible_fresh_var` from the post state (i.e., after `crucible_execute_func`).

The SetupValue Type

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

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

Executing

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

```
crucible_execute_func : [SetupValue] -> CrucibleSetup ()
```

Return Values

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

```
crucible_return : SetupValue -> CrucibleSetup ()
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:

```
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 <- crucible_fresh_var "x" (llvm_int 32);
    y <- crucible_fresh_var "y" (llvm_int 32);
    crucible_execute_func [crucible_term x, crucible_term y];
    crucible_return (crucible_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 <- crucible_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 `crucible_llvm_verify` and `crucible_jvm_verify` functions returns values of type `CrucibleMethodSpec` and `JVMMethodSpec`, respectively. These values are opaque objects that internally contain both the information provided in the associated `JVMSetup` or `CrucibleSetup` 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 `crucible_points_to` and `crucible_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 <- crucible_fresh_var "x" (llvm_int 32);
    crucible_execute_func [crucible_term x];
    crucible_return (crucible_term {{ x + x : [32] }});
};
```


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

```
crucible_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 `crucible_alloc` command allows you to specify that a function expects a particular pointer to refer to an allocated region appropriate for a specific type.

```
crucible_alloc : LLVMType -> CrucibleSetup 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, `crucible_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
jvm_alloc_object : String -> JVMSetup JVMValue
```

The former specifies an array of the given concrete size, with elements of the given type. The latter 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):

```
crucible_fresh_pointer : LLVMType -> CrucibleSetup SetupValue
```

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

```
crucible_null : SetupValue
jvm_null : JVMValue
```

One final, slightly more obscure command is the following:

```
crucible_alloc_readonly : LLVMType -> CrucibleSetup SetupValue
```

This works like `crucible_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 `crucible_alloc`, regions allocated with `crucible_alloc_readonly` are allowed to alias other read-only regions.

Specifying Heap Values

Pointers returned by `crucible_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 `crucible_points_to` command.

```
crucible_points_to : SetupValue -> SetupValue -> CrucibleSetup ()
```

This command 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, `crucible_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.

```
crucible_points_to_untyped : SetupValue -> SetupValue -> CrucibleSetup
    ()
```

This function works like `crucible_points_to` but omits type checking. Rather than omitting type checking across the board, we introducted this additional function to make it clear when a type reinterpretation is intentional.

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. To access them piecewise, the `crucible_elem` function yields a pointer to an internal element of a compound value.

```
crucible_elem : SetupValue -> Int -> SetupValue
```

For arrays, the `Int` parameter is the array index. For `struct` values, it is the field index. For `struct` values, it can be more convenient to use field names. If debugging information is available in the bitcode file, the `crucible_field` function yields a pointer to a particular named field:

```
crucible_field : SetupValue -> String -> SetupValue
```

Either of these functions can be used with `crucible_points_to` to specify the value of a particular array element or `struct` field. Sometimes, however, it is more convenient to specify all array elemnts or field values at onces. The `crucible_array` and `crucible_struct` functions construct compound values from lists of element values.

```
crucible_array : [SetupValue] -> SetupValue
crucible_struct : [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 `crucible_fresh_var` and `crucible_elem` or `crucible_field` commands. However, the following function can simplify the common case where you want every element or field to have a fresh value.

```
crucible_fresh_expanded_val : LLVMType -> CrucibleSetup SetupValue
```

Finally, `crucible_struct` 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:

```
crucible_packed_struct : [SetupValue] -> SetupValue
```

In the experimental Java verification implementation, the following functions can be used to state the equivalent of a combination of `crucible_points_to` and either `crucible_elem` or `crucible_field`.

```
jvm_elem_is : JVMValue -> Int -> JVMValue -> JVMSetup ()
jvm_field_is : JVMValue -> String -> JVMValue -> JVMSetup ()
```

Global variables

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

```
crucible_global : String -> SetupValue
```

Like the pointers returned by `crucible_alloc`, however, these aren't initialized at the beginning of symbolic simulation. This is intentional – 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 <- crucible_llvm_verify m "f" [] true (do {
    y <- crucible_fresh_var "y" (llvm_int 32);
    crucible_execute_func [crucible_term y];
    crucible_return (crucible_term {{ 1 + y : [32] }});
}) abc;

g_spec <- crucible_llvm_verify m "g" [] true (do {
    z <- crucible_fresh_var "z" (llvm_int 32);
    crucible_execute_func [crucible_term z];
    crucible_return (crucible_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.

Instead, the specifications for `f` and `g` must make this reliance on the value of `x` explicit, e.g. one could write

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


let init_global name = do {
  crucible_points_to (crucible_global name)
                     (crucible_global_initializer name);
```

```
};

f_spec <- crucible_llvm_verify m "f" [] true (do {
    y <- crucible_fresh_var "y" (llvm_int 32);
    init_global "x";
    crucible_execute_func [crucible_term y];
    crucible_return (crucible_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`.

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 pre-condition or post-condition, using any values in scope at the time.

```
crucible_precond : Term -> CrucibleSetup ()
crucible_postcond : Term -> CrucibleSetup ()
```

Similar functions exist in the experimental JVM implementation:

```
jvm_precond : Term -> JVMSetup ()
jvm_postcond : Term -> JVMSetup ()
```

These two commands take `Term` arguments, and therefore cannot describe the values of pointers. The `crucible_equal` command states that two `SetupValue`s should be equal, and can be used in either the initial or the final state.

```
crucible_equal : SetupValue -> SetupValue -> CrucibleSetup ()
```

The use of `crucible_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 the assume_unsat tactic omits proof but does not preven simulation of the function. To skip simulation altogether, one can use:

```
crucible_llvm_unsafe_assume_spec :
  LLVMModule -> String -> CrucibleSetup () -> TopLevel
     CrucibleMethodSpec
```

Or, in the experimental JVM implementation:

```
crucible_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 <- crucible_alloc ty;
    crucible_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 <- crucible_fresh_var n ty;
    p <- alloc_init ty (crucible_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 = crucible_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 = crucible_struct [ xsp, nt ];
     let yval = crucible_struct [ ysp, nt ];
     xp <- alloc_init (llvm_struct "struct.vec_t") xval;
     yp <- alloc_init (llvm_struct "struct.vec_t") yval;
     crucible_execute_func [xp, yp];
     crucible_return (crucible_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`.

Using Ghost State

TODO: `crucible_declare_ghost_state` is missing TODO: `crucible_ghost_value` is missing

Notes

TODO: `enable_deprecated` is missing