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.

© 2013 Galois, Inc.

Contents

© 2013 Galois, Inc.

Introduction

The SAWScript language 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 functional languages, with some special features specifically targeted at the coordination of verification 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 a simple function that identifies 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 exists in a number of standard C libraries, and can be implemented in several ways.

Reference Implementation

One simple implementation take the form of a loop in which the index starts out at zero, and we keep track of a mask initialized to have the least signficant 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 uses this approach.

```
#include <stdint.h>

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;
```

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, 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; }
```

© 2013 Galois, Inc.

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.

Generating LLVM Code

The SAWScript interpreter knows how to 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 code/ffs.bc code/ffs.c
```

Equivalence Proof

We now show how to use SAWScript to prove the equivalence of the reference and implementation versions of the FFS algorithm.

A SAWScript program is typically structured as a set of commands within a `main` function, potentially along with other functions defined to abstract over commonly-used combinations of commands.

The following script is sufficient to automatically prove the equivalence of the `ffs_ref` and `ffs_imp` functions.

```
main = do {
  print "Extracting reference term";
  (ffs_ref : [32] -> [32]) <- llvm_extract "ffs.bc" "ffs_ref" llvm_pure;

  print "Extracting implementation term";
  (ffs_imp : [32] -> [32]) <- llvm_extract "ffs.bc" "ffs_imp" llvm_pure;

  print "Proving equivalence";
  let thm1 x = ffs_ref x == ffs_imp x;
  prove abc thm1;

  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. The `prove` command can verify the validity of such an assertion. The `abc` parameter indicates what theorem prover to use.

© 2013 Galois, Inc.

Cross-Language Proofs

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

The reference version 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 code/FFS.java
```

Now we can do the proof both within and across languages:

```
main = do {
    (java_ffs_ref : [32] -> [32]) <- java_extract "FFS" "ffs_ref" java_pure;
    (java_ffs_imp : [32] -> [32]) <- java_extract "FFS" "ffs_imp" java_pure;
    (c_ffs_ref : [32] -> [32]) <- llvm_extract "ffs.bc" "ffs_ref" llvm_pure;
    (c_ffs_imp : [32] -> [32]) <- llvm_extract "ffs.bc" "ffs_imp" llvm_pure;

    let thm1 x = java_ffs_ref x == java_ffs_imp x;
    prove abc thm1;

    let thm2 x = c_ffs_ref x == c_ffs_imp x;
    prove abc thm2;

    let thm3 x = java_ffs_imp x ==  c_ffs_imp x;
    prove abc thm3;
```

© 2013 Galois, Inc.

```
    print "Done.";
};
```

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, other proof tools can be used, as well. The current version of SAWScript includes support for exporting models representing theorems as goals in the SMT-Lib language. These goals can then be solved using an external SMT solver such as Yices or CVC4.

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 verifies that the two are equivalent using both ABC, and by exporting an SMT-Lib theorem to be checked by an external solver.

```
main = do {
    (double_imp : [32] -> [32]) <- llvm_extract "double.bc" "double_imp" llvm_pure;
    (double_ref : [32] -> [32]) <- llvm_extract "double.bc" "double_ref" llvm_pure;
    let thm x = double_ref x == double_imp x;
    r <- prove abc thm;
    print r;
    let thm_neg x = not (thm x);
    write_smtlib1 "double.smt" thm_neg;

    print "Done.";
};
```

The new primitives introduced here are `not`, which constructs the logical negation of a term, and `write_smtlib1`, which writes a term as a proof obligation in SMT-Lib version 1 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.

At the moment, SMT-Lib version 1 is best supported, though version 2 is implemented for a smaller number of primitives. Writing SMT-Lib files tends to work well only with terms constructed from LLVM or SBV input files. The JVM simulator and AIG reading code construct terms including primitives that the SMT-Lib exporter doesn’t yet support.

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.

© 2013 Galois, Inc.

As a simple example of compositional reasoning, 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.

```
setup : JavaSetup ();
setup = do {
    x <- java_var "x" java_int;
    y <- java_var "y" java_int;
    java_return ((x + y) : [32]);
    java_verify_tactic abc;
};

setup' : JavaSetup ();
setup' = do {
    x <- java_var "x" java_int;
    java_return (x + x : [32]);
    java_verify_tactic abc;
};

main : TopLevel ();
main = do {
    ms <- java_verify "Add" "add" [] setup;
    ms' <- java_verify "Add" "dbl" [ms] setup';
    print "Done.";
};
```

In this example, the definitions of `setup` and `setup'` 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`. 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.

Future Enhancements

Improved Integration with External Proof Tools

- More complete SMT-Lib export support.
- Support for automatic invocation of SMT solvers other than Yices, and interpretation of their output.
- Support for generation of (Q)DIMACS CNF and QBF files, for use with SAT and QBF solvers.
- Support for automatic invocation of SAT and QBF solvers, and interpretation of their output.

Improved Support for Manipulating Formal Models

- Applying formal models automatically to a large collection of randomly-generated concrete arguments.

© 2013 Galois, Inc.