LLVM/Java Verification with SAW

The SAW Development Team

Dec 15, 2025

Contents

1 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. The complete examples are available on GitHub (https://github.com/GaloisInc/saw-script/tree/master/doc/tutorial/code). Most of the examples make use of inline specifications written in Cryptol, a language originally designed for high-level descriptions of cryptographic algorithms. For readers unfamiliar with Cryptol, various documents describing its use are available here (http://cryptol.net/documentation.html).

2 Example: Find First Set

As a first example, we consider showing the equivalence of several quite 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 (zero being the least significant). 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.

2.1 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 `ffs.c` file on GitHub) 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.

2.2 Alternative Implementations

An alternative implementation, taken by the following function (also in `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;
}
```

Another optimized version, in the following rather mysterious program (also in `ffs.c`), based on the `ffs` implementation in musl libc (http://musl.libc.org/).

```
uint32_t ffs_musl (uint32_t x)
{
  static const char debruijn32[32] = {
    0, 1, 23, 2, 29, 24, 19, 3, 30, 27, 25, 11, 20, 8, 4, 13,
    31, 22, 28, 18, 26, 10, 7, 12, 21, 17, 9, 6, 16, 5, 15, 14
  };
  return x ? debruijn32[(x&-x)*0x076be629 >> 27]+1 : 0;
}
```

These optimized versions are much less obvious than the reference implementation. They might be faster, but how do we gain confidence that they calculate the same results as the reference implementation?

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

```
uint32_t ffs_bug(uint32_t word) {
    // injected bug:
    if(word == 0x101010) return 4; // instead of 5
    return ffs_ref(word);
}
```

SAWScript allows us to state these problems concisely, and to quickly and automatically 1) prove the equivalence of the reference and optimized implementations on all possible inputs, and 2) find an input exhibiting the bug in the third version.

2.3 Generating LLVM Code

SAW 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 -g -c -emit-llvm -o ffs.bc ffs.c
```

The `-g` flag instructs `clang` to include debugging information, which is useful in SAW to refer to variables and struct fields using the same names as in C. We have tested SAW successfully with versions of `clang` from 3.6 to 7.0. Please report it as a bug on GitHub (https://github.com/GaloisInc/saw-script/issues) if SAW fails to parse any LLVM bitcode file.

This command, and following command examples in this tutorial, can be run from the `code` directory on GitHub (https://github.com/GaloisInc/saw-script/tree/master/doc/tutorial/code). 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 running:

```
$ make ffs.bc
```

2.4 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 `ffs_llvm.saw`) is sufficient to automatically prove the equivalence of `ffs_ref` with `ffs_imp` and `ffs_musl`, and identify the bug in `ffs_bug`.

```
set_base 16;

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

print "Extracting implementation term: ffs_imp";
ffs_imp <- llvm_extract l "ffs_imp";

print "Extracting implementation term: ffs_musl";
ffs_musl <- llvm_extract l "ffs_musl";

print "Extracting buggy term: ffs_bug";
ffs_bug <- llvm_extract l "ffs_bug";
```

(continues on next page)

(continued from previous page)

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

print "Proving equivalence: ffs_ref == ffs_musl";
let thm2 = {{ \x -> ffs_ref x == ffs_musl x }};
result <- prove abc thm2;
print result;

print "Finding bug via sat search: ffs_ref != ffs_bug";
let thm3 = {{ \x -> ffs_ref x != ffs_bug x }};
result <- sat abc thm3;
print result;

print "Finding bug via failed proof: ffs_ref == ffs_bug";
let thm4 = {{ \x -> ffs_ref x == ffs_bug x }};
result <- prove abc thm4;
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 `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` command can verify the validity of such an assertion, or produce a counter-example that invalidates it. 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.

Similarly, the `sat` command works in the opposite direction to `prove`. It attempts to find a value for which the given assertion is true, and fails if there is no such value.

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

```
$ saw ffs_llvm.saw
```

(continues on next page)

(continued from previous page)

```
Loading file "ffs_llvm.saw"
Extracting reference term: ffs_ref
Extracting implementation term: ffs_imp
Extracting implementation term: ffs_musl
Extracting buggy term: ffs_bug
Proving equivalence: ffs_ref == ffs_imp
Valid
Proving equivalence: ffs_ref == ffs_musl
Valid
Finding bug via sat search: ffs_ref != ffs_bug
Sat: [x = 0x101010]
Finding bug via failed proof: ffs_ref == ffs_bug
prove: 1 unsolved subgoal(s)
Invalid: [x = 0x101010]
Done.
```

Note that 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>))`.

2.5 Cross-Language Proofs

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

The reference version (in `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; }
```

(continues on next page)

(continued from previous page)

```
        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
```

Like with `clang`, the `-g` flag instructs `javac` to include debugging information, which can be useful to preserve variable names.

Using `saw` with Java code requires a command-line option `-b` that locates Java. Run the code in this section with the command:

```
$ saw -b <path to directory where Java lives> ffs_compare.saw
```

Alternatively, if Java is located on your `PATH`, you can omit the `-b` option entirely.

Both Oracle JDK and OpenJDK versions 6 through 8 work well with SAW. SAW also includes experimental support for JDK 9 and later. Code that only involves primitive data types (such as `FFS.java` above) is known to work well under JDK 9+, but there are some as-of-yet unresolved issues in verifying code involving classes such as `String`. For more information on these issues, refer to this GitHub issue (https://github.com/GaloisInc/crucible/issues/641).

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

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

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

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

(continues on next page)

(continued from previous page)

```
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.";
```

Here, the `jvm_extract` function works like `llvm_extract`, but on a Java class and method name. The `prove_print` command works similarly to the `prove` followed by `print` combination used for the LLVM example above.

3 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 Z3 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:

```
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 (in `double.saw`) verifies that the two are equivalent using both internal Yices

and Z3 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";
double_ref <- llvm_extract l "double_ref";
let thm = {{ \x -> double_ref x == double_imp x }};

r <- prove yices thm;
print r;

r <- prove z3 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, their default behavior is to treat free variables as existentially quantified. By negating the input term, we can instead treat the free variables as universally quantified: a result of "unsatisfiable" from the solver indicates that the original term (before negation) is a valid theorem. 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, Bitwuzla, Boolector, CVC4, CVC5, MathSAT, Yices, and Z3.

4 External SAT Solvers

In addition to the `abc`, `z3`, and `yices` proof tactics used above, SAWScript can also invoke arbitrary external SAT solvers that read CNF files and produce results according to the SAT competition input and output conventions (https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html), using the `external_cnf_solver` tactic. For example, you can use PicoSAT (http://fmv.jku.at/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;x
```

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.

5 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.

5.1 Compositional Imperative Proofs

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

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

    public static 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 (`java_add.saw` on GitHub).

```
let add_spec = do {
    x <- jvm_fresh_var "x" java_int;
    y <- jvm_fresh_var "y" java_int;
    jvm_execute_func [jvm_term x, jvm_term y];
    jvm_return (jvm_term {{ x + y }});
};

let dbl_spec = do {
```

(continues on next page)

(continued from previous page)

```
    x <- jvm_fresh_var "x" java_int;
    jvm_execute_func [jvm_term x];
    jvm_return (jvm_term {{ x + x }});
};

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

This can be run as follows:

```
$ saw -b <path to directory where Java lives> 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 `jvm_extract` (used in the earlier Java FFS example and in the next section). The `jvm_fresh_var` commands instruct the simulator to create fresh symbolic inputs to correspond to the Java variables `x` and `y`. Then, the `jvm_return` commands indicate the expected return value of the each method, in terms of existing models (which can be written inline). Because Java methods can operate on references, as well, which do not exist in Cryptol, Cryptol expressions must be translated to JVM values with `jvm_term`.

To make use of these setup blocks, the `jvm_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 `jvm_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.

6 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
```

(continues on next page)

(continued from previous page)

```
    :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
```

(continues on next page)

(continued from previous page)

```
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.

```
sawscript> m <- cryptol_load "NQueens.cry"
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 [qs = [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
```

7 Other Examples

The `code` directory on GitHub (https://github.com/GaloisInc/saw-script/tree/master/doc/tutorial/code) 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.

7.1 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 `ffs_java.saw` compares two different Java implementations, instead.

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

print "Extracting implementation term";
ffs_imp <- jvm_extract j "ffs_imp";

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 `-b` flag to tell the interpreter where to find Java:

```
$ saw -b <path to directory where Java lives> ffs_java.saw
```

7.2 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 (http://fmv.jku.at/aiger/), 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 <- jvm_extract cls "ffs_ref";
java_ffs_imp <- jvm_extract cls "ffs_imp";
c_ffs_ref <- llvm_extract bc "ffs_ref";
c_ffs_imp <- llvm_extract bc "ffs_imp";
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 `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 -b <path to directory where Java lives> 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.