Analyzing Java and LLVM Programs with SAW

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

Symbolic Execution

Analysis of Java and LLVM within SAWScript builds 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*, rather than concrete values, potentially containing free variables. Therefore, each symbolic execution corresponds to some set of possible concrete executions.

As a concrete example, consider the following C program, which 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 consider what it will do for *arbitrary* inputs, as well. 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 that, based on the value of its 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 to *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 with a different notion of what constitutes a value, and which executes *all* paths through the program instead of just one. Therefore, the execution process follows a similar process 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 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, 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 let's get into 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 is 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 directly 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:

```
java_extract : JavaClass -> String -> JavaSetup () -> TopLevel Term

llvm_extract : LLVMModule -> String -> LLVMSetup () -> TopLevel Term
```

The structure of these two 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 function or method to extract.

The third argument provides the ability to configure other aspects of the symbolic execution process. At the moment, only one option possible: pass in `java_pure` or `llvm_pure`, for Java and LLVM respectively, and the default extraction process is simply to set both arguments to fresh symbolic variables.

When the `..._extract` functions complete, they return a `Term` corresponding to the value returned by the function or method.

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 and not visible afterward is allowed).

Creating Symbolic Variables

The direct extraction process just discussed automatically introduces symbolic variables and then abstracts over them, yielding a function in the intermediate language of SAW 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 it usually convenient to specify it using Cryptol syntax using the type quoting brackets `{|` and `|}`. So, 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] |};
```

Monolithic Symbolic Execution

In many cases, the inputs and outputs of a function are more complex than supported by the direct extraction process just described. In that case, it's necessary to provide more information. In particular, following the structure described earlier, we need:

- For every pointer or object reference, how much storage space it refers to.
- A list of (potentially symbolic) values for some elements of the initial program state.
- A list of elements of the final program state to treat as outputs.

This capability is provided by the following built-in functions:

```
java_symexec : JavaClass ->
               String ->
               [(String, Term)] ->
               [String] ->
               Bool ->
               TopLevel Term

llvm_symexec : LLVMModule ->
               String ->
               [(String, Int)] ->
               [(String, Term, Int)] ->
               [(String, Int)] ->
               Bool ->
               TopLevel Term
```

For both functions, the first two arguments are the same as for the direct extraction functions from the previous section, identifying what code to execute. The final argument for both indicates whether or not to do branch satisfiability checking.

The remaining arguments are slightly different for the two functions, due to the differences between JVM and LLVM programs.

For `java_symexec`, the third argument, of type `[(String, Term)]`, provides information to configure the initial state of the program. Each `String` is an expression describing a component of the state, such as the name of a parameter, or a field of an object. Each `Term` provides the initial value of that component (which may include symbolic variables returned by `fresh_symbolic`).

The syntax of these expressions is as follows:

- Arguments to the method being analyzed can be referred to by name (if the `.class` file contains debugging information, as it will be if compiled with `javac -g`). The expression referring to the value of the argument `x` in the `max` example is simply `x`. For Java methods that do not have debugging information, arguments can be named positionally with `args[0]`, `args[1]` and so on. The name `this` refers to the same implicit parameter as the keyword in Java.
- The expression form `pkg.C.f` refers to the static field `f` of class `C` in package `pkg` (and deeper nesting of packages is allowed).
- The expression `return` refers to the return value of the method under analysis.
- For an expression `e` of object type, `e.f` refers to the instance field `f` of the object described by `e`.
- The value of an expression of array type is the entire contents of the array. At the moment, there is no way to refer to individual elements of an array.

The fourth argument of `java_symexec` is a list of expressions describing the elements of the state to return as outputs. The returned `Term` will be of tuple type if this list contains more than one element, or simply the value of the one state element if the list contains only one.

The `llvm_symexec` command uses an expression syntax similar to that for `java_symexec`, but not identical. The syntax is as follows:

- Arguments to the function being analyzed can be referred to by name (if the name is reflected in the LLVM code, as it generally is with code generated by Clang). The expression referring to the value of the argument `x` in the `max` example is simply `x`. For LLVM functions that do not have named arguments (such as those generated by the Rust compiler, for instance), arguments can be named positionally with `args[0]`, `args[1]` and so on.
- Global variables can be referred to directly by name.
- The expression `return` refers to the return value of the function under analysis.
- For any valid expression `e` referring to something with pointer type, the expression `*e` refers to the value pointed to. There are some differences between this and the equivalent expression in C, however. If, for instance, `e` has type `int *`, then `*e` will have type `int`. If `e` referred to a pointer to an array, the C expression `*e` would refer to the first element of that array. In SAWScript, it refers to the entire contents of the array, and there is no way to refer to individual elements of an array.
- For any valid expression `e` referring to a pointer to a `struct`, the expression `e->n`, for some natural number `n`, refers to the `n`th field of that `struct`. Unlike the `struct` type in C, the LLVM `struct` type does not have named fields, so fields are described positionally.
- For any valid expression `e` referring to a `struct` (directly, not via pointer), the expression `e.n`, for some natural number `n`, refers to the `n`th field of that `struct`. This is particularly useful for fields of nested structs, even if the outer struct is passed by pointer.

In addition to the different expression language, the arguments are similar but not identical. The third argument, of type `[(String, Int)]`, indicates for each pointer how many elements it points to. Before execution, SAW will allocate the given number of elements of the static type of the given expression. The strings given here should be expressions identifying *pointers* rather than the values of those pointers.

The fourth argument, of type `[(String, Term, Int)]` indicates the initial values to write to the program state before execution. The elements of this list should include *value* expressions. For example, if a function has an argument named `p` of type `int *`, the allocation list might contain the element `("p", 1)`, whereas the initial values list might contain the element `("*p", v, 1)`, for some value `v`. The `Int` portion of each tuple indicates how large the term is: for a term with Cryptol type `[n]a`, the `Int` argument should be `n`. In the future we expect this value to be inferred.

Finally, the fifth argument, of type `[(String, Int)]` indicates the elements to read from the final state. For each entry, the `String` should be a value expression, and the `Int` parameter indicates how many elements to read. The number of elements does not need to be the same as the number of elements allocated or written in the initial state. However, reading past the end of an object or reading a location that has not been initialized will lead to an error.

Examples

The following code is a complete example of using the `java_symexec` function.

```
// show that add(x,y) == add(y,x) for all x and y
cadd <- java_load_class "Add";
x <- fresh_symbolic "x" {| [32] |};
y <- fresh_symbolic "y" {| [32] |};
ja <- java_symexec cadd "add" [("x", x), ("y", y)] ["return"] true;
print_term ja;
ja' <- abstract_symbolic ja;
prove_print abc {{ \a b -> ja' a b == ja' b a }};
print "Done.";
```

It first loads the `Add` class and creates two 32-bit symbolic variables, `x` and `y`. It then symbolically execute the `add` method with the symbolic variables just created passed in as its two arguments, and returns the symbolic expression denoting the method's return value.

Once the script has a `Term` in hand (the variable `ja`), it prints it out and then translates the version containing symbolic variables into a function that takes concrete values for those variables as arguments. Finally, it proves that the resulting function is commutative.

Running this script through `saw` gives the following output:

```
% saw -j <path to>rt.jar java_symexec.saw
Loading module Cryptol
Loading file "java_symexec.saw"
let { x0 = Cryptol.ty
             (Cryptol.TCSeq (Cryptol.TCNum 32) Cryptol.TCBit);
    }
 in Prelude.bvAdd 32 x y
Valid
Done.
```

Limitations

Although the `symexec` functions are more flexible than the `extract` functions, they also have some limitations and assumptions.

- When allocating memory for objects or arrays, each allocation is done independently. Therefore, there is currently no way to create data structures that share sub-structures. No aliasing is possible. Therefore, it is important to take into account that any proofs performed on the results of symbolic execution will not necessarily reflect the behavior of the code being analyzed if it is called in a context where its inputs involve aliasing or overlapping memory regions.
- The sizes and pointer relationships between objects in the heap must be specified before doing symbolic execution. Therefore, the results may not reflect the behavior of the code when called with, for example, arrays of different sizes.
- In Java, any variables of class type are initialized to refer to an object of that specific, statically-declared type, while in general they may refer to objects of subtypes of that type. Therefore, the code under

analysis may behave differently when given parameters of more specific types.

Specification-Based Verification

The built-in functions described so far work by extracting models of code which 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 Java or LLVM code and a specification, however, sometimes a more declarative approach is convenient. The following two functions allow for combined model extraction and verification.

```
java_verify : JavaClass ->
              String ->
              [JavaMethodSpec] ->
              JavaSetup () ->
              TopLevel JavaMethodSpec

llvm_verify : LLVMModule ->
              String ->
              [LLVMMethodSpec] ->
              LLVMSetup () ->
              TopLevel LLVMMethodSpec
```

Like all of the functions for Java and LLVM analysis, the first two parameters indicate what code to analyze. The third parameter is used for compositional verification, as described in the next section. For now, the empty list works fine. The final parameter describes the specification of the code to be analyzed, built out of commands of type `JavaSetup` or `LLVMSetup`. In most cases, this parameter will be a `do` block containing a sequence of commands of this type. Specifications are slightly different between Java and LLVM, but make use of largely the same set of concepts.

- Several commands are available to configure the contents of the initial state, before symbolic execution.
- Several commands are available to describe what to check of the final state, after symbolic execution.
- One final command describes how to prove that the code under analysis matches the specification.

The following sections describe the details of configuring initial states, stating the expected properties of the final state, and proving that the final state actually satisfies those properties.

Configuring the Initial State

The first step in configuring the initial state is to specify which program variables are important, and to specify their types more precisely. The symbolic execution system currently expects the layout of memory before symbolic execution to be completely specified. As in `llvm_symexec`, SAW needs information about how much space every pointer or reference variable points to. And, with one exception, SAW assumes that every pointer points to a distinct region of memory.

Because of this structure, the are separate functions used to describe variables with values of base types versus variables of pointer type.

For simple integer values, use `java_var` or `llvm_var`.

```
java_var : String -> JavaType -> JavaSetup Term
llvm_var : String -> LLVMType -> LLVMSetup Term
```

These functions both take a variable name and a type. The variable names use the same syntax described earlier for `java_symexec` and `llvm_symexec`. The types are built up using the following functions:

```
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

llvm_int : Int -> LLVMType
llvm_array : Int -> LLVMType -> LLVMType
llvm_struct : String -> LLVMType
llvm_float : LLVMType
llvm_double : LLVMType
```

Most of these types are straightforward mappings to the standard Java and LLVM types. The one key difference is that arrays must have a fixed, concrete size. Therefore, all analysis results are 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.

The `Term` returned by `java_var` and `llvm_var` is a representation of the *initial value* of the variable being declared. It can be used in any later expression.

While `java_var` and `llvm_var` declare elements of the program state that have values representable in the logic of SAW, pointers and references exist only inside the simulator: they are not representable before or after symbolic execution. Because of this, different functions are available to declare variables of pointer or reference type.

```
java_class_var : String -> JavaType -> JavaSetup ()

llvm_ptr : String -> LLVMType -> LLVMSetup ()
```

For both functions, the first argument is the name of the state element that they refer to. For `java_class_var`, the second argument is the type of the object, which should always be the result of the `java_class` function called with an appropriate class name. Arrays in Java are treated as if they were values, rather than references, since their values are directly representable in SAWCore. For `llvm_ptr`, the second argument is the type of the value pointed to. Both functions return no useful value (the unit type `()`), since the values of pointers are not meaningful in SAWCore. In LLVM, arrays are represented as pointers, and therefore the pointer and the value pointed to must be declared separately:

```
llvm_ptr "a" (llvm_array 10 (llvm_int 32));
a <- llvm_var "*a" (llvm_array 10 (llvm_int 32));
```

The `java_assert` and `llvm_assert` functions take a `Term` of boolean type as an argument which states a condition that must be true in the initial state, before the function under analysis executes. The term can refer to the initial values of any declared program variables.

When the condition required of an initial state is that a variable always has a specific, concrete value, optimized forms of these functions are available. The `java_assert_eq` and `llvm_assert_eq` functions take two arguments: an expression naming a location in the program state, and a `Term` representing an initial value. These functions work as destructive updates in the state of the symbolic simulator, and can make branch conditions more likely to reduce to constants. This means that, although `..._assert` and `..._assert_eq` functions can be used to make semantically-equivalent statements, using the latter can make symbolic termination more likely.

Finally, although the default configuration of the symbolic simulators in SAW is to make every pointer or reference refer to a fresh region of memory separate from all other pointers, it is possible to override this behavior for Java programs by declaring that a set references can alias each other.

```
java_may_alias : [String] -> JavaSetup ()
```

This function takes a list of names referring to references, and declares that any element of this set may (or may not) alias any other. Because this is a may-alias relationship, the verification process involves a separate proof for each possible aliasing configuration. At the moment, LLVM heaps must be completely disjoint.

Checking the Final State

The simplest statement about the expected final state of the method or function under analysis is to declare what value it should return (generally as a function of the variables declared as part of the initial state).

```
java_return : Term -> JavaSetup ()

llvm_return : Term -> LLVMSetup ()
```

For side effects, the following two functions allow declaration of the final expected value of that the program state should contain when execution finishes.

```
java_ensure_eq : String -> Term -> JavaSetup ()

llvm_ensure_eq : String -> Term -> LLVMSetup ()
```

For the most part, these two functions may refer to the same set of variables used to set up the initial state. However, for functions that return pointers or objects, the special name `return` is also available. It can be used in `java_class_var` and `llvm_ptr` calls, to declare the more specific object or array type of a return value, and in the `..._ensure_eq` function to declare the associated values. For LLVM arrays, typical use is like this:

```
llvm_ensure_eq "*return" v;
```

The `return` expression is also useful for fields of returned objects or structs:

```
java_ensure_eq "return.f" v;

llvm_ensure_eq "return->0" v;
```

Running Proofs

Once the constraints on the initial and final states have been declared, what remains is to prove that the code under analysis actually meets these specifications. The goal of SAW is to automate this proof as much as possible, but some degree of input into the proof process is sometimes necessary, and can be provided with the following functions:

```
java_verify_tactic : ProofScript SatResult -> JavaSetup ()

llvm_verify_tactic : ProofScript SatResult -> LLVMSetup ()
```

Both of these take a proof script as an argument, which specifies how to perform the proof. If the setup block does not call one of these functions, SAW will print a warning message and skip the proof (which can sometimes be a useful behavior during debugging, or in compositional verification as described later). The next section describes the structure of proof scripts, which are also useful with the standalone `proof` and `sat` functions within SAWScript, regardless of whether Java or LLVM code is involved.

Proof Scripts

The simplest proof scripts just indicate which automated prover to use. The `ProofScript` values `abc` and `z3` select the ABC and Z3 theorem provers, respectively, and are typically good choices. In addition to these, the `boolector`, `cvc4`, `mathsat`, and `yices` provers are available.

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.

Rewriting

The basic concept involved in rewriting `Terms` is that of a `Simpset`, which includes a collection of rewrite rules. A few basic, pre-defined values of this type 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 will be useful in most proof scripts. The `cryptol_ss` value includes a collection of Cryptol-specific rules, including rules to 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. When comparing Cryptol to Cryptol code, leaving these abstractions in place can be most appropriate, 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 : Theorem -> Simpset -> Simpset
addsimp' : Term -> Simpset -> Simpset
addsimps : [Theorem] -> Simpset -> Simpset
addsimps' : [Term] -> Simpset -> Simpset
```

The functions that take a `Theorem` type work with rewrite rules that are proved to be correct. The `prove_print` function returns a `Theorem` when successful, and any such theorem that is an equality statement can be used as a rewrite rule. The behavior is that any instance of the left-hand side of this equality in the goal is replaced with the right-hand side of the equality.

The functions above that take `Term` arguments work with terms of the same shape, but which haven't been proven to be correct. When using these functions, the soundness of the proof process depends on the correctness of these rules as a side condition.

Finally, there are some built-in rules that are not included in either `basic_ss` and `cryptol_ss` because they are not always beneficial, but can sometimes be helpful or essential.

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

A rewrite rule in SAW can be specified in multiple ways, the third due to the dependent type system used in SAWCore:

- as the definition of functions 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* whose body encodes a proof that the equality in the type is valid.

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.

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 with "opaque constants". Each of these is a named subterm in a SAWCore term that can be treated as "folded", in which case it is just an opaque name (essentially an uninterpreted function), or "unfolded", in which case it looks just like any other subterm. The `unfolding` tactic unfolds any instances of opaque constants with the any of the given names.

Using `unfolding` is mostly useful for proofs based entirely on rewriting, since default behavior for automated provers is to unfold all opaque constants before sending a goal to a prover. However, with Z3 and CVC4, it is possible to indicate that specific constants should be left folded (uninterpreted).

```
unint_cvc4 : [String] -> ProofScript SatResult
unint_z3 : [String] -> ProofScript SatResult
```

The list of `String` arguments in these two cases indicates the names of the constants 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 below.

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 parameters to pass to that executable. Within this list, any element that consists simply of `%f` 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
```

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.

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 opaque constants that are folded in the current goal, and `print_goal_size` prints the number of nodes in the DAG representation of the goal.

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.

Compositional Verification

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

The `java_verify` and `llvm_verify` functions returns values of type `JavaMethodSpec` and `LLVMMethodSpec`, respectively. These values are opaque objects that internally contain all of the information provided in the associated `JavaSetup` or `LLVMSetup` blocks, along with 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 `..._assert` and `..._assert_eq` statements in the specification are satisfied.
- Check that any aliasing is compatible with the aliasing restricted stated with `java_may_alias`, for Java programs.
- Check that all classes required by the target method have already been initialized, for Java programs.
- Update the simulator state as described in the specification.

Normally, a `MethodSpec` comes as the result of both simulation and proof of the target code. However, in some cases, it can be useful to use it to specify some code that either doesn't exist or is hard to prove. In those cases, the `java_no_simulate` or `llvm_no_simulate` functions can be used to indicate not to even try to simulate the code being specified, and instead return a `MethodSpec` that is assumed to be correct.

The default behavior of `java_verify` disallows allocation within the method being analyzed. This restriction makes it possible to reason about all possible effects of a method, since only effects specified with `java_ensure_eq` or `java_modify` are allowed. For many cryptographic applications, this behavior is ideal, because it is important to know whether, for instance, temporary variables storing key material have been cleared after use. Garbage on the heap that has been collected but not cleared could let confidential information leak.

If allocation is not a concern in a particular application, the `java_allow_alloc` function makes allocation within legal within the method being specified.

Controlling Symbolic Execution

One other set of commands is available to control the symbolic execution process. These control the use of satisfiability checking to determine whether both paths are feasible when encountering branches in the program, which is particularly relevant for branches controlling the iteration of loops.

```
java_sat_branches : Bool -> JavaSetup ()
llvm_sat_branches : Bool -> LLVMSetup ()
```

The `Bool` parameter has the same effect as the `Bool` parameter passed to `java_symexec` and `llvm_symexec`.