SawScript

The Galois HA Java Team
hajava@galois.com

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 Java Verifier and Cryptol based formal verification technologies. We use various simple Java programs as examples, inspired the domain of elliptic-curve cryptography. Our proofs either directly take place within the SawScript specification language, or use SawScript to equivalence check Java functions against their Cryptol counterparts.

Contents

1. Introduction

SawScript is a proof scripting language and its associated proof engine, aimed at simplifying the use of Galois's Java Verifier and Cryptol [1] based verification technologies. SawScript allows concise descriptions of proof steps, freeing the users from the details of how the proofs are actually constructed and run.

NB While this initial version of SawScript closely reflects our experiences with proofs about Java and Cryptol programs, we consider it a general tool for describing proofs for programs written in arbitrary languages. Therefore, the surface syntax of SawScript is likely to evolve as we add capabilities on both the proof construction and specification aspects, and as we focus on supporting other languages, such as LLVM or C.

2. Basics

In this section we will consider a couple of very simple Java programs and see how we can specify and prove them correct using SawScript.

2.1. Checking that an array is properly reset

Consider the following Java method:

```
static private boolean is_zero(int[] x) {
   for (int i = 0; i != x.length; ++i) {
     if (x[i] != 0) return false;
   }
   return true;
 }
```

The function `is_zero` takes a reference to an arbitrary sized integer array, and returns `true` if all the elements are set to `0`, i.e., if the array is properly reset. In our first example, we will see how we can write a specification for this function, and prove that it is correct using SawScript.

NB We should emphasize that all Java proofs in SawScript are done over the corresponding JVM bytecode. While Java programmers typically do not think in terms of JVM during day-to-day programming, there are subtle details that do come up when we try to prove properties of Java programs in this way. We will try to point out these differences explicitly as necessary.

Given the `is_zero` Java method above, here is how we would express the proof that it does indeed check that its argument is properly set to all `0`s in SawScript:

```
1  method com.galois.ecc.P384ECC64.is_zero {
2    var args[0] :: int[12];
3    return join(valueOf(args[0])) == 0:[384];
4    verify abc;
5  };
```

Let us walk through this proof script in detail to explain the basic concepts behind SawScript:

Line 1 This is where the property specification starts for the `is_zero` method. We tag it with its class name, which happens to be `com.galois.ecc.P384ECC64` in this case. Note that SawScript will follow the Java inheritance hierarchy to find the method `is_zero` in this class: It can either be directly defined in that class, or it can be defined in any one of its superclasses: SawScript will properly locate it, and will complain if it cannot.

Tip To be able to locate java classes, you will need to tell SawScript the class and `jar` paths as installed on your system. See the `-c` and `-j` options of SawScript for details. (You can issue `sawScript --help` to see all available options.)

Line 2 The first thing we do in a method specification is to provide the type information for the arguments to the method. The `var` declaration states that the first argument to the method, i.e., `arg[0]`, is an array of length 12, containing Java `int`s. We write this type in SawScript like this: `int[12]`.

NB The attentive reader would have no doubt wondered how we came up with the number 12 for the length of our array. After all, the Java program we have given works for arrays of arbitrary sizes, not just 12. (Using Cryptol terminology, we would say that the Java method `is_zero` is size polymorphic.) Ideally, we would like to be able to say that the property holds for arbitrary arrays. Unfortunately, SawScript currently only allows for monomorphic specifications: We cannot write polymorphic properties. This is not a mere limitation of the current implementation of SawScript: As we have demonstrated before in the context of Cryptol, one cannot expect to be able to automatically prove arbitrary polymorphic properties of bit-precise programs [5].

In SawScript, we aim for push-button automated verification, and hence we focus our attention on monomorphic properties only. Therefore, we have to give a fixed constant size for the array, and in this case we choose that number to be 12. (While it's unlikely that this monomorphism restriction will ever be completely removed from SawScript, future versions might relax the requirements, and allow for some parametric proofs to be encoded directly.)

Line 3 At this point, we are ready to describe the desired behavior of the `is_zero` method directly in SawScript. This is the goal of the `return` statement. The expression on the right hand-side is a direct encoding of how we would expect this method to work: If the argument has all 0 elements, then it should return `true`, otherwise it should return `false`. It should be noted that the JVM internally represents `bool` values as 32-bit integers. SAWScript implicitly converts `true` to the 32-bit value 1, and `false` to the 32-bit value 0.

To understand how the specification is given, let us consider the test-expression in more detail:

```
join(valueOf(args[0])) == 0:[384]
```

Here are the constituents:

- `args[0]`: This is the first argument to the `is_zero` method, that we declared to have type `int[12]` on line 1. Most importantly, this is a Java value, and not a SawScript value.
- `valueOf`: This function lifts values from the Java world into SawScript. Whenever we refer to a Java value (such as `args[0]`), we have to explicitly bring it to the SawScript level using `valueOf`. This two-level type-system is crucial in keeping the two worlds separate: The object level at which Java operates, and the proof level at which SawScript works. The `valueOf` call basically tells SawScript to take the Java value and bring it to the level where we write the specifications.
- `join`: The function `join` is a primitive in SawScript, similar to Cryptol's `join`. It takes a sequence of words and joins them together to make one big word out of them. In this case, its argument is a 12 element array containing 32-bit values, and hence it returns a $12 \times 32 = 384$ bit word.
- `0:[384]`: This is the SawScript constant 0 at 384 bits. SAWScript follows Cryptols notation, word types are written `[n]`, where `n` is the bit-size. You will also notice that SawScript requires annotations on all constants, i.e., the type specification `[32]` is mandatory on the constant 0.

In summary, the `return` statement on lines 3 is telling SawScript that when the method `is_zero` completes execution, it will return `true` if all the elements of the input array are equal to 0, and `false` otherwise. This is how we capture the operation of the Java method as a functional SawScript specification.

Line 4 Finally, the user is giving a hint to SawScript to use the the `abc` tool to verify this specification, essentially showing correctness using first bit-blasting and then discharging the associated verification conditions using Mishchenko's ABC tool [3]. Other tactics include `rewrite`, `yices`, and `smtlib`. The `rewrite` tactic tries the internal rewriting engine to show correctness. Users can also specify a sequence of tactics such as

```
        verify { rewrite; abc; }
```

which means to first apply the rewriter, and send the result to abc. If the `verify` command is ommitted, SAWScript will perform no verification at all, and print a message stating that the specification is assumed correct.

NB In this particular example, the proof is specified directly within SawScript's specification language: We do not perform equivalence checking, but rather state and prove the property directly within SawScript. Later on, we will also see how to use SawScript to equivalence check Java programs against Cryptol reference implementations.

If you were to try this example using SawScript, you will see that the proof completes instantaneously with success. If the specification was in error, then SawScript will print a counter example if the `abc` tactic is used, or the final verification condition that it failed to discharge. (See Section 2.3 for details.)

2.2. Resetting an array

The `is_zero` method of the previous section introduced the basic concepts behind SawScript. One important thing to note about `is_zero` is that it is a function without any side effects: It does not modify its arguments. The goal of this section is to illustrate how we can prove properties of Java programs that do modify their arguments. We will do so by studying the corresponding Java procedure that sets the elements of its argument array to all `0`s:

```
static void set_zero(int[] x) {
  for (int i = 0; i != x.length; ++i)
     x[i] = 0;
}
```

Unlike `is_zero`, `set_zero` does not return any values, but rather simply wipes out the elements of the given array. We would like to state that it indeed zeros out the given array as a property, and prove it correct using SawScript. Here is the corresponding functional specification:

```
1  method com.galois.ecc.P384ECC64.set_zero {
2    var args[0] :: int[12];
3    ensure valueOf(args[0]) := split(0:[384]) : [12][32];
4    verify abc;
5  };
```

What is new in this specification is the use of the `ensure` clause (line 3). What `ensure` allows us is to state precisely what observable modifications the Java method will have performed after it *completes* its execution. In this case, we are stating that `args[0]` must essentially be equivalent to the 384-bit value `0` split into an array of length 12, containing all 0 elements.

NB Similar to the specification for `is_zero`, the above property is stated for arrays of length 12, even though the corresponding Java method can accept arrays of arbitrary size. Also, you will find that SawScript requires type annotations for uses of the `split` function, similar to the mandatory annotations on constants.

A given specification may contain both a `return` clause and multiple `ensure` clauses, as necessary. If you were to try the above example using `sawScript`, you will again see that the proof completes instantaneously.

2.3. Basic execution of a SawScript proof

Before moving on to bigger examples, and in particular to equivalence checking between Java and Cryptol, let us take a moment to summarize how a basic SawScript proof proceeds. When SawScript first encounters the specification, it first typechecks it to ensure that the Java, Cryptol, and SawScript types are compatible and used in a consistent way. This initial typechecking can catch many early specification bugs, resulting in huge time savings from a development/evaluation perspective.

When verifying a method, SawScript will perform the following actions on each method specification:

1. Search the current class search path to locate the correct Java class.

2. Find the corresponding method body to be verified, by looking in the Java class thus located, and its superclasses.

3. Create an initial simulator state, with symbolic input values derived using the type information from the `var` declarations, and constants from the `const` declarations.

4. Run the method symbolically on these inputs, using the symbolic simulation capabilities supported by the Java Symbolic Simulator (`jss`),

5. Compare the final JVM state returned by the symbolic simulator against the values specified in the method specification `return` and `ensure` statements. The main steps in this comparison are to:

   - Identify any new memory that was allocated for classes not in the current specification, or modified fields that were not specifically mentioned in the specification. If either new memory is allocated, or unexpected modifications are found, fail immediately with an appropriate message.
   - Iterate through the defined field values, arrays, and the `return` statement. For each one, generate a *equivalence condition* $v_i = s_i$ asserting that the value v_i from the symbolic simulator equals the relevant specification value s_i obtained from the *ensure* clause. If no *ensure* clause is defined for the value, then the specification value s_i is the value in the initial state, i.e., it must not have been modified by the method.

6. Combine the assumptions $a_1, \ldots, a_n$ defined in the specification with the equivalence conditions from the previous step to generate a single verification condition:

$$a_1 \wedge \cdots \wedge a_n \implies v_1 = s_1 \wedge \cdots \wedge v_{n'} = s_{n'}.$$

7. Use the user specified verification tactic to attempt to reduce the verification condition to `True` in all cases. If the tactic fails to discharge the verification goal, we report the failure to the user along with any relevant information to help understand the failure.

8. Once a specification has been processed, the symbolic simulator will use the specification instead of the Java bytecode in symbolically simulating later methods. This capability not only accelerates symbolic simulator performance, but is essential for the compositional verification of methods that produce proof terms too large for automated technologies (SAT or otherwise) to handle.

SAWScript currently supports two main verification tactics: (1) using rewriting to simplify or solve a verification condition; (2) applying bitblasting to generate an And-

Inverter Graph (AIG) from the verification problem, and sending it to the ABC equivalence checker for verification. These verification tactics can either be used by themselves, or sequentially composed by listing the tactics in a block.

The two verification tactics are useful in different situations. Using the *abc* tactic is much more automatic, but *abc* may fail to terminate on a verification problem in a reasonable amount of time, without providing any insight into why. Using the *rewrite* tactic is a more labor intensive affair, and requires the SawScript author to invest effort in coming up with a suitable set of rewrite rules to solve the verification problem. However, when the right rules are used, the rewriting based proofs can go much faster, as we shall see later in Section 4.

The verification tactic also has an impact on what information is returned to the user when verification fails:

- If the `abc` is used and `abc` is able to generate a counterexample, then SawScript will display the counterexample to the user with a detailed description of what Java input values triggered it, and how the expected and actual states differed.
- If `abc` is not used as the proof method, then SawScript will display the final formula to the user as the remaining VC. The user can then inspect this formula manually and decide how to continue the proof, potentially by adding further rewrite rules.

2.4. Language mechanics

Some important points to keep in mind when writing SawScript proofs:

- The input language is case-sensitive.
- Indentation is immaterial, and is ignored.
- Statements are always terminated with a semicolon.
- Comments are C++ style: Line comments start with `//` and continue till the end of line. Block comments start with `/*` and end with `*/`. Block comments can be nested.
- Aside from the usual decimal notation, constants can also be written in hexadecimal (start with `0x`), octal (start with `0o`), and binary (start with `0b`) notations. In addition, they can also be written using the polynomial syntax, which can be very

handy in writing very large numbers. For instance, we can write the prime for the P384 curve like this [2, Section 2.8.1]:

```
let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];
```

- The following words are reserved, and cannot be used as identifiers:

```
abc      args     assert     Bit           blif     boolean
byte     char     disable    double        else     enable
ensure   expand   extern     false         float    forAll
from     if       import     int           let      line
locals   long     mayAlias   method        modify   off
on       pc       pragma     quickcheck    return   rewrite
rule     SBV      set        short         smtlib   then
this     true     var        verification  verify   yices
```

- SawScript supports literate programming. In this format, the code is enclosed in `\begin{code}` and `\end{code}` markers, which should appear at the beginning of lines that enclose the proof segments. There can be any number of segments in a given literate file. Segments residing outside these blocks are treated as comments. Birdtrack style literate scripts (i.e., code lines starting with the character >) are supported as well.

- While the preferred extension for SawScript files is `.saw`, you can pick any extension you like for regular SawScript files. However, literate SawScript files **must** have either the extension `.lsaw` or `.tex`.

- You can break your proof into multiple files, and import them in others, using the statement:

```
import "subProofs/mySuperLemma.saw";
```

  Import statements can appear anywhere in the file. However, if you do refer to a symbol defined in an imported file, then the corresponding `import` must precede the reference to the symbol. Cyclic imports are prohibited and will be rejected.

3. Equivalence checking Java against Cryptol

In our proofs so far, we simply used SawScript to state and prove properties about simple Java programs. In traditional equivalence checking style verification, we would like to prove that a given *implementation* of an algorithm in one language behaves exactly the same as a *reference* implementation in another (not necessarily the same) language. Thus, we need a means to be able to relate different source languages within SawScript. In our case, for instance, implementations will be in Java, while reference specifications will be in Cryptol, although other languages such as C or LLVM might also be added to the set of supported languages in the future as well.

The aim of this section is to illustrate how to verify Java implementations against Cryptol reference specifications in SawScript. Note that the treatment of these languages is symmetric: One can also view these proofs as verifying Cryptol implementations correct against reference Java specifications, depending on which implementation the user "trusts" in a given verification task.

3.1. Interacting via SBV files

In SawScript, the interaction between Java and Cryptol is handled through Cryptol's SBV backend [1, 4]. For a given Cryptol function (subject to certain restrictions), Cryptol's SBV backend is capable of compiling it into a word-level formal model, and writing it to the disk as a binary file for further processing. These files are also known as SBV files, and use the extension `.sbv`.[1] The overall architecture of how Cryptol and Java programs are combined is depicted in Figure 1.

Once the SBV files are obtained from Cryptol, we simply import them in SawScript, using the `extern SBV` command. As an example, we will use an implementation of the field addition operation over the prime field underlying the P384 curve, coded in a Cryptol function with the following signature:

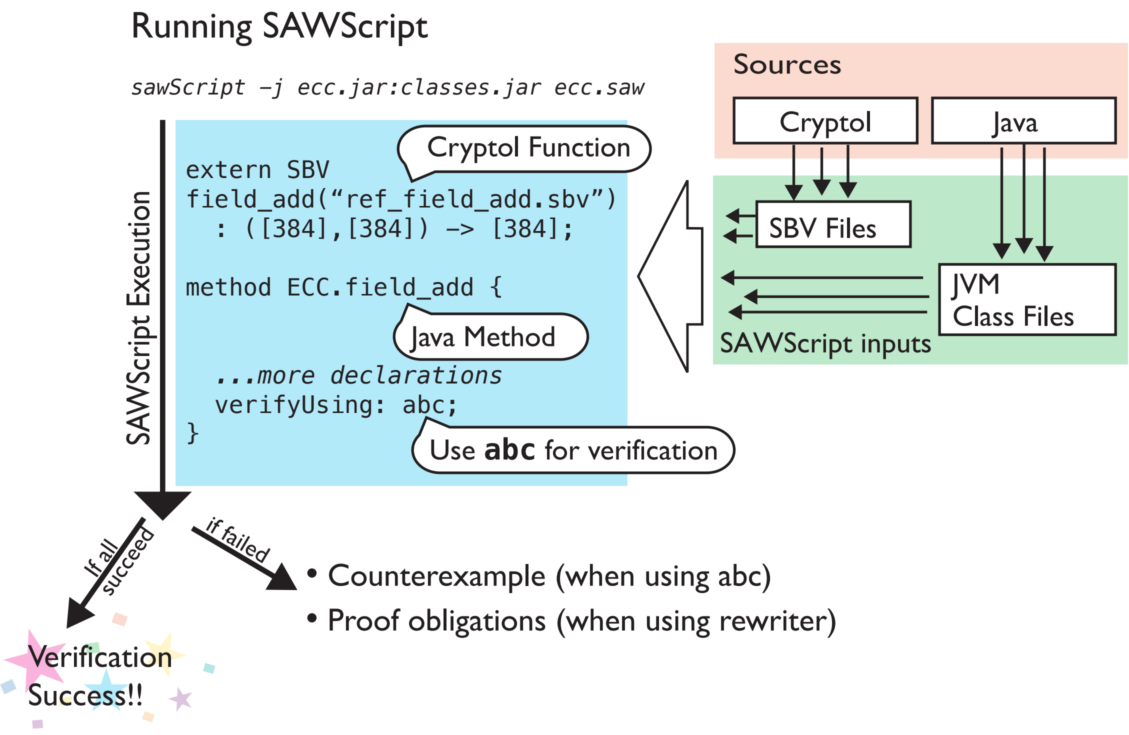
```
  ref_p384_add : ([384],[384]) -> [384];
```

We will not need to know how `ref_p384_add` itself is coded in Cryptol, as SawScript will simply be using the SBV file corresponding to it. We will assume that the SBV file is already generated and stored in a file named `"sbv/ref_p384_add.sbv"`. We simply import this file into SawScript as follows:

```
 extern SBV cry_fadd("sbv/ref_p384_add.sbv") : ([384],[384]) -> [384];
```

The above declaration tells SawScript to load the SBV file and admit it as a function

[1] We will not go into the details of how to generate SBV files from Cryptol in this document, as it is detailed elsewhere [4]. Unless there are termination related issues, Cryptol's `:set sbv_strictWords=True` option should be used to generate the SBV files for verification purposes. (Note that the default for this setting is `False`, hence it should be explicitly set to `True` by the user within Cryptol.)



```
sawScript -j ecc.jar:classes.jar ecc.saw
```

```
extern SBV
field_add("ref_field_add.sbv")
  : ([384],[384]) -> [384];

method ECC.field_add {

  ...more declarations
  verifyUsing: abc;
}
```

- Counterexample (when using abc)
- Proof obligations (when using rewriter)

Figure 1: SAWSCRIPT verification workflow

with the given type. (The type annotation is mandatory. SAWSCRIPT will make sure that the declared type does indeed match what the SBV file defines.)

Once this declaration is given, we can refer to `cry_fadd` in our SAWSCRIPT proof as if it was one of the primitive functions supported by SAWSCRIPT itself.

3.2. Field addition

Let us now turn to the Java implementation of `field_add`, which we will assume to have been defined in the class `com.galois.ecc.P384ECC64`, implementing field addition for the prime field underlying the P384 curve [2, Section 2.8.1]:

```
public void field_add(int[] z, int[] x, int[] y) {
    if (add(z, x, y) != 0 || leq(field_prime, z)) decFieldPrime(z);
}
```

Note that we are *not* showing the definitions of `add`, `leq` or `decFieldPrime` functions here for brevity, as our specification will not need their details. The crucial thing to note is that `field_add` is not a pure function: It will add the field elements represented by `x` and `y`, and store the result in `z`, destructively updating its first argument. Hence, our SawScript specification will have to account for this state change.

The constant `field_prime` refers to the prime of the field underlying P384. This constant is represented in the Java version using an int array with 12 elements. We will represent the field prime in SawScript, as given in the SEC documentation [2, Section 2.8.1]:

```
let field_prime = <| 2^384 - 2^128 - 2^96 + 2^32 - 1 |> : [384];
```

Note the use of the polynomial notation as an aid in getting the specification looking as close as possible to the published documentation.

Here is the SawScript method specification for `field_add`:

```
1   method com.galois.ecc.P384ECC64.field_add {
2     var args[0], args[1], args[2] :: int[12];
3     mayAlias { args[0], args[1], args[2] };
4
5     var this.field_prime :: int[12];
6     assert valueOf(this.field_prime) := split(field_prime) : [12][32];
7
8     ensure valueOf(args[0]) :=
9         split(cry_fadd( join(valueOf(args[1]))
10                      , join(valueOf(args[2])))) : [12][32];
11
12    verify abc;
13  };
```

Let us go over this spec line by line, explaining how we constructed it:

Lines 1,2 As before, we indicate for which Java method the specification is written for, along with the types of the arguments to the method. Again, the Java method takes arbitrary length integer arrays as arguments, but the use case is for 384-bit numbers, which require an array of 12 32-bit Java `int`s. Thus we fix the types at `int[12]`.

Line 3 Whenever we have a Java method that takes multiple arguments as references, we have to be concerned about aliasing. That is, whether the method arguments `x`, `y`, and z might actually refer to the same array. This is very important since field_add modifies z, and it might have some assumptions regarding whether z might alias x, y, or both. Unfortunately, Java does not let programmers to indicate aliasing explicitly, but our proof has to account for all possibilities. By default, SawScript assumes that reference arguments do *not* alias each other. If this is not the case, then the user needs to tell SawScript explicitly which arguments might alias each other. We do this with a mayAlias declaration, as illustrated in line 4. As the name mayAlias suggests, aliasing is *not* required: It just tells SawScript to make sure the proof holds even if these arguments can alias each other. (Naturally, adding aliasing constraints will make proofs more complicated to finish, but the specification complexity will be hidden from the user.) In this case, we are telling SawScript that any two, or all three arguments to field_add might alias each other, i.e., refer to the same array.

Lines 5,6 Remember that the Java method referred to the array field_prime, which happens to be a protected final field in com.galois.ecc.P384ECC64. In line 6, we are telling SawScript that the constant field_prime we have defined above using the polynomial notation, and the Java value this.field_prime are indeed equivalent. (Note that we have to use split to turn our 384-bit field prime into a Java array, as Java represents field_prime internally as an array.)

Lines 8-10 We now express the correctness condition for the field_add method. Since there is no return value for this procedure, we cannot use a return statement. Instead, we express that the value of the first argument (args[0]) will have changed in a certain way when the method completes execution. We do this with the ensure statement on lines 8-10. The specification simply says that if you take the values of args[1] and args[2], and then call the cry_fadd function we have imported from Cryptol, then the result is what the value args[0] will get assigned at the end of the method execution. That is, the specification is simply:

```
// type incorrect, but illustrates the idea!
ensure args[0] := cry_fadd(args[1], args[2]);
```

We simply have to insert calls to valueOf to get values from Java into SawScript, and split/join appropriately to make sure the treatment of arrays and 384-bit words do match their intended usage.

Note that the values of args[0] and args[1] on the right hand side of the ensure statement will refer to the initial values of the arguments when the method starts executing, i.e., any changes to them due to aliasing will be properly accounted for in the proof.

Line 12 Finally, we give a hint to SawScript to use bit-blasting and in particular the ABC tool to complete the proof.

Note To give a sense of the performance of SawScript, the above proof completes in about 45 seconds on a decent commodity laptop.

3.3. Field subtraction

Similar to field addition, the correctness proof of field subtraction proceeds in a similar fashion:

```
extern SBV cry_fsub("sbv/ref_p384_sub.sbv") : ([384],[384]) -> [384];

method com.galois.ecc.P384ECC64.field_sub {
  var args[0], args[1], args[2] :: int[12];
  mayAlias { args[0], args[1], args[2] };
  var this.field_prime :: int[12];
  assert valueOf(this.field_prime) := split(field_prime) : [12][32];
  ensure valueOf(args[0]) :=
       split(cry_fsub( join(valueOf(args[1]))
                     , join(valueOf(args[2])))) : [12][32];
  verify abc;
};
```

(This particular method specification will come in handy when we attempt to prove the double decrement operation correct in Section 4.1.)

Again, the proof for `field_sub` takes around 2 minutes to complete on a commodity laptop.

3.4. Using assumptions

The `assert` directive tells SawScript that a particular boolean expression will be assumed to hold when a proof is being attempted. This directive is useful for stating invariants on the data. For instance, we can state:

```
assert is_field(join(valueOf(args[0])));
```

to state that the Java value of `args[0]` satisfies the predicate `is_field`, which we might have imported from an SBV file earlier:

```
extern SBV is_field("sbv/ref_p384_is_val.sbv") : [384] -> Bit;
```

When SawScript sees this assert directive, it will use it as a hypothesis during the proof, indicating that the method expects its inputs to satisfy certain conditions. Note that the proof may not go through without this assert statement, if the method does not guarantee to work correctly when its inputs do not have the property indicated.

On the flip side of the coin, any other method that uses a method with assert directives will have to make sure that the required conditions do indeed hold when the call is made. SawScript will automatically track such dependencies and discharge them as appropriate, or the proof will fail with a bug identified.

3.5. Indicating "don't care" conditions

When SawScript attempts a proof, it will make sure that *all* the changes made by the method are captured in ensure statements. SawScript will not allow a proof to complete unless all of its side effects are properly accounted for.

It is, however, quite possible that a method might be making some internal state changes that we really do not care about, as far as the verification tasks are concerned. For instance, it is unlikely that we want to explicitly track changes to private class fields that are used for temporary storage purposes. In such cases, we can use the modify directive to tell SawScript to ignore any changes made to such references. For instance, the directive:

```
modify valueOf(this.tmpBuffer);
```

tells SawScript that the method changes the value of the array tmpBuffer during its execution, but we do not care about such changes. If we omit this directive, SawScript will make sure that the field tmpBuffer does *not* change during the execution of the method, indicating a much stronger requirement.

4. Using rules

The only verification technology we have so far employed in our SawScript proofs is calling out to ABC, which essentially bit-blasts the problem and applies SAT based reasoning. While this technique is suitable for a variety of verification tasks, it is not powerful enough to handle many problems that crop up in cryptography due to large word sizes needed. For instance, a typical ECC algorithm will operate on 384-bit words (or larger), and consequently will suffer from the classic state-explosion problem when bit-blasted. In particular, problems involving multiplication are known to take time exponential to the word size of the variables involved, at least when using the current SAT based technologies [6, Section 6.3.1].

As an alternative proof technology, SawScript comes with a rule-based rewrite engine to perform traditional equational rewriting style proofs. This mode is selected using the directive:

```
verify rewrite;
```

Alternatively, the user can also tell SawScript to first try rewriting, and then send the remaining unsolved goals to ABC, using the directive:

```
verify { rewrite; abc };
```

Note that the `verify` directive is per method specification: Different specifications in the same SawScript file can use different tactics.

In the remainder of this section we will illustrate how to use the rewriting engine to perform equivalence proofs of Java programs against their Cryptol counterparts.

4.1. Field double-decrement

As a simple example of how rewriting can significantly outperform a bit-blasting based proof, consider the operation of double decrementing a field point that shows up often in ECC algorithms. Given two field points z and x, the double-decrement operation computes the value of $z - 2 \times x$:

```
private void field_dbl_dec(int[] z, int[] x) {
  field_sub(z, z, x);
  field_sub(z, z, x);
}
```

As usual, field-elements are represented as integer arrays, and we modify the first argument (z) directly to subtract the x argument twice. Note that the implementation is given as a procedure that modifies its first argument, as opposed to a pure function that returns a new result. We can write the method specification for `field_dbl_dec` as follows (note the use of the local `let` bindings on lines 5 and 6 to improve readability):

```
1   method com.galois.ecc.P384ECC64.field_dbl_dec {
2     var args[0], args[1] : int[12];
3      const valueOf(this.field_prime) := split(field_prime) : [12][32];
4
5      let jarg0 = join(valueOf(args[0]));
6      let jarg1 = join(valueOf(args[1]));
7      ensure args[0] :=
8          split(cry_fsub(cry_fsub(jarg0, jarg1), jarg1)) : [12][32];
9
10     verify abc;
11   };
```

If you let SawScript run this proof using bit-blasting you will see that it indeed does finish the proof, in about 2 minutes of run time.

4.2. Adding rules

Let us try the double decrement proof again, this time using the rewriter. To do so, replace the verify command on line 10 as follows:

```
verify rewrite;
```

If you try now with SawScript, you will get an immediate verification failure, indicating that the rewriter was not able to discharge the verification condition. The reason for the failure is that the rewriter comes with no pre-installed rules on its own: SawScript's rewriter features a rewrite engine that makes no assumptions about the underlying domain, providing maximum flexibility to the user for scripting the proof. This is the essence of the modular proof architecture featured by SawScript, where users can guide the prover to more and more efficient proofs by providing appropriate lemmas via rewrite rules.

Here is the error message from SawScript:

```
The rewriter failed to reduce the verification condition generated
from "args[0]" in the Java method
com.galois.ecc.P384ECC64.field_dbl_dec to 'true'.
The remaining goal is:
  (implies true
   (==
    (split
     (cry_fsub (join (split (cry_fsub{2} (join ?1) (join{1} ?0)))) n1))
```

```
      (split (cry_fsub n2 n1))))

  Please add new rewrite rules or modify existing ones to reduce the
  goal to true.
```

It will take a while for users to get familiar with SAWSCRIPT's internal term language, but the following is the essence of this message: When the rewriter tried to show the equivalence, it was not able to *combine* the usages of calls to split and join. In particular, we need to tell SAWSCRIPT that splitting a large word into an array and joining it right back will produce precisely the same word. Thus, we add the following rewrite rule:[2]

```
  rule join_split: forAll {x:[384]}. join(split(x) : [12][32]) -> x;
```

What the above rule is stating that if we take a 384-bit value x, split it into an array of 12 elements each of which is 32 bits wide, and join it back together, then we would get x back. (See Section 4.4 for more information on how rules are written.)

If you now try this proof, you will see that it fails again, with a slightly simpler unresolved term. Luckily, the rule we need in this case is trivial:

```
  rule eq_elim: forAll {x:a}. x == x -> true;
```

After the addition of this rule (again before the method specification), the proof will complete in less than a second! Compare this to the run-time of the pure bit-blasting based proof which took about 2 minutes. The performance gains can be even more substantial for larger proofs. In fact, a rewriting based proof might be the only means to complete a proof in practice, which would otherwise not be completed using reasonable amounts of time and/or memory with existing SAT based technologies.

4.3. How to determine the rules

Perhaps the most difficult part of a rewrite based proof is to determine what rules to add to the system. As we have indicated before, SAWSCRIPT's rewriting engine comes with no hard-coded rules, to make sure we have a system that is as flexible as possible. However, it can be frustrating to determine what rules a proof might need in order to discharge all verification conditions, especially for new users. Aside from the need for getting used to the internal term language of SAWSCRIPT to understand the unresolved goals,

[2] Make sure to add the rule *before* the method specification, as otherwise it will not be taken into account during the proof.

one needs to develop some intuitive understanding of how a rewriting based proof proceeds. We consider the tasks of determining and working with the rules amongst the most important future development tasks for improving SawScript's usability. In particular, we anticipate the need for being able to group rules into rule sets that can be selectively applied at different times, along with facilities to automatically extract rules from unresolved goals, all aimed at simplifying the proof construction process.

In the mean time, Galois intends to provide a set of rewrite rules that we have found to be useful in working with Java programs, especially those arising in the domain of elliptic-curve cryptography. Galois will be making these rules available as a separate saw file that users can simply import into their own proof scripts and tweak as needed. We believe that we can develop a robust set of rewrite rules that should simplify the proof construction tasks for the domain of cryptography considerably.

4.4. Anatomy of a rule

To illustrate how rules are syntactically constructed, consider the following example stating that append is associative:

```
rule appendAssoc:
      forAll {x:[a], y:[b], z:[c]}.  (x # y) # z -> x # (y # z);
```

Each rule is given a name, appendAssoc above, which is used for diagnostic purposes. Following this, we list the free variables mentioned in the rule body. For the appendAssoc example, we have three variables, named x, y, and z. Note that each parameter should be given an explicit type, although the type itself can be polymorphic. For instance, the above rule states that x is a word that has a bits, y is a word of size b bits, etc. Following this comes the method body, which consists of two expressions separated by ->. The idea is that the left-hand-size expression will be rewritten to the right-hand-side expression. Note the directionality of the rule: While the underlying rule is actually an equation, we syntactically write the rule in the form lhs -> rhs, since the rewriter will never try to reduce the right-hand-side to the left-hand-side.

This asymmetry is important for termination purposes, as we never want to create larger terms during a rewriting based proof. What ”larger” means in this context is not specified by SawScript: In fact, SawScript will not check that the given set of rewrite rules will reduce the sizes of the expressions as they are applied, so loops are certainly possible. For instance, a rule of the form:

```
rule badRule: forAll {x:[a], y:[a]}. x + y -> y + x;
```

would cause non-termination, as it will immediately create an expression that will trigger the rule again, ad infinitum. It is the user's responsibility to make sure the rules are specified in a manner not to trigger such behavior.

The other important thing to note about rules is that their correctness is not checked by SawScript. Once the rewrite rule is defined, it is taken as is, regardless whether it actually represents a sound reduction. For instance, a rule of the form:

```
rule unsound: forAll {x:[a], y:[a]}. x + y -> y - x;
```

would render the system unsound. Note that this is not a fundamental restriction on the system, but rather a limitation of the current implementation. In the future, we plan to incorporate means of verifying the correctness of rules themselves as well, at least for certain monomorphic instances.

We conclude this section with a collection of example rules to give a flavor of what is expressible.

Array operations The following rule captures how arrays operate:

```
rule getSet : forAll { a:[l]e, i:[idx], j:[idx], x:e }.
    aget(aset(a, i, x), j) -> if i == j then x else aget(a, j);
```

The rule states that if we update the element at index `i` of an array `a` with the value `x`, and read that entry out, we should get back `x`. If we read some other index (i.e., when $i \neq j$), then we should get back whatever was stored in the array before the update took place. Note how the rule is written polymorphically over arbitrary length arrays, indexes, and element types.

Rules about Cryptol functions It is possible to state rules about functions that are imported into SawScript via Cryptol's SBV files. For instance, we can assert the following rule about the `cry_fadd` function that we have imported from Cryptol in Section 3.2:

```
rule fieldAddAssoc : forAll {x:[384], y:[384], z:[384]}.
  cry_fadd (cry_fadd (x, y), z) -> cry_fadd (x, cry_fadd (y, z));
```

Simplifying `if` expressions As we have mentioned before, the rewriter comes with no hard-coded rules, hence the following rules come in handy in simplifying `if-then-else` expressions:

```
  rule ifTrue : forAll {x:a, y:a}.     if true  then x else y -> x;
  rule ifFalse: forAll {x:a, y:a}.     if false then x else y -> y;
  rule ifNot  : forAll {c:Bit, d:Bit}. if c then d else true  -> not c || d;
  rule ifId   : forAll {c:Bit, d:Bit}. if c then true else d  -> c || d;
  rule ifEq   : forAll {c:Bit, x:a}.   if c then x else x     -> x;
```

If-then-else congruences The following rules manipulate `if-then-else` expressions, pushing proof obligations to leafs:

```
rule iteEq : forAll {b:Bit, x:a, y:a, z:a}.
    (if b then x else y) == z -> if b then (x == z) else (y == z);
rule iteSplit: forAll {b:Bit, x:[384], y:[384]}.
    split(if b then x else y) : [12][32] ->
       if b then split(x) : [12][32] else split(y) : [12][32];
rule iteJoin: forAll {b:Bit, x:[12][32], y:[12][32]}.
    join(if b then x else y) -> if b then join(x) else join(y);
```

Boolean operations Several rules relating how boolean connectives work:

```
  rule and_true_elim1 : forAll {x:Bit}. true && x  -> x;
  rule and_true_elim2 : forAll {x:Bit}. x && true  -> x;
  rule and_false_elim1: forAll {x:Bit}. false && x -> false;
  rule and_false_elim2: forAll {x:Bit}. x && false -> false;
```

Record extraction SawScript supports structural records, similar to those found in Cryptol. The following rules capture some field-selection properties related to records:

```
  rule selX: forAll { xV:a, yV:b }. {x = xV; y = yV}.x -> xV;
  rule selY: forAll { xV:a, yV:b }. {x = xV; y = yV}.y -> yV;
```

We urge the reader to refer to the SawScript distribution for a larger collection of rules that come in handy for constructing proofs about ECC operations.

4.5. Disabling/enabling rules

As proof scripts get larger, users will inevitably find themselves needing tools for managing their SawScript files. In particular, the pair of directives:

```
  disable r;
  enable r;
```

will tell the rewriting engine to stop/start using the rule named r, respectively. (When a rule is first defined, it is enabled by default.)

Similarly, we can also start/stop proofs as well. The directive

```
  set verification off;
```

turns of proofs at that point, globally. This is useful for skipping over a number of methods as far as proofs are concerned, so you can focus on later method specs in your proof script. Analogously, you can use the directive

```
  set verification on;
```

to re-enable verification.

5. Summary

In this document, we have introduced the SawScript language, providing a means for end users to script functional equivalence proofs using Galois's Java Verifier and Cryptol technologies. While we have designed SawScript to be language agnostic, it is clearly closely tied to Java and Cryptol for the time being. However, we do expect to be able to leverage this work and support other languages in the future as well, such as LLVM or C. Furthermore, our current focus was on proving properties of programs that come up in the domain of elliptic-curve cryptography. Research and development activities targeting extension to other domains of interest are clearly worth pursuing as well.

While this document is intended to serve as an introductory material on basic SawScript concepts, becoming a proficient SawScript user will require investment from end users, as with any other language or proof system. It is our hope that SawScript will make formal proofs feasible for everyday reasoning tasks, enabling the evaluators to achieve their goals in a more automated fashion. Any feedback, feature requests, or comments on SawScript is therefore most appreciated.

A. SawScript operators

The following table lists all primitive SawScript operators.

Operator	Associativity	Notes
not, ~	right	Boolean negation, and bitwise complement
.	left	Record field selection
:	left	Type annotation
*, /s, %s	left	Multiplication, signed division and remainder
+, -	left	Addition and subtraction
<<, >>s, >>u	left	Shift left, signed/unsigned shift right
&	left	Bit-wise and
^	left	Bit-wise exclusive-or
|	left	Bit-wise or
#	right	Concatenation
==, !=	none	Boolean equality, inequality
>=s, >=u, >s, >u, <=s, <=u, <s, <u	none	Signed/unsigned comparisons
&&	left	Boolean conjunction
||	left	Boolean disjunction

Figure 2: SawScript operators: Higher precedence operators are listed first. Operators in the same row are at the same precedence level, use parentheses to disambiguate if necessary. Usual associativity rules do apply as well, as indicated in the table.

Modular arithmetic All arithmetic (*, +, -, /s, %s, etc.) is performed modulo 2^s, where s is the bit-width of the operands. Both operands must have the same width.

Signed/unsigned operators Some operators come in both signed and unsigned versions, such as comparisons (>u, >s, etc.), while some have only one version, such as shift-left (<<) or multiplication (*). SawScript provides separate operators when the signed versions are operationally different than unsigned ones, providing only one operator if the signedness does not matter. To make the signedness absolutely clear, all operators that have different semantics for signed/unsigned versions are tagged with the letters s and u, respectively.

Missing operators Note that several operators are missing from this table, including unsigned division and remainder (/u, %u), and left/right rotations. We plan to add these operators in future versions of SawScript.

References

[1] Cryptol web site. `http://cryptol.net`.

[2] Certicom Research: SEC 2: Recommended Elliptic Curve Domain Parameters, Standards for Efficient Cryptography. `http://www.secg.org/collateral/sec2_final.pdf`, 2000.

[3] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV'10*, pages 24–40, 2010.

[4] L. Erkök. Equivalence and safety checking in Cryptol. Technical report, Galois, Inc., Aug. 2008.

[5] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Programming Languages meets Program Verification, PLPV'09, Savannah, Georgia, USA*, pages 73–81. ACM Press, Jan. 2009.

[6] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.