|galois|

SAWScript

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

Abstract

We introduce the SAWScript language, aiming to provide a programmable interface to Galois's formal verification technologies, covering the Cryptol, Java, and LLVM languages. We use various simple programs as examples, taken from the domain of cryptography. Our proofs use SAWScript to equivalence check functions written in one language against their counterparts in another, or reference implementations in one language against more efficient production implementations in the same language.

Contents

Introduction

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

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

This tutorial introduces the details of the language by walking through several examples, and showing how simple verification tasks can be described.

Example: Find First Set

As a first example, we consider equivalence checking different implementations of the POSIX `ffs` function, which identifies the position of the first `1` bit in a word. The function takes an integer as input, treated as a vector of bits, and returns another integer which indicates the index of the first bit set. This function can be implemented in several ways with different performance and code clarity tradeoffs, and we would like to show those different implementations are equivalent.

Reference Implementation

One simple implementation takes the form of a loop with an index initialized to zero, and a mask initialized to have the least significant bit set. On each iteration, we increment the index, and shift the mask to the left. Then we can use a bitwise "and" operation to test the bit at the index indicated by the index variable. The following C code (which is also in the `code/ffs.c` file accompanying this tutorial) uses this approach.

```
uint32_t ffs_ref(uint32_t word) {
    int i = 0;
    if(!word)
        return 0;
    for(int cnt = 0; cnt < 32; cnt++)
        if(((1 << i++) & word) != 0)
            return i;
    return 0; // notreached
}
```

This implementation is relatively straightforward, and a proficient C programmer would probably have little difficulty believing its correctness. However, the number of branches taken during execution could be as many as 32, depending on the input value. It's possible to implement the same algorithm with significantly fewer branches, and no backward branches.

Optimized Implementation

An alternative implementation, taken by the following program (also in `code/ffs.c`), treats the bits of the input word in chunks, allowing sequences of zero bits to be skipped over more quickly.

```
uint32_t ffs_imp(uint32_t i) {
    char n = 1;
    if (!(i & 0xffff)) { n += 16; i >>= 16; }
    if (!(i & 0x00ff)) { n += 8;  i >>= 8; }
    if (!(i & 0x000f)) { n += 4;  i >>= 4; }
    if (!(i & 0x0003)) { n += 2;  i >>= 2; }
```

```
    return (i) ? (n+((i+1) & 0x01)) : 0;
}
```

However, this code is much less obvious than the previous implementation. If it is correct, we would like to use it, since it has the potential to be faster. But how do we gain confidence that it calculates the same results as the original program?

SAWScript allows us to state this problem concisely, and to quickly and automatically prove the equivalence of these two functions for all possible inputs.

Buggy Implementation

Finally, a buggy implementation which is correct on all but one possible input (also in `code/ffs.c`). Although contrived, this program represents a case where traditional testing – as opposed to verification – is unlikely to be helpful.

```
  char n = 1;
  int s1 = !(i & 0xffff) << 4;
  n += s1; i >>= s1;
  int s2 = !(i & 0x00ff) << 3;
  n += s2;  i >>= s2;
```

SAWScript allows us to quickly identify an input exhibiting the bug.

Generating LLVM Code

The SAWScript interpreter can analyze LLVM code, but most programs are originally written in a higher-level language such as C, as in our example. Therefore, the C code must be translated to LLVM, using something like the following command:

```
# clang -c -emit-llvm -o ffs.bc ffs.c
```

This command, and following command examples in this tutorial, can be run from the `code` directory accompanying the tutorial document. A `Makefile` also exists in that directory, providing quick shortcuts for tasks like this. For instance, we can get the same effect as the previous command by doing:

```
# make ffs.bc
```

Equivalence Proof

We now show how to use SAWScript to prove the equivalence of the reference and implementation versions of the FFS algorithm, and exhibit the bug in the buggy implementation.

A SAWScript program is typically structured as a sequence of commands, potentially along with definitions of functions that abstract over commonly-used combinations of commands.

The following script (in `code/ffs_llvm.saw`) is sufficient to automatically prove the equivalence of the `ffs_ref` and `ffs_imp` functions, and identify the bug in `ffs_bug`.

```
print "Extracting reference term";
l <- llvm_load_module "ffs.bc";
ffs_ref <- llvm_extract l "ffs_ref" llvm_pure;
```

```
print "Extracting implementation term";
ffs_imp <- llvm_extract l "ffs_imp" llvm_pure;

print "Extracting buggy term";
ffs_bug <- llvm_extract l "ffs_bug" llvm_pure;

print "Proving equivalence";
let thm1 = {{ \x -> ffs_ref x == ffs_imp x }};
result <- prove abc thm1;
print result;

print "Finding bug via sat search";
let thm2 = {{ \x -> ffs_ref x != ffs_bug x }};
result <- sat abc thm2;
print result;

print "Finding bug via failed proof";
let thm3 = {{ \x -> ffs_ref x == ffs_bug x }};
result <- prove abc thm3;
print result;

print "Done.";
```

In this script, the `print` commands simply display text for the user. The `llvm_extract` command instructs the SAWScript interpreter to perform symbolic simulation of the given C function (e.g., `ffs_ref`) from a given bitcode file (e.g., `ffs.bc`), and return a term representing the semantics of the function. The final argument, `llvm_pure` indicates that the function to analyze is a "pure" function, which computes a scalar return value entirely as a function of its scalar parameters.

The `let` statement then constructs a new term corresponding to the assertion of equality between two existing terms. Arbitrary Cryptol expressions can be embedded within SAWScript; to distinguish Cryptol code from SAWScript commands, the Cryptol code is placed within double brackets `{{` and `}}`.

The `prove_print` command can verify the validity of such an assertion, and print out the results of verification. The `abc` parameter indicates what theorem prover to use; SAWScript offers support for many other SAT and SMT solvers as well as user definable simplification tactics.

If the `saw` executable is in your PATH, you can run the script above with

```
# saw ffs_llvm.saw
```

producing the output

```
Loading module Cryptol
Loading file "ffs_llvm.saw"
Extracting reference term
Extracting implementation term
Extracting buggy term
Proving equivalence
Valid
Finding bug via sat search
Sat: 1052688
Finding bug via failed proof
```

```
Invalid: 1052688
Done.
```

Note that `0x101010 = 1052688`, and so both explicitly searching for an input exhibiting the bug (with `sat`) and attempting to prove the false equivalence (with `prove`) exhibit the bug. Symmetrically, we could use `sat` to prove the equivalence of `ffs_ref` and `ffs_imp`, by checking that the corresponding disequality is unsatisfiable. Indeed, this exactly what happens behind the scenes: `prove abc <goal>` is essentially `not (sat abc (not <goal>))`.

Cross-Language Proofs

We can implement the FFS algorithm in Java with code almost identical to the C version.

The reference version (in `code/FFS.java`) uses a loop, like the C version:

```
    static int ffs_ref(int word) {
        int i = 0;
        if(word == 0)
            return 0;
        for(int cnt = 0; cnt < 32; cnt++)
            if(((1 << i++) & word) != 0)
                return i;
        return 0;
    }
```

And the efficient implementation uses a fixed sequence of masking and shifting operations:

```
    static int ffs_imp(int i) {
        byte n = 1;
        if ((i & 0xffff) == 0) { n += 16; i >>= 16; }
        if ((i & 0x00ff) == 0) { n +=  8; i >>=  8; }
        if ((i & 0x000f) == 0) { n +=  4; i >>=  4; }
        if ((i & 0x0003) == 0) { n +=  2; i >>=  2; }
        if (i != 0) { return (n+((i+1) & 0x01)); } else { return 0; }
    }
```

Although in this case we can look at the C and Java code and see that they perform almost identical operations, the low-level operators available in C and Java do differ somewhat. Therefore, it would be nice to be able to gain confidence that they do, indeed, perform the same operation.

We can do this with a process very similar to that used to compare the reference and implementation versions of the algorithm in a single language.

First, we compile the Java code to a JVM class file.

```
# javac -g FFS.java
```

Now we can do the proof both within and across languages (from `code/ffs_compare.saw`):

```
import "ffs.cry";
j <- java_load_class "FFS";
java_ffs_ref <- java_extract j "ffs_ref" java_pure;
java_ffs_imp <- java_extract j "ffs_imp" java_pure;
```

```
l <- llvm_load_module "ffs.bc";
c_ffs_ref <- llvm_extract l "ffs_ref" llvm_pure;
c_ffs_imp <- llvm_extract l "ffs_imp" llvm_pure;

print "java ref <-> java imp";
let thm1 = {{ \x -> java_ffs_ref x == java_ffs_imp x }};
prove_print abc thm1;

print "c ref <-> c imp";
let thm2 = {{ \x -> c_ffs_ref x == c_ffs_imp x }};
prove_print abc thm2;

print "java imp <-> c imp";
let thm3 = {{ \x -> java_ffs_imp x == c_ffs_imp x }};
prove_print abc thm3;

print "cryptol imp <-> c imp";
let thm4 = {{ \x -> ffs_imp x == c_ffs_imp x }};
prove_print abc thm4;

print "cryptol imp <-> cryptol ref";
let thm5 = {{ \x -> ffs_imp x == ffs_ref x }};
prove_print abc thm5;

print "Done.";
```

We can run this with the `-j` flag to tell it where to find the Java standard libraries:

```
# saw -j <path to rt.jar or classes.jar from JDK> ffs_compare.saw
```

If you're using a Sun Java, you can find the standard libraries JAR by grepping the output of `java -v`:

```
# java -v 2>&1 | grep Opened
```

Using SMT-Lib Solvers

The examples presented so far have used the internal proof system provided by SAWScript, based primarily on a version of the ABC tool from UC Berkeley linked into the `saw` executable. However, there is internal support for other proof tools – such as Yices and CVC4 as illustrated in the next example – and more general support for exporting models representing theorems as goals in the SMT-Lib language. These exported goals can then be solved using an external SMT solver.

Consider the following C file:

```c
int double_ref(int x) {
    return x * 2;
}

int double_imp(int x) {
    return x << 1;
}
```

In this trivial example, an integer can be doubled either using multiplication or shifting. The following

SAWScript program (`code/double.saw`) verifies that the two are equivalent using both internal ABC, Yices, and CVC4 modes, and by exporting an SMT-Lib theorem to be checked later, by an external SAT solver.

```
l <- llvm_load_module "double.bc";
double_imp <- llvm_extract l "double_imp" llvm_pure;
double_ref <- llvm_extract l "double_ref" llvm_pure;
let thm = {{ \x -> double_ref x == double_imp x }};

r <- prove abc thm;
print r;

r <- prove yices thm;
print r;

r <- prove cvc4 thm;
print r;

let thm_neg = {{ \x -> ~(thm x) }};
write_smtlib2 "double.smt2" thm_neg;

print "Done.";
```

The new primitives introduced here are the tilde operator, `~`, which constructs the logical negation of a term, and `write_smtlib2`, which writes a term as a proof obligation in SMT-Lib version 2 format. Because SMT solvers are satisfiability solvers, negating the input term allows us to interpret a result of "unsatisfiable" from the solver as an indication of the validity of the term. The `prove` primitive does this automatically, but for flexibility the `write_smtlib2` primitive passes the given term through unchanged, because it might be used for either satisfiability or validity checking.

The SMT-Lib export capabilities in SAWScript make use of the Haskell SBV package, and support ABC, Boolector, CVC4, MathSAT, Yices, and Z3.

External SAT Solvers

In addition to the `abc`, `cvc4`, and `yices` proof tactics used above, SAWScript can also invoke arbitrary external SAT solvers that that read CNF files and produce results according to the SAT competition input and output conventions, using the `external_cnf_solver` tactic. For example, you can use PicoSAT to prove the theorem `thm` from the last example, with the following commands:

```
let picosat = external_cnf_solver "picosat" ["%f"];
prove_print picosat thm;
```

The use of `let` is simply a convenient abbreviation. The following would be equivalent:

```
prove_print (external_cnf_solver "picosat" ["%f"]) thm;
```

The first argument to `external_cnf_solver` is the name of the executable. It can be a fully-qualified name, or simply the bare executable name if it's in your PATH. The second argument is an array of command-line arguments to the solver. Any occurrence of `%f` is replaced with the name of the temporary file containing the CNF representation of the term you're proving.

The `external_cnf_solver` tactic is based on the same underlying infrastructure as the `abc` tactic, and is generally capable of proving the same variety of theorems.

To write a CNF file without immediately invoking a solver, use the `offline_cnf` tactic, or the `write_cnf` top-level command.

Compositional Proofs

The examples shown so far treat programs as monolithic entities. A Java method or C function, along with all of its callees, is translated into a single mathematical model. SAWScript also has support for more compositional proofs, as well as proofs about functions that use heap data structures.

Compositional Imperative Proofs

As a simple example of compositional reasoning on imperative programs, consider the following Java code.

```
class Add {
    public int add(int x, int y) {
        return x + y;
    }

    public int dbl(int x) {
        return add(x, x);
    }
}
```

Here, the `add` function computes the sum of its arguments. The `dbl` function then calls `add` to double its argument. While it would be easy to prove that `dbl` doubles its argument by following the call to `add`, it's also possible in SAWScript to prove something about `add` first, and then use the results of that proof in the proof of `dbl`, as in the following SAWScript code (`code/java_add.saw`).

```
let add_spec : JavaSetup () = do {
    x <- java_var "x" java_int;
    y <- java_var "y" java_int;
    java_return {{ x + y }};
    java_verify_tactic abc;
};

let dbl_spec : JavaSetup () = do {
    x <- java_var "x" java_int;
    java_return {{ x + x }};
    java_verify_tactic abc;
};

cls <- java_load_class "Add";
ms <- java_verify cls "add" [] add_spec;
ms' <- java_verify cls "dbl" [ms] dbl_spec;
print "Done.";
```

This can be run as follows:

```
# saw -j <path to rt.jar or classes.jar from JDK> java_add.saw
```

In this example, the definitions of `add_spec` and `dbl_spec` provide extra information about how to configure the symbolic simulator when analyzing Java code. In this case, the setup blocks provide explicit descriptions of the implicit configuration used by `java_extract` (used in the earlier Java FFS example and in the next section). The `java_var` commands instruct the simulator to create fresh symbolic inputs to correspond to the Java variables `x` and `y`. Then, the `java_return` commands indicate the expected return value of the each method, in terms of existing models (which can be written inline).

Finally, the `java_verify_tactic` command indicates what method to use to prove that the Java methods do indeed return the expected value. In this case, we use ABC.

To make use of these setup blocks, the `java_verify` command analyzes the method corresponding to the class and method name provided, using the setup block passed in as a parameter. It then returns an object that describes the proof it has just performed. This object can be passed into later instances of `java_verify` to indicate that calls to the analyzed method do not need to be followed, and the previous proof about that method can be used instead of re-analyzing it.

Interactive Interpreter

The examples so far have used SAWScript in batch mode on complete script files. It also has an interactive Read-Eval-Print Loop (REPL) which can be convenient for experimentation. To start the REPL, run SAWScript with no arguments:

```
# saw
```

The REPL can evaluate any command that would appear at the top level of a standalone script, or in the `main` function, as well as a few special commands that start with a colon:

```
:env      display the current sawscript environment
:type     check the type of an expression
:browse   display the current environment
:eval     evaluate an expression and print the result
:?        display a brief description about a built-in operator
:help     display a brief description about a built-in operator
:quit     exit the REPL
:load     load a module
:add      load an additional module
:cd       set the current working directory
```

As an example of the sort of interactive use that the REPL allows, consider the file `code/NQueens.cry`, which provides a Cryptol specification of the problem of placing a specific number of queens on a chess board in such a way that none of them threaten any of the others.

```
all : {n, a} (fin n) => (a -> Bit, [n]a) -> Bit
all (f, xs) = [ f x | x <- xs ] == ~zero

contains xs e = [ x == e | x <- xs ] != zero

distinct : {n,a} (fin n, Cmp a) => [n]a -> Bit
distinct xs =
    [ if n1 < n2 then x != y else True
    | (x,n1) <- numXs , (y,n2) <- numXs
    ] == ~zero
  where
  numXs = [ (x,n) | x <- xs | n <- [ (0:[width n]) ... ] ]
```

```
type Position n = [width (n - 1)]

type Board n = [n](Position n)

type Solution n = Board n -> Bit

checkDiag : {n} (fin n, n >= 1) => Board n -> (Position n, Position n) -> Bit
checkDiag qs (i, j) = (i >= j) || (diffR != diffC)
  where   qi = qs @ i
          qj = qs @ j
          diffR = if qi >= qj then qi-qj else qj-qi
          diffC = j - i                   // we know i < j

nQueens : {n} (fin n, n >= 1) => Solution n
nQueens qs = all (inRange qs, qs) && all (checkDiag qs, ijs `{n}) && distinct qs

ijs : {n}(fin n, n>= 1)=> [_](Position n, Position n)
ijs = [ (i, j) | i <- [0 .. (n-1)], j <- [0 .. (n-1)]]

inRange : {n} (fin n, n >= 1) => Board n -> Position n -> Bit
inRange qs x = x <= `(n - 1)

property nQueensProve x = (nQueens x) == False
```

This example gives us the opportunity to use the satisfiability checking capabilities of SAWScript on a problem other than equivalence verification.

First, we can load a model of the `nQueens` term from the Cryptol file.

```
# saw

   ___   __ _ _ _ _
  / __|/ _' | | | |
  \__ \ (_| | | | |
  |___/\__,_|\_,_/

sawscript> m <- cryptol_load "NQueens.cry"
Loading module Cryptol
Loading module Main
sawscript> let nq8 = {{ m::nQueens`{8} }}
```

Once we've extracted this model, we can try it on a specific configuration to see if it satisfies the property that none of the queens threaten any of the others.

```
sawscript> print {{ nq8 [0,1,2,3,4,5,6,7] }}
False
```

This particular configuration didn't work, but we can use the satisfiability checking tools to automatically find one that does.

```
sawscript> sat_print abc nq8
Sat [3,1,6,2,5,7,4,0]
```

And, finally, we can double-check that this is indeed a valid solution.

```
sawscript> print (nq8 [3,1,6,2,5,7,4,0])
True
```

More Sophisticated Imperative Models

The analysis of JVM and LLVM programs presented so far have been relatively simple and automated. The `java_extract` and `llvm_extract` commands can extract models from simple methods or functions with minimal effort. For more complex code, however, more flexibility is necessary.

The `java_symexec` and `llvm_symexec` commands provide greater control over the use of symbolic execution to generate models of JVM and LLVM programs. These two commands have similar structure, but slight differences due to the differences between the underlying languages.

The shared structure is intuitively the following: both commands take parameters that set up the initial symbolic state of the program, before execution begins, and parameters that indicate which portions of the program state should be returned as output when execution completes.

The initial state before symbolic execution typically includes unknown (symbolic) elements. To construct `Term` inputs that contain symbolic variables, you can start by using the `fresh_symbolic` command, which takes a name and a type as arguments, and returns a `Term`. A type can be written using Cryptol type syntax by enclosing it within `{| |}`. The name is used only for pretty-printing, and the type is used for later consistency checking. For example, consider the following command:

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

This creates a new `Term` stored in the SAWScript variable `x` that is a 32-bit symbolic word.

These symbolic variables are most commonly used by the more general Java and LLVM model extraction commands. The Java version of the command has the following signature:

```
java_symexec : JavaClass        // Java class object
            -> String           // Java method name
            -> [(String, Term)] // Initial state elements
            -> [String]         // Final (output) state elements
            -> Bool             // Check satisfiability of branches?
            -> TopLevel Term    // Resulting Term
```

This first two parameters are the same as for `java_extract`: the class object and the name of the method from that class to execute. The third parameter describes the initial state of execution. For each element of this list, SAWScript writes the value of the `Term` to the destination variable or field named by the `String`. Typically, the `Term` will either be directly the result of `fresh_symbolic` or an more complex expression containing such a result, though it is allowed to be a constant value. The syntax of destination follows Java syntax. For example, `o.f` describes field `f` of object `o`. The fourth parameter indicates which elements of the final state to return as output. The syntax of the strings in this list is the same as for the initial state description. The final parameter indicates whether to perform satisfiability checks on branch conditions. If this is `true`, SAW will use its internal version of ABC to check the satisfiability of each branch condition before executing the associated branch. If this is `false`, SAW will simply check whether the branch condition has a constant value.

An example of using `java_symexec` on a simple function (using just scalar arguments and return values) appears in the `code/java_symexec.saw` file, quoted below.

```
cadd <- java_load_class "Add";
add <- define "add" {{ \x y -> (x : [32]) + y }};
x <- fresh_symbolic "x" {| [32] |};
y <- fresh_symbolic "y" {| [32] |};
t <- java_symexec cadd "add" [("x", x), ("y", y)] ["return"] true;
print_term t;
t' <- abstract_symbolic t;
prove_print abc {{ \a b -> t' a b == add a b }};
print "Done.";
```

This script uses `fresh_symbolic` to construct two fresh variables, `x` and `y`, and then passes them in as the initial values of the method parameters of the same name. It then uses the special name `return` to refer to the return value of the method in the output list. Finally, it uses the `abstract_symbolic` command to convert a `Term` containing symbolic variables into a function that takes the values of those variables as parameters. This last step exists partly to illustrate the use of `abstract_symbolic`, and partly because the `prove_print` command currently cannot process terms that contain symbolic variables (though we plan to adapt it to be able to in the near future).

The LLVM version of the command has some additional complexities, due to the less structured nature of the LLVM memory model.

```
llvm_symexec : LLVMModule            // LLVM module object
            -> String                // Function name
            -> [(String, Int)]       // Initial allocations
            -> [(String, Term, Int)] // Initial state element
            -> [(String, Int)]       // Final state elements
            -> Bool                  // Enable branch SAT checking
            -> TopLevel Term         // Resulting Term
```

Symmetrically with the Java version, the first two arguments are the same as for `llvm_extract`. However, while the Java version of this command takes two additional arguments, the LLVM version takes three. The first list describes allocations, the second describes initial values, and the third describes results. For the first list, SAWScript will initialize the pointer named by the given string to point to the number of elements indicated by the `Int`. For the second list, SAWScript will write to the given location with the given number of elements read from the given term. The name given in the initial assignment list should be written as an r-value, so if `"p"` appears in the allocation list then `"*p"` should appear in the initial assignment list. The third list describes the results, using the same convention: read n elements from the named location. Finally, the `Bool` parameter indicates whether to perform satisfiability checking of branch conditions, instead of simply comparing them with the constant `False`.

The numbers given for a particular location in the three lists need not be the same. For instance, we might allocate 10 elements for pointer `p`, write 8 elements to `*p` at the beginning, and read 4 elements from `*p` at the end. However, both the initialization and result sizes must be less than or equal to the allocation size.

An example of using `llvm_symexec` on a function similar to the Java method just discussed appears in the `code/llvm_symexec.saw` file, quoted below.

```
m <- llvm_load_module "basic.bc";
add <- define "add" {{ \x y -> (x : [32]) + y }};
x <- fresh_symbolic "x" {| [32] |};
y <- fresh_symbolic "y" {| [32] |};
```

```
t <- llvm_symexec m "add" [] [("x", x, 1), ("y", y, 1)] [("return", 1)] false;
print_term t;
t' <- abstract_symbolic t;
prove_print abc {{ \a b -> t' a b == add a b }};
print "Done.";
```

This has largely the same structure as the Java example, except that the `llvm_symexec` command takes and extra argument, describing allocations, and the input and output descriptions take sizes as well as values, to compensate for the fact that LLVM does not track how much memory a given variable takes up. In simple scalar cases such as this one, the size argument will always be `1`. However, if an input or output parameter is an array, it will take on the corresponding size value. For instance, say an LLVM function takes as a parameter an array `a` containing 10 elements of type `uint32_t *`, which it reads and writes. We could then call `llvm_symexec` with an allocation argument of `[("a", 10)]`, and both input and output arguments of `[("*a", 10)]` (note the additional `*` in the latter).

Concretely, consider a function to calculate the dot product of two vectors. We can define this operation functionally in Cryptol as follows (and as in `code/dotprod.cry`).

```
zip : {n, a} (fin n, Arith a) => (a -> a -> a) -> [n]a -> [n]a -> [n]a
zip f xs ys = [ f x y | x <- xs | y <- ys ]

sum : {n, a} (fin n, fin a) => [n][a] -> [a]
sum xs = ys!0
  where ys = [0] # [ x + y | x <- xs | y <- ys ]

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

This code uses a very functional style, and declares several generic, polymorphic functions. A more specialized implementation of dot product in C might look more like the following, from `code/dotprod.c`.

```
#include <stdint.h>
#include <stdlib.h>

uint32_t dotprod(uint32_t *x, uint32_t *y, uint32_t size) {
    uint32_t res = 0;
    for(size_t i = 0; i < size; i++) {
        res += x[i] * y[i];
    }
    return res;
}
```

Here, we have two arrays of 32-bit integers, which we assume to both contain `size` elements. We can prove the equivalence between the C and Cryptol dot product functions with the following SAWScript program (in `code/dotprod.saw`).

```
import "dotprod.cry";
m <- llvm_load_module "dotprod.bc";
xs <- fresh_symbolic "xs" {| [12][32] |};
ys <- fresh_symbolic "ys" {| [12][32] |};
let allocs = [ ("x", 12), ("y", 12) ];
let inputs = [ ("*x", xs, 12)
```

```
              , ("*y", ys, 12)
              , ("size", {{ 12:[32] }}, 1)
              ];
let outputs = [("return", 1)];
t <- llvm_symexec m "dotprod" allocs inputs outputs true;
thm1 <- abstract_symbolic {{ t == dotprod xs ys }};
prove_print abc thm1;
```

The structure of this script is similar to the previous example, but has some additional complexities. First, we pass in an allocation list that declares that `x` and `y` each point to 12 elements of their respective types (both `uint32_t` in this case). Next, we state that the *values* pointed to by `x` and `y` are the (symbolic) values of `xs` and `ys` respectively, each of which consists of 12 elements. Finally, the `size` parameter is the constant `12`. Because the type of `t` is fixed after the `llvm_symexec` command has run, the Cryptol type checker can specialize the `dotprod` function to the appropriate type. ABC can then easily prove the equivalence between the C and Cryptol implementations.

Other Examples

The `code` directory contains a few additional examples not mentioned so far. These remaining examples don't cover significant new material, but help fill in some extra use cases that are similar, but not identical to those already covered.

Java Equivalence Checking

The previous examples showed comparison between two different LLVM implementations, and cross-language comparisons between Cryptol, Java, and LLVM. The script in `code/ffs_java.saw` compares two different Java implementations, instead.

```
print "Extracting reference term";
j <- java_load_class "FFS";
ffs_ref <- java_extract j "ffs_ref" java_pure;

print "Extracting implementation term";
ffs_imp <- java_extract j "ffs_imp" java_pure;

print "Proving equivalence";
let thm1 = {{ \x -> ffs_ref x == ffs_imp x }};
prove_print abc thm1;
print "Done.";
```

As with previous Java examples, this one needs to be run with the `-j` flag to tell the interpreter where to find the Java standard libraries.

```
# saw -j <path to rt.jar or classes.jar from JDK> ffs_java.saw
```

AIG Export and Import

Most of the previous examples have used the `abc` tactic to discharge theorems. This tactic works by translating the given term to And-Inverter Graph (AIG) format, transforming the graph in various ways, and then using a SAT solver to complete the proof.

Alternatively, the `write_aig` command can be used to write an AIG directly to a file, in AIGER format, for later processing by external tools, as shown in `code/ffs_gen_aig.saw`.

```
cls <- java_load_class "FFS";
bc <- llvm_load_module "ffs.bc";
java_ffs_ref <- java_extract cls "ffs_ref" java_pure;
java_ffs_imp <- java_extract cls "ffs_imp" java_pure;
c_ffs_ref <- llvm_extract bc "ffs_ref" llvm_pure;
c_ffs_imp <- llvm_extract bc "ffs_imp" llvm_pure;
write_aig "java_ffs_ref.aig" java_ffs_ref;
write_aig "java_ffs_imp.aig" java_ffs_imp;
write_aig "c_ffs_ref.aig" c_ffs_ref;
write_aig "c_ffs_imp.aig" c_ffs_imp;
print "Done.";
```

Conversely, the `read_aig` command can construct an internal term from an existing AIG file, as shown in `code/ffs_compare_aig.saw`.

```
java_ffs_ref <- read_aig "java_ffs_ref.aig";
java_ffs_imp <- read_aig "java_ffs_imp.aig";
c_ffs_ref <- read_aig "c_ffs_ref.aig";
c_ffs_imp <- read_aig "c_ffs_imp.aig";

let thm1 = {{ \x -> java_ffs_ref x == java_ffs_imp x }};
prove_print abc thm1;

let thm2 = {{ \x -> c_ffs_ref x == c_ffs_imp x }};
prove_print abc thm2;

print "Done.";
```

We can use external AIGs to verify the equivalence as follows, generating the AIGs with the first script and comparing them with the second:

```
# saw -j <path to rt.jar or classes.jar from JDK> ffs_gen_aig.saw
# saw ffs_compare_aig.saw
```

Files in AIGER format can be produced and processed by several external tools, including ABC, Cryptol version 1, and various hardware synthesis and verification systems.