Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2014-007 April 22, 2014

Symbolic Execution for (Almost) Free: Hijacking an Existing Implementation to Perform Symbolic Execution

Joseph P. Near and Daniel Jackson

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Symbolic Execution for (Almost) Free

Hijacking an Existing Implementation to Perform Symbolic Execution

Joseph P. Near Daniel Jackson

Massachusetts Institute of Technology
jnear,dnj@csail.mit.edu

Abstract

Symbolic execution of a language is traditionally achieved by replacing the language's interpreter with an entirely new interpreter. This may be an unnecessary burden, and it is tempting instead to try to use as much of the existing interpret infrastructure as possible, both for handling aspects of the computation that are not symbolic, and for propagating symbolic ones.

This approach was used to implement Rubicon, a bounded verification system for Ruby on Rails web applications, in less than 1000 lines of Ruby code. Rubicon uses symbolic execution to derive verification conditions from Rails applications and an off-the-shelf solver to check them. Despite its small size, Rubicon has been used to find previously unknown bugs in open-source Rails applications.

The key idea is to encode symbolic values and operations in a *library* written *in the target language itself*, overriding only a small part of the standard interpreter. We formalize this approach, showing that replacing a few key operators with symbolic versions in a standard interpreter gives the same effect as replacing the entire interpreter with a symbolic one.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords symbolic execution, domain-specific languages, static analysis, web programming

1. Introduction

Symbolic execution [13, 22] is one of the oldest strategies for reasoning about programs, and yet still forms the basis of many modern tools [6, 17, 21, 27, 30, 31].

Building a symbolic executor, however, is difficult, and building one that also handles concrete computation efficiently is more difficult still. Analysis tools based on symbolic execution, such as the symbolic extension for Java PathFinder [21, 27] and the CUTE concolic testing engine for C [31], are capable of analyzing real-world programs, but these tools are themselves large projects comprising hundreds of thousands of lines of code.

The similarity of symbolic execution and standard execution (indeed, the sharing of the very term "execution") suggests a simpler approach, in which the standard engine is used to propagate symbolic values, and to compute in the normal way with concrete values when available. The uniquely symbolic component is achieved by introducing a *library* written *in the target language itself*. Such a library comprises an encoding of symbolic values and new symbolic definitions for the primitive operations of the language, and effectively transforms the standard (concrete) implementation of the target language into a symbolic executor.

Implementing symbolic execution as a library means that concrete parts of the target program execute at full speed, just as they would during concrete execution. As a result, even large programs become amenable to symbolic execution if the number of symbolic inputs is small. At the same time, the library-based approach eliminates much of the burden of building a specialized symbolic execution engine, since a small number of primitive definitions often suffice to extend symbolic execution to the entire language.

We used this approach to build Rubicon [25], a scalable bounded verification system for Ruby on Rails web applications, in fewer than 1000 lines of Ruby code. Rubicon uses symbolic execution to derive verification conditions from specifications of a web application's behavior and its implementation, then uses the Alloy Analyzer to check the derived conditions. Because web applications typically have only a few variable inputs, the number of symbolic inputs to Rubicon's symbolic execution is low, and so Rubicon's analysis scales to large applications. Despite its small size, Rubicon has been used to find previously unknown bugs in open-source Rails applications.

The contributions of this paper include:

- A new technique for implementing symbolic execution, in which symbolic values and operations are encapsulated in a library written in the target language itself;
- A formal description of this approach, applied to the untyped λ-calculus with side effects;
- An informal proof that the formal description provided corresponds to the standard semantics of symbolic execution;
- A description of the implementation of Rubicon, a tool that performs bounded verification of Ruby-on-Rails web applications using this strategy.

2. Rubicon

Rubicon [25] is a library for writing and checking specifications for Ruby on Rails [18] web applications. Rubicon provides an embedded domain-specific language for programmers to specify web application behavior, and performs automated bounded analysis to check those specifications against the application's implementation.

Rubicon's specification language is based on the RSpec [9] testing library for Ruby; Rubicon extends RSpec with first-order quantifiers to allow the expression of general specifications. Rubicon's

similarity to RSpec is intentional, and is intended to encourage RSpec users to write specifications. The following is an example of a Rubicon specification for the open-source customer relationship management system Fat Free CRM:

```
describe ContactsController do
  it "should not display other users' private
      opportunities" do
    User.forall do |u|
      Contact.forall do |c|
        Opportunity.forall do |o|
          set_current_user(u)
          get :show, :id => c.id
          (o.access == 'private' and
           o.user != u).implies do
            assigns[:contact].opportunities should_not
              include o
          end
        end
      end
    end
  end
end
```

This specification checks that when a contact is displayed, the private opportunities associated with the contact are *not* displayed unless the logged-in user owns them. Rubicon's symbolic evaluator transforms this specification into the following verification condition:

```
Exp(forall, [User,
 Exp(forall, [Contact,
  Exp(forall, [Opportunity,
   Exp(implies, [
    Exp(and, [Exp(==, [Exp(field_ref, [o, :access]),
                       'private']),
              Exp(!=, [Exp(field_ref, [o, :user]),
                       u])]),
    Exp(not,
     [Exp(include, [Exp(field_ref, [c,
          :opportunities]),
                    o])])])])])])
```

Rubicon uses this simple abstract-syntax tree representation internally; a Rubicon user would instead see the following pretty-printed verification condition:

```
all u: User, c: Contact, o: Opportunity |
  o.access = 'private' and o.user != u implies
    not (o in c.opportunities)
```

This formula is obviously false, meaning that Fat Free CRM has a bug. Any value for c such that $c.opportunities$ contains a private opportunity not owned by u represents a counterexample. This situation does, in fact, reflect a bug in Fat Free CRM: after checking the permissions on the *contact* being displayed, the system displays all of the associated *opportunities*, without checking their permissions. This bug was previously unknown, and has since been fixed by the Fat Free CRM developers.

Rubicon's symbolic execution is implemented as a library in fewer than 1000 lines of Ruby. Symbolic variables are represented using a distinguished class of objects, and methods invoked on those objects are defined so as to return symbolic expressions like the one listed above. Having loaded the Rubicon library, executing the specification above *in the standard Ruby interpreter* produces the associated symbolic expression. Taking this approach greatly reduced both the size and development time of Rubicon.

2.1 A Simple Symbolic Evaluator

To illustrate the ease with which symbolic execution can be implemented in Ruby, we construct a simple symbolic evaluator for side-effect free programs. We first introduce a class to represent symbolic values, and a descendent of that class to represent symbolic expressions:

```
class SymbolicObject
  def method_missing(meth, *args)
    Exp.new(meth, [self] + args)
  end

  def ==(other)
    Exp.new(:equals, [self, other])
  end
end

class Exp < SymbolicObject
  def initialize(rator, rands)
    @rator = rator
    @rands = rands
  end
end
```

An instance of "SymbolicObject" represents a symbolic variable. The class defines the "method_missing" method so that an arbitrary method invocation on a symbolic object yields a symbolic expression representing that invocation. For example, the following program:

```
x = SymbolicObject.new
y = SymbolicObject.new
x.foo(y)
```

Produces the following symbolic expression:

```
Exp(foo, [x, y])
```

We have also defined the "==" method, because invoking this method will not trigger a call to "method_missing" since equality is defined for all objects. Consider the following similar program, for example:

```
x = SymbolicObject.new
y = SymbolicObject.new
x + y == y + x
```

This program produces the symbolic expression:

```
Exp(==, [Exp(+, [x, y]), Exp(+, [y, x])])
```

2.1.1 Conditionals

Handling conditionals is a key part of symbolic execution, since the system must execute both branches of conditional that depends on

a symbolic value. In the ideal implementation of Ruby, we could write the following definition of "**if**" as a call-by-name function:

```
def if(condition, then_do, else_do)
  c = condition.call
  if c.is_a? SymbolicObject then
    Exp.new(:if, [c, then_do.call, else_do.call])
  else
    if c then then_do.call else else_do.call end
  end
end
```

This definition would enable the user to write code like the following:

```
x = SymbolicObject.new
if x.even? then
  (x+1).odd?
end
```

Which would evaluate to the following symbolic expression:

```
Exp(if, [Exp(even?, [x]), Exp(odd?, [Exp(+, [x, 1])])])
```

The reality of Ruby's implementation makes handling conditionals slightly more complicated. By default, Ruby does not allow the programmer to redefine "**if**." To solve this problem, we use a library called VIRTUAL_KEYWORDS [29], which was developed with the motivation of handling conditionals in Rubicon. VIRTUAL_KEYWORDS allows the programmer to redefine Ruby's hard-coded keywords, including "**if**." When one of these redefinitions is called, its arguments are passed inside of blocks to simulate call-by-name function invocation.

2.2 A More Complicated Evaluator: Handling Side Effects

Supporting side effects in the presence of symbolic values requires a significant change to the way conditionals are handled. Since both branches of the conditional may contain updates to the same variable, it becomes necessary to save both values, along with the *path condition* under which the variable takes a particular value.

We begin by adding a representation of symbolic state, which we store in symbolic objects themselves. We add a method to symbolic objects that adds a new possible value, along with the associated path condition, to the symbolic state:

```
class SymbolicObject
  def initialize
    @vals = []
  end

  def add_val(cond, val)
    @vals << [cond,val]
  end
end
```

To handle side effects properly, the new definition of "**if**" must save the current state, execute the conditional's first branch, update the symbolic state based on the updates made during that execution, and repeat the process for the second branch. In addition, the path condition must be set appropriately for the execution of each branch, and used in updating the symbolic state.

```
def get_state(binding)
  Hash[eval(" local_variables ", binding).
       map{|var| [var, eval(var, binding)]}]
end

def save_state(binding)
  state = get_state(binding)

  state.each_pair do |var, val|
    eval(var + "_old = " + var, binding)
  end

  get_state(binding)
end


def update_state(state, state1, binding)
  state1.each_pair do |var, val|
    if val.equal? state[var] then
      # no change
    else
      if state[var].is_a? SymbolicObject then
        eval(var + " = " + var + "_old", binding)
        state[var].add_val($path_condition, val)
      else
        eval(var + " = SymbolicObject.new", binding)
        new_obj = eval(var, binding)
        new_obj.add_val(true, state[var]) if state[var]
        new_obj.add_val($path_condition, val)
      end
    end
  end
end

def if(condition, then_do, else_do)
  c = condition.call
  if c.is_a? SymbolicObject then
    pc = $path_condition

    state = save_state(then_do.binding)
    $path_condition = Exp.new(:and, [c, pc])
    v1 = then_do.call
    state1 = get_state(then_do.binding)
    update_state(state, state1, then_do.binding)

    state = save_state(else_do.binding)
    $path_condition = Exp.new(:and, [Exp.new(:not,
        [c]), pc])
    v2 = else_do.call
    state2 = get_state(else_do.binding)
    update_state(state, state2, else_do.binding)

    $path_condition = pc
    Exp.new(:if, [c, v1, v2])
  else
    if c then then_do.call else else_do.call end
  end
end
```

Figure 1. Redefinition of "**if**" to Handle Side Effects

Figure 1 contains a definition of "**if**" that handles side effects. It works by saving the current state, updating the path condition based on the branch being executed, executing the branch, and updating the symbolic state based on the updates to the concrete state. Because it is impossible to set the value of a variable stored in a Ruby Binding object directly, the "save_state" procedure saves a copy of each variable with the suffix "_old" and the "update_state" procedure uses these copies to retrieve previous values of updated variables.

This redefinition allows the programmer to write code like the following:

```
x = SymbolicObject.new
y = SymbolicObject.new
if x.even? then
  y = true
else
  y = false
end
```

Executing this program gives the variable "y" the following symbolic state:

```
Exp(even?, [x]) => true,
Exp(not, [Exp(even?, [x])]) => false
```

This symbolic state represents the two possibilities for "y:" if "x" is even, then the value of "y" will be **true**; otherwise, it will be **false**.

2.3 Stubbing Rails

Rails web applications interact with a persistent database through ActiveRecord, an object-relational mapper. In general, the properties of Rails applications that Rubicon checks should be true for all configurations of the database. As a result, Rubicon must treat the database as symbolic data when checking properties.

Fortunately, Rails enforces the use of the ActiveRecord interface for accessing the database, so we can simply provide a new implementation of that interface which returns symbolic values instead of actual database records. In a Rails application, objects to be stored in the database extend ActiveRecord, and the ActiveRecord class provides methods such as "find" and "all" to query the database for records representing objects of the receiver's type. For example, given the following User class:

```
class User < ActiveRecord::Base
end
```

A Rails application could find all users with the name "Joe" using the following expression, which evaluates to a list of records with the given property:

```
User.find :name => "Joe"
```

Our goal is for expressions like these to evaluate to symbolic expressions representing the database query itself. The obvious way to accomplish this is to use Ruby's open classes to redefine "find" and the other querying methods of ActiveRecord. Unfortunately, Rails prevents this approach by defining the methods on ActiveRecord objects dynamically. At runtime, then, our redefinitions would be overwritten with the originals as defined by Rails. Rails defines methods dynamically so that each ActiveRecord object responds to a set of methods representing the fields of the corresponding database records. Given a User object "u," for example, the expression u.name evaluates to the name field of the database record corresponding to the user "u."

```
1  class TypedSymbolicObject < SymbolicObject
2    def initialize (type)
3      @type = type
4    end
5  end
6
7  klasses = ActiveRecord::Base.descendants
8
9  klasses.each do |klass|
10   metaklass = class << klass; self; end
11   metaklass.send(:define_method, :new, lambda {
         TypedSymbolicObject.new(self) })
12   metaklass.send(:define_method, :my, lambda {
         TypedSymbolicObject.new(self) })
13   metaklass.send(:define_method, :all, lambda {
         TypedSymbolicObject.new(self) })
14   ...
15
16   klass.column_names.each do |name|
17     klass.send(:define_method, name.to_sym, lambda {
           Exp.new(:field_get, [self, name.to_sym]) })
18     klass.send(:define_method, (name + "=").to_sym,
           lambda {|arg| Exp.new(:field_set, [self,
           name.to_sym, arg]) })
19   end
20
21   klass.  reflect_on_all_associations  .each do |assoc|
22     klass.send(:define_method, assoc.name, lambda {
           Exp.new(:field_get, [self, assoc.name]) })
23   end
24 end
```

Figure 2. Code to Stub Rails Database Accessor Methods

The solution is to employ dynamic redefinition ourselves. We redefine each instance method corresponding to a database field so as to return a symbolic expression, and we redefine the class methods for constructing database queries to return symbolic queries.

Figure 2 contains the code used to perform this step, along with a definition of typed symbolic objects. The code works by redefining both the class methods and instance methods of ActiveRecord's descendents. Lines 10-13 show how some of the database query methods are redefined. Lines 16-19 redefine the getters and setters for the database field methods of the class, and lines 21-23 do the same for the *associations*—relationships with other objects through a separate database table—belonging to the class.

With this redefinition, we can symbolically execute code like the following definition of the "show" method of the "User" controller of Fat Free CRM, which method starts by fetching the user associated with the provided ID:

```
class UsersController < ApplicationController
  def show
    @user = User.my.find(params[:id])
    ...
  end
end
```

Given a symbolic ID, the "@user" variable will get the value:

```
Exp(:find, [Exp(:query, [User]), SymbolicObject1])
```

2.4 Writing Specifications

Rubicon also provides the user with a language for writing down properties to be checked. Rubicon borrows much of this language from RSpec, with" forall " being the only new language feature. We also redefine RSpec's "should" method to produce symbolic expressions.

2.4.1 Quantifiers

Symbolic values in Rubicon specifications are constructed through the use of quantifiers. We define the " forall " quantifier, for example, by using Ruby's open classes facility to define a new method for all objects:

```
class Object
  def forall (&block)
    Exp.new(:forall, [self,
        block.call(TypedSymbolicObject.new(self))])
  end
end
```

This definition allows the programmer to quantify over objects; the typical use is to quantify over a class. The " forall " method takes a block and invokes it, passing a new symbolic object whose type information is represented by the receiver of the method call. For example:

```
Integer.forall do |i|
  Integer.forall do |j|
    i + j == j + i
  end
end
```

This expression evaluates to the following symbolic expression:

```
Exp(forall, [Integer, Exp(forall, [Integer,
    Exp(equals, [Exp(+, [i, j]), Exp(+, [j, i])])])])
```

2.4.2 Should

RSpec provides the "should" and "should_not" methods to construct assertions. We redefine these methods to construct symbolic assertions as follows:

```
class SymbolicObject
  def should(matcher = nil, message = nil)
    Exp.new(:should, [self,
        matcher.instance_variable_get('@name'),
        matcher.instance_variable_get('@expected')])
  end

  def should_not(matcher = nil, message = nil)
    Exp.new(:should_not, [self,
        matcher.instance_variable_get('@name'),
        matcher.instance_variable_get('@expected')])
  end
end
```

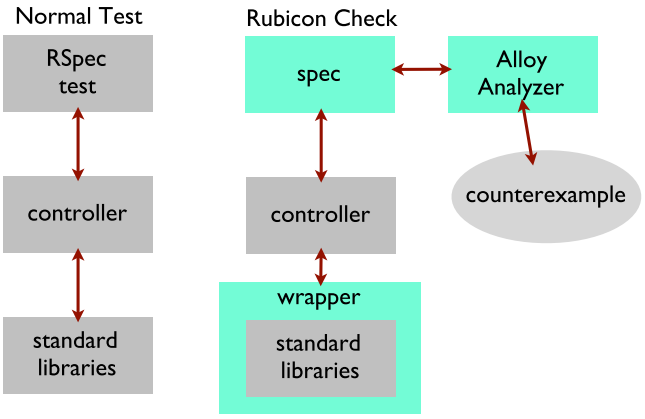

Figure 3. Rubicon Specification Compared to RSpec Test

The following expression says that all users should be included in the list of users obtained from the database:

```
User.forall do |u|
  User.all.should include? u
end
```

This expression evaluates to the following symbolic expression:

```
Exp(:should, [Exp(:all, [User]), include?,
    SymbolicObject1])
```

2.5 Putting it All Together

The diagram in Figure 3 compares the model for RSpec testing to that of Rubicon's specifications. Like RSpec tests, Rubicon specifications invoke Rails controllers, and through them, the standard Rails and Ruby libraries. Unlike RSpec tests, however, Rubicon specifications are written using domain-specific language constructs that produce symbolic expressions, and are executed in an environment that wraps the standard libraries, allowing those libraries to compute with symbolic values.

The remaining piece of Rubicon's architecture is the constraint solver used to solve the verification conditions generated by Rubicon's symbolic execution. Rubicon uses the Alloy Analyzer [34], a bounded analysis tool for the Alloy language [20]. Alloy is a specification language based on first-order relational logic with transitive closure. The language is a perfect match for the relational-database-oriented verification conditions generated by Rubicon, and the bounded analysis performed by the Alloy Analyzer is guaranteed to terminate even in the presence of quantifiers.

Another constraint solver, such as an SMT solver, model checker, or even interactive theorem prover, could just as easily fill this role. In exchange for giving up the Alloy Analyzer's push-button automation, another solver may be able to provide an unbounded proof of Rubicon's verification conditions.

3. Formalizing Rubicon

To illustrate our approach, we apply it to the untyped λ-calculus and compare the resulting embedding with the standard symbolic semantics. For simplicity, we begin with a side-effect free target language, to which we later add side effects.

3.1 Untyped λ-calculus

In this section, we apply our approach to the side-effect-free untyped λ-calculus. We adopt the standard syntax and semantics, as well as the conventions, of Pierce [28]; we present the syntax and semantics in Figure 4.

We formalize the standard notion of symbolic execution in Figure 5. This formalization adds symbolic values to the existing concrete ones, and adds evaluation rules for terms containing symbolic values. We use s to denote the class of symbolic values, which may be either symbolic variables or symbolic expressions containing an arbitrary number of symbolic or concrete values. The result of a symbolic execution under these semantics will be a concrete value if no symbolic values are involved, or a symbolic expression if computation using symbols is performed.

Because the target language omits side effects, no notion of symbolic state or a global path constraint is required. The resulting symbolic expression itself represents the condition necessary for the given term to yield a particular value. For example, consider the following term and its value under the symbolic semantics:

$$\begin{array}{lcl} t & = & \mathsf{if\ iszero}\ s_1 \mathsf{\ then\ 0\ else\ succ\ 0} \\ & \stackrel{*}{\rightarrow} & \mathsf{Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \end{array}$$

The resulting symbolic expression alone represents the possible results of executing t concretely: if the input variable is zero, then the result is zero; otherwise, it is zero's successor.

Figure 6 contains our approach to symbolic execution for the pure untyped λ-calculus. Our approach comprises a set of symbolic values—identical to the ones added in Figure 5—and a set of redefinitions for the primitive operations of the language. Aside from the addition of symbolic values, this implementation can be executed directly under the standard semantics in Figure 4. This approach also works in languages without symbolic values, by encoding symbolic values in a form the language does support. Our implementation in Ruby, for example, uses a special set of classes to represent symbolic values.

Using our approach also requires the ability to selectively redefine language primitives, including "if." Many existing languages make this possible, and there are workarounds for some of those (like Ruby) that do not. When programs are side-effect free, invocations of these primitives can be either call-by-value or call-by-name; when side effects are introduced, call-by-name must be used.

Our approach produces the same results as the traditional symbolic semantics shown in Figure 5. The example given above, for example, evaluates as follows under the standard semantics with our redefinitions:

$$\begin{array}{lcl} \mathrm{t} & = & \mathsf{if\ iszero}\ s_1\ \mathsf{then\ 0\ else\ succ\ 0} \\ & \stackrel{*}{\rightarrow}_\theta & \mathsf{if\ sym?\ Exp(iszero,}\ s_1\mathsf{)} \\ & & \quad \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \\ & & \quad \mathsf{else\ if\ (iszero}\ s_1\mathsf{)\ then\ 0\ else\ succ\ 0} \\ & \rightarrow_\theta & \mathsf{if\ true} \\ & & \quad \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \\ & & \quad \mathsf{else\ if\ (iszero}\ s_1\mathsf{)\ then\ 0\ else\ succ\ 0} \\ & \rightarrow_\theta & \mathsf{then\ Exp(if,\ Exp(iszero,}\ s_1\mathsf{),\ 0,\ succ\ 0)} \end{array}$$

The proof that our approach corresponds to the symbolic semantics is straightforward, and is accomplished by induction on terms. A short version of this proof, considering only the relevant cases, follows.

THEOREM 3.1. *Let $\rightarrow$ be the transition relation of the symbolic semantics described in Figure 5, and let $\rightarrow_\theta$ be the transition relation of the standard semantics plus redefinitions of primitive operations described in Figure 6. Then for all terms t, $t \stackrel{*}{\rightarrow} t' \iff t \stackrel{*}{\rightarrow}_\theta t'$.*

$$\begin{array}{rcl} t & ::= & x \mid \lambda x.t \mid t\ t \\ & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & 0 \mid \mathsf{succ}\ t \mid \mathsf{pred}\ t \mid \mathsf{iszero}\ t \\ \\ v & ::= & \lambda x.t \\ & \mid & \mathsf{true} \mid \mathsf{false} \\ & \mid & nv \\ \\ nv & ::= & 0 \mid \mathsf{succ}\ nv \end{array}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 t_2 \rightarrow t_1' t_2}\ (\textsc{App1}) \qquad \frac{t_2 \rightarrow t_2'}{v\ t_2 \rightarrow v\ t_2'}\ (\textsc{App2})$$

$$(\lambda x.t)v \rightarrow [x \mapsto v]t \quad (\textsc{AppAbs})$$

$$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_2 \quad (\textsc{IfTrue})$$

$$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \rightarrow t_3 \quad (\textsc{IfFalse})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}\ (\textsc{If})$$

$$\mathsf{pred}\ 0 \rightarrow 0 \quad (\textsc{PredZero})$$

$$\mathsf{iszero}\ 0 \rightarrow \mathsf{true} \quad (\textsc{IsZeroZero})$$

$$\mathsf{iszero}\ (\mathsf{succ}\ nv_1) \rightarrow \mathsf{false} \quad (\textsc{IsZeroSucc})$$

$$\mathsf{pred}\ (\mathsf{succ}\ nv_1) \rightarrow nv_1 \quad (\textsc{PredSucc})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{succ}\ t_1 \rightarrow \mathsf{succ}\ t_1'}\ (\textsc{Succ})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{pred}\ t_1 \rightarrow \mathsf{pred}\ t_1'}\ (\textsc{Pred})$$

Figure 4. Syntax and Reduction Rules for Untyped λ-calculus with Booleans and Natural Numbers

Proof By induction on t, considering the relevant cases.

- $t = \mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3$. We have:

  $$t \stackrel{*}{\rightarrow} \mathsf{Exp(if,}\ sv_1, v_2, v_3\mathsf{)}$$

  and:

  $$\begin{array}{lcl} t & \rightarrow_\theta & \mathsf{if\ sym?}\ sv_1\ \mathsf{then\ Exp(if,}\ sv_1, v_2, v_3\mathsf{)} \\ & & \quad \mathsf{else\ if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3 \\ & \stackrel{*}{\rightarrow}_\theta & \mathsf{Exp(if,}\ sv_1, v_2, v_3\mathsf{)} \end{array}$$

  which are equivalent.

- $t = \mathsf{pred}\ sv_1$. We have:

  $$t \stackrel{*}{\rightarrow} \mathsf{Exp(pred,}\ sv_1\mathsf{)}$$

  and:

  $$\begin{array}{lcl} t & \rightarrow_\theta & \mathsf{if\ sym?}\ sv_1\ \mathsf{then\ Exp(pred,}\ sv_1\mathsf{)\ else\ pred}\ sv_1 \\ & \stackrel{*}{\rightarrow}_\theta & \mathsf{Exp(pred,}\ sv_1\mathsf{)} \end{array}$$

  which are equivalent.

$$
\begin{array}{rcl}
v & ::= & \ldots \\
 & | & sv \\
\\
sv & ::= & sx \mid \mathsf{Exp}\ v^*
\end{array}
$$

$$\frac{t_2 \rightarrow t_2'}{\mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \rightarrow \mathsf{if}\ sv_1\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3} \quad (\textsc{IfSymb1})$$

$$\frac{t_3 \rightarrow t_3'}{\mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ t_3 \rightarrow \mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ t_3'} \quad (\textsc{IfSymb2})$$

$$\mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3 \rightarrow \mathsf{Exp}(\mathsf{if}, sv_1, v_2, v_3) \quad (\textsc{IfSymb})$$

$$\mathsf{pred}\ sv_1 \rightarrow \mathsf{Exp}(\mathsf{pred}, sv_1) \quad (\textsc{PredSymb})$$

$$\mathsf{succ}\ sv_1 \rightarrow \mathsf{Exp}(\mathsf{succ}, sv_1) \quad (\textsc{SuccSymb})$$

$$\mathsf{iszero}\ sv_1 \rightarrow \mathsf{Exp}(\mathsf{iszero}, sv_1) \quad (\textsc{IsZeroSymb})$$

Figure 5. Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Booleans and Natural Numbers

$$
\begin{array}{rcl}
v & ::= & \ldots \\
 & | & sv \\
\\
sv & ::= & sx \mid \mathsf{Exp}\ v^*
\end{array}
$$

$$\mathsf{sym?}\ sv_1 \rightarrow \mathsf{true} \quad (\textsc{Sym})$$

$$\mathsf{sym?}\ v_1 \rightarrow \mathsf{false} \quad \text{where } v_1 \notin sv \quad (\textsc{NotSym})$$

$$\frac{t_1 \rightarrow t_1'}{\mathsf{sym?}\ t_1 \rightarrow \mathsf{sym?}\ t_1'} \quad (\textsc{Sym2})$$

$$
\begin{array}{rcl}
\mathsf{if} & = & \lambda\,\mathsf{t,c,a.\ if\ sym?\ t\ then\ Exp(if,\ t,\ c,\ a)\ else\ if\ t\ then\ c\ else\ a} \\
\mathsf{pred} & = & \lambda\,\mathsf{v.\ if\ sym?\ v\ then\ Exp(pred,\ v)\ else\ pred\ v} \\
\mathsf{succ} & = & \lambda\,\mathsf{v.\ if\ sym?\ v\ then\ Exp(succ,\ v)\ else\ succ\ v} \\
\mathsf{iszero} & = & \lambda\,\mathsf{v.\ if\ sym?\ v\ then\ Exp(iszero,\ v)\ else\ iszero\ v}
\end{array}
$$

Figure 6. Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus Under Standard Semantics

- Similarly for succ and iszero. ∎

3.2 Adding Side Effects

We now turn our attention to target languages with side effects. We present the standard semantics for the untyped λ-calculus with side effects in Figure 7. This formalization of mutable state introduces the set l of labels and the store μ to hold a mapping from labels to values. The corresponding reduction rules update the store as a given term is reduced, and the final result of a program is represented by both the fully-reduced value of the term and the final value of the store.

This new style of execution makes symbolic execution more difficult. Given symbolic inputs, a program with side effects should produce both a symbolic value and a store—but the value of the store depends on the path taken through the program. To perform symbolic execution, we introduce a new *symbolic store* σ that represents all the possible values a symbolic variable *could* take, and also records the conditions necessary for the variable to take each of those values.

Each condition recorded in the symbolic store represents a single path through the program, and is therefore called a *path constraint*. Symbolic execution keeps track of the current path constraint during execution, and uses that path constraint when updating the symbolic store.

We formalize the symbolic semantics with side effects in Figure 8. We call the current path constraint ϕ, and the symbolic store is σ. The majority of the reduction rules correspond to those of the standard semantics in Figure 7, except that the new rules propagate the values of ϕ and σ.

The rules for handling assignment and conditionals have changed significantly to deal with symbolic values. Intuitively, the rule for "**if**" must execute both branches of the conditional, constructing the appropriate path constraint for each branch. Rules IFSYMB1 and IFSYMB2 perform this task, using the condition's value to extend the path constraint. When both branches have evaluated to values, IFSYMB transforms the conditional into a symbolic expression.

The rules for assignment are responsible for extending and merging symbolic states. The ASSIGN rule handles the entirely concrete case, and operates just as before. ASSIGNSYMB1 handles situations in which the variable being assigned to is not symbolic, but its new value is dependent on a symbolic value, as in the following program:

```
1 x := 5;
2 if sv then x := 6
```

In this case, "x" must take a symbolic value, even though it is only assigned concrete values, since its value is dependent on the symbolic value "sv." ASSIGNSYMB1 constructs a new symbolic variable for this purpose, assigns that symbolic variable to the given location, and adds both possible values for the variable to the symbolic state.

The final case, handled by ASSIGNSYMB2, is the situation in which the target of an assignment is already symbolic. Consider the following program, for example:

```
1 x := 5;
2 if sv then x := 6
3 else x := 7
```

After executing the first assignment, the symbolic state for "x" will be:

$$(\mathsf{sv} \Rightarrow 6), (\mathsf{true} \wedge \neg\mathsf{sv} \Rightarrow 5)$$

For the second assignment, since "x" already has a symbolic value, we take its symbolic state and duplicate it. One copy of the symbolic state has is path conditions conjuncted with the current path condition, and its values replaced with the value being assigned (this part of the symbolic state represents all possible paths through the program *that end up going through the current path*). The other copy has its path conditions conjuncted with the *negation* of the current path condition, and its values remain unchanged (this part of the symbolic state represents the possible paths through the program that do *not* end up going through the current path). After the second assignment, then, the symbolic state for "x" is:

$$(\mathsf{sv} \wedge \neg\mathsf{sv} \Rightarrow 7), (\mathsf{true} \wedge \neg\mathsf{sv} \wedge \neg\mathsf{sv} \Rightarrow 7),$$
$$(\mathsf{sv} \wedge \mathsf{sv} \Rightarrow 6), (\mathsf{true} \wedge \neg\mathsf{sv} \wedge \mathsf{sv} \Rightarrow 5)$$

Of these possible outcomes, the first and the last are impossible, reflecting the fact that there is no way to take more than one path through the program simultaneously, and making sure that some path is taken (as a result, it is impossible for "x" to have the final value 5). The rule ASSIGNSYMB2 implements this duplication and updating process on the symbolic state, computing a σ' that correctly merges the possible symbolic states.

The end result of executing a program with symbolic inputs is a symbolic value, a store μ mapping labels to symbolic variables or concrete values, and a symbolic store σ mapping symbolic variables to sets of values paired with path conditions.

Figure 9 contains our primitive redefinitions that produce the same results as the symbolic semantics. Like the symbolic semantics, these redefinitions keep track of the current path constraint and symbolic state; in the absence of semantic constructs, however, these elements are encoded in the target language. The path constraint is stored in a global variable "pc," while the symbolic state is encoded as a data structure tagged with the unique value we label "SYM_TAG" and is treated symbolically by the redefined primitives.

Just as in the symbolic semantics, the major changes occur in the definitions of assignment and conditionals. The definition of assignment performs the same additions to the symbolic state as the rules of the symbolic semantics do, but the redefinition places these changes in tagged data structures inside the store, rather than in σ. Similarly, the redefinition of conditionals modifies the path constraint by updating its value in the store before executing the first branch of the conditional, updates the path constraint again before executing the second branch, and resets it before returning.

THEOREM 3.2. *Let $\rightarrow$ be the transition relation of the symbolic semantics described in Figure 8, let $\rightarrow_\theta$ be the transition relation of the standard semantics plus redefinitions of primitive operations described in Figure 9, and let γ be a concretization function encoding the contents of the store and symbolic state such that $\gamma(\mu,\sigma)(l) = (\mathsf{SYM\_TAG}, \sigma(\mu(l)))$ if $\mu(l) \in sx$, and $\gamma(\mu,\sigma)(l) = \mu(l)$ otherwise. Then for all terms t, $\phi \vdash t|\mu,\sigma \xrightarrow{*} t'|\mu',\sigma' \iff t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \xrightarrow{*}_\theta t'|(\gamma(\mu',\sigma'), \mathsf{pc} = \phi')$.*

Proof By induction on t, considering the "if" and assignment cases involving symbolic values.

CASE 1. $t = \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma$

By IFSYMB1, IFSYMB2, and IFSYMB, if:

$$\phi \wedge sv_1 \vdash t_2|\mu,\sigma \xrightarrow{*} v_2|\mu',\sigma'$$

$$\phi \wedge \neg sv_1 \vdash t_3|\mu',\sigma' \xrightarrow{*} v_3|\mu'',\sigma''$$

Then we have that:

$$\phi \vdash t|\mu,\sigma \xrightarrow{*} \mathsf{Exp}(sv_1, v_2, v_3)|\mu'',\sigma''$$

By $\rightarrow_\theta$, we have that:

$$\begin{array}{rcl} t & ::= & \ldots \\ & | & \mathsf{ref}\ t \mid !\,t \mid t := t \mid l \\ v & ::= & \ldots \\ & | & \mathsf{unit} \mid l \\ \mu & ::= & \emptyset \mid \mu, l = v \end{array}$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{t_1 t_2|\mu \rightarrow t_1' t_2|\mu'} \quad (\text{APP1}) \qquad \frac{t_2|\mu \rightarrow t_2'|\mu'}{v\ t_2|\mu \rightarrow v\ t_2'|\mu'} \quad (\text{APP2})$$

$$(\lambda x.t)v|\mu \rightarrow [x \mapsto v]t|\mu \quad (\text{APPABS})$$

$$\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3|\mu \rightarrow t_2|\mu \quad (\text{IFTRUE})$$

$$\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3|\mu \rightarrow t_3|\mu \quad (\text{IFFALSE})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu \rightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu'} \quad (\text{IF})$$

$$\mathsf{pred}\ 0|\mu \rightarrow 0|\mu \quad (\text{PREDZERO})$$

$$\mathsf{iszero}\ 0|\mu \rightarrow \mathsf{true}|\mu \quad (\text{ISZEROZERO})$$

$$\mathsf{iszero}\ (\mathsf{succ}\ nv_1)|\mu \rightarrow \mathsf{false}|\mu \quad (\text{ISZEROSUCC})$$

$$\mathsf{pred}\ (\mathsf{succ}\ nv_1)|\mu \rightarrow nv_1|\mu \quad (\text{PREDSUCC})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{\mathsf{succ}\ t_1|\mu \rightarrow \mathsf{succ}\ t_1'|\mu'} \quad (\text{SUCC})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{\mathsf{pred}\ t_1|\mu \rightarrow \mathsf{pred}\ t_1'|\mu'} \quad (\text{PRED})$$

$$\frac{l \notin dom(\mu)}{\mathsf{ref}\ v_1|\mu \rightarrow l|(\mu, l \mapsto v_1)} \quad (\text{REFV})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{\mathsf{ref}\ t_1|\mu \rightarrow \mathsf{ref}\ t_1'|\mu'} \quad (\text{REF})$$

$$\frac{\mu(l) = v}{!l|\mu \rightarrow v|\mu} \quad (\text{DEREFLOC})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{!t_1|\mu \rightarrow !t_1'|\mu'} \quad (\text{DEREF})$$

$$l{:=}v_2|\mu \rightarrow \mathsf{unit}|[l \mapsto v_2]\mu \quad (\text{ASSIGN})$$

$$\frac{t_1|\mu \rightarrow t_1'|\mu'}{t_1{:=}t_2|\mu \rightarrow t_1'{:=}t_2|\mu'} \quad (\text{ASSIGN1})$$

$$\frac{t_2|\mu \rightarrow t_2'|\mu'}{v_1{:=}t_2|\mu \rightarrow v_1{:=}t_2'|\mu'} \quad (\text{ASSIGN2})$$

Figure 7. Syntax and Reduction Rules for Untyped λ-calculus with Side Effects

$$
\begin{array}{rcl}
v & ::= & \ldots \\
 & | & sv \\
\\
sv & ::= & sx \mid \mathsf{Exp}\ v^* \\
\\
\sigma & ::= & \emptyset \mid \sigma, sx = \{\phi, v\}
\end{array}
$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash t_1 t_2|\mu,\sigma \to t_1' t_2|\mu',\sigma'} \quad (\textsc{App1})$$

$$\frac{\phi \vdash t_2|\mu,\sigma \to t_2'|\mu',\sigma'}{\phi \vdash v\ t_2|\mu,\sigma \to v\ t_2'|\mu',\sigma'} \quad (\textsc{App2})$$

$$\phi \vdash (\lambda x.t)v|\mu,\sigma \to [x \mapsto v]t|\mu,\sigma \quad (\textsc{AppAbs})$$

$$\phi \vdash \mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \to t_2|\mu,\sigma \quad (\textsc{IfTrue})$$

$$\phi \vdash \mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \to t_3|\mu,\sigma \quad (\textsc{IfFalse})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \to \\ \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu',\sigma'\end{array}} \quad (\textsc{If})$$

$$\phi \vdash \mathsf{pred}\ 0|\mu,\sigma \to 0|\mu,\sigma \quad (\textsc{PredZero})$$

$$\phi \vdash \mathsf{iszero}\ 0|\mu,\sigma \to \mathsf{true}|\mu,\sigma \quad (\textsc{IsZeroZero})$$

$$\phi \vdash \mathsf{iszero}\ (\mathsf{succ}\ nv_1)|\mu,\sigma \to \mathsf{false}|\mu,\sigma \quad (\textsc{IsZeroSucc})$$

$$\phi \vdash \mathsf{pred}\ (\mathsf{succ}\ nv_1)|\mu,\sigma \to nv_1|\mu,\sigma \quad (\textsc{PredSucc})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash \mathsf{succ}\ t_1|\mu,\sigma \to \mathsf{succ}\ t_1'|\mu',\sigma'} \quad (\textsc{Succ})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash \mathsf{pred}\ t_1|\mu,\sigma \to \mathsf{pred}\ t_1'|\mu',\sigma'} \quad (\textsc{Pred})$$

$$\frac{l \notin dom(\mu)}{\phi \vdash \mathsf{ref}\ v_1|\mu,\sigma \to l|(\mu, l \mapsto v_1),\sigma} \quad (\textsc{RefV})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash \mathsf{ref}\ t_1|\mu,\sigma \to \mathsf{ref}\ t_1'|\mu',\sigma'} \quad (\textsc{Ref})$$

$$\frac{\mu(l) = v}{\phi \vdash !l|\mu,\sigma \to v|\mu,\sigma} \quad (\textsc{DerefLoc})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash !t_1|\mu,\sigma \to !t_1'|\mu',\sigma'} \quad (\textsc{Deref})$$

$$\frac{\phi \vdash t_1|\mu,\sigma \to t_1'|\mu',\sigma'}{\phi \vdash t_1{:=}t_2|\mu,\sigma \to t_1'{:=}t_2|\mu',\sigma'} \quad (\textsc{Assign1})$$

$$\frac{\phi \vdash t_2|\mu,\sigma \to t_2'|\mu',\sigma'}{\phi \vdash v_1{:=}t_2|\mu,\sigma \to v_1{:=}t_2'|\mu',\sigma'} \quad (\textsc{Assign2})$$

$$\frac{\mu(l) \notin sx}{\mathsf{true} \vdash l{:=}v_2|\mu,\sigma \to \mathsf{unit}|[l \mapsto v_2]\mu,\sigma} \quad (\textsc{Assign})$$

$$\frac{\mu(l) \notin sx \qquad \phi \neq \mathsf{true} \qquad sx_1 \notin dom(\sigma)}{\begin{array}{c}\phi \vdash l{:=}v_2|\mu,\sigma \to \\ \mathsf{unit}|[l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})\end{array}} \quad (\textsc{AssignSymb1})$$

$$\frac{\mu(l) = sx_1}{\begin{array}{c}\phi \vdash l{:=}v_2|\mu,\sigma \to \mathsf{unit}|\mu,\sigma' \\ \text{where } \sigma' = [sx_1 \mapsto \{(\phi \wedge \phi', v_2)|(\phi', v) \in \sigma(sx_1)\} \cup \\ \qquad \{(\neg\phi \wedge \phi', v)|(\phi', v) \in \sigma(sx_1)\}]\sigma\end{array}} \quad (\textsc{AssignSymb2})$$

$$\phi \vdash \mathsf{pred}\ sv_1|\mu,\sigma \to \mathsf{Exp}(\mathsf{pred}, sv_1)|\mu,\sigma \quad (\textsc{PredSymb})$$

$$\phi \vdash \mathsf{succ}\ sv_1|\mu,\sigma \to \mathsf{Exp}(\mathsf{succ}, sv_1)|\mu,\sigma \quad (\textsc{SuccSymb})$$

$$\phi \vdash \mathsf{iszero}\ sv_1|\mu,\sigma \to \mathsf{Exp}(\mathsf{iszero}, sv_1)|\mu,\sigma \quad (\textsc{IsZeroSymb})$$

$$\frac{\phi \wedge sv_1 \vdash t_2|\mu,\sigma \to t_2'|\mu',\sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \to \\ \mathsf{if}\ sv_1\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3|\mu',\sigma'\end{array}} \quad (\textsc{IfSymb1})$$

$$\frac{\phi \wedge \neg sv_1 \vdash t_3|\mu,\sigma \to t_3'|\mu',\sigma'}{\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3|\mu,\sigma \to \\ \mathsf{if}\ sv_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3'|\mu',\sigma'\end{array}} \quad (\textsc{IfSymb2})$$

$$\begin{array}{c}\phi \vdash \mathsf{if}\ sv_1\ \mathsf{then}\ v_2\ \mathsf{else}\ v_3|\mu,\sigma \to \\ \mathsf{Exp}(\mathsf{if}, sv_1, v_2, v_3)|\mu,\sigma\end{array} \quad (\textsc{IfSymb})$$

Figure 8. Syntax and Reduction Rules for Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects

$$
\begin{array}{rcl}
v & ::= & \ldots \\
 & | & sv \mid \mu \\
 & & \\
sv & ::= & sx \mid \mathsf{Exp}\ v^*
\end{array}
$$

```
:=    =  λ a, b.
           if sym? !a then
             a:=sym({(!pc ∧ pc', b)|(pc',v) ∈ !a} ∪
                    {(¬!pc ∧ pc', v)|(pc',v) ∈ !a})
           else if !pc != true then
             a:=sym({(pc, b), (¬pc, !a)})
           else a := b

if    =  λ c, t, e.
           if sym? c then
              let old_pc = !pc in
                pc := old_pc ∧ c;
                let v1 = t.call in
                pc := old_pc ∧ ¬c;
                let v2 = e.call in
                pc := old_pc;
                Exp(if, c, v1, v2)
           else if c then t.call else e.call

sym   =  λ vals. (SYM_TAG, vals)
sym?  =  λ v. v = (SYM_TAG, vals) or sym? v

pred  =  λ v. if sym? v then Exp(pred, v) else pred v
succ  =  λ v. if sym? v then Exp(succ, v) else succ v
iszero=  λ v. if sym? v then Exp(iszero, v) else iszero v
```

Figure 9. Implementation of Primitives to Achieve Mixed Concrete and Symbolic Execution of Untyped λ-calculus with Side Effects Under Standard Semantics

$$
\begin{array}{rl}
 & t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{if\ sym?}\ sv_1\ \mathsf{then\ ...\ else\ ...}|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{pc} := \phi \wedge sv_1; \ ...|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{let\ v_1} = t_2.\mathsf{call\ in\ ...}|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi \wedge sv_1) \\
\xrightarrow{*}_\theta & \mathsf{let\ v_1} = v_2\ \mathsf{in\ ...}|(\gamma(\mu',\sigma'), \mathsf{pc} \mapsto \phi \wedge sv_1) \\
 & \text{(by inductive hypothesis)} \\
\xrightarrow{*}_\theta & \mathsf{pc} := \phi \wedge \neg sv_1; \ ...|(\gamma(\mu',\sigma'), \mathsf{pc} \mapsto \phi \wedge sv_1) \\
\xrightarrow{*}_\theta & \mathsf{let\ v_2} = t_3.\mathsf{call\ in\ ...}|(\gamma(\mu',\sigma'), \mathsf{pc} \mapsto \phi \wedge \neg sv_1) \\
\xrightarrow{*}_\theta & \mathsf{let\ v_2} = v_3\ \mathsf{in\ ...}|(\gamma(\mu'',\sigma''), \mathsf{pc} \mapsto \phi \wedge \neg sv_1) \\
 & \text{(by inductive hypothesis)} \\
\xrightarrow{*}_\theta & \mathsf{pc} := \phi; \ ...|(\gamma(\mu'',\sigma''), \mathsf{pc} \mapsto \phi \wedge \neg sv_1) \\
\xrightarrow{*}_\theta & \mathsf{Exp}(sv_1,\ v_2,\ v_3)|(\gamma(\mu'',\sigma''), \mathsf{pc} \mapsto \phi)
\end{array}
$$

CASE 2. $t = l{:=}v_2|\mu,\sigma$ *where* $\mu(l) \notin sx$ *and* $\phi = \mathsf{true}$.

By ASSIGN, we have:

$$\mathsf{true} \vdash t|\mu,\sigma \xrightarrow{*} \mathsf{unit}|[l \mapsto v_2]\mu,\sigma$$

By $\rightarrow_\theta$:

$$
\begin{array}{rl}
 & t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & l := v_2|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{unit}|[l \mapsto v_2](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
= & \mathsf{unit}|(\gamma([l \mapsto v_2]\mu,\sigma), \mathsf{pc} \mapsto \phi)
\end{array}
$$

CASE 3. $t = l{:=}v_2|\mu,\sigma$ *where* $\mu(l) \notin sx$ *and* $\phi \neq \mathsf{true}$.

By ASSIGNSYMB1, we have:

$$\phi \vdash t|\mu,\sigma \xrightarrow{*} \mathsf{unit}|[l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})$$

By $\rightarrow_\theta$:

$$
\begin{array}{rl}
 & t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & l := \mathsf{sym}(\{(\phi,\ v_2),\ (\neg\phi,\ !l)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & l := \mathsf{sym}(\{(\phi,\ v_2),\ (\neg\phi,\ \mu(l))\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
 & \text{(because } \mu(l) \notin sx\text{)} \\
\xrightarrow{*}_\theta & l := (\mathsf{SYM\_TAG},\ \{(\phi,\ v_2),\ (\neg\phi,\ \mu(l))\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{unit}|[l \mapsto (\mathsf{SYM\_TAG}, \{(\phi, v_2), (\neg\phi, \mu(l))\})](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
= & \mathsf{unit}|(\gamma([l \mapsto sx_1]\mu, (\sigma, sx_1 \mapsto \{(\phi, v_2), (\neg\phi, \mu(l))\})), \mathsf{pc} \mapsto \phi) \\
 & \text{(by definition of } \gamma\text{)}
\end{array}
$$

CASE 4. $t = l{:=}v_2|\mu,\sigma$ *where* $\mu(l) = sx_1$.

By ASSIGNSYMB2, we have:

$$\phi \vdash t|\mu,\sigma \xrightarrow{*} \mathsf{unit}|\mu,\sigma'$$

$$\text{where } \sigma' = [sx_1 \mapsto \{(\phi \wedge \phi', v_2)|(\phi', v) \in \sigma(sx_1)\} \cup \{(\neg\phi \wedge \phi', v)|(\phi', v) \in \sigma(sx_1)\}]\sigma$$

By $\rightarrow_\theta$:

$$
\begin{array}{rl}
 & t|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & l{:=}\mathsf{sym}(\{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in !l\} \cup \\
 & \quad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in !l\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & l{:=}\mathsf{sym}(\{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\} \cup \\
 & \quad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
 & \text{(since } \mu(l) = sx_1, \gamma(\mu,\sigma)(l) = \sigma(sx_1)\text{)} \\
\xrightarrow{*}_\theta & l{:=}(\mathsf{SYM\_TAG},\ \{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\} \cup \\
 & \quad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\})|(\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
\xrightarrow{*}_\theta & \mathsf{unit}|[l \mapsto (\mathsf{SYM\_TAG}, S)](\gamma(\mu,\sigma), \mathsf{pc} \mapsto \phi) \\
 & \text{where } S = \{(\phi \wedge \mathsf{pc'},\ v_2)|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\} \cup \\
 & \quad \{(\neg\phi \wedge \mathsf{pc'},\ \mathsf{v})|(\mathsf{pc'},\mathsf{v}) \in \sigma(sx_1)\}) \\
= & \mathsf{unit}|(\gamma(\mu,\sigma'), \mathsf{pc} \mapsto \phi) \\
 & \text{(by definition of } \gamma\text{)}
\end{array}
$$

The remaining cases are straightforward. ∎

4. Related Work

Research on symbolic execution has a long history, with the first systems due to King [22] and to Clarke [13], both in 1976. Interest in symbolic execution has continued, and new developments have greatly increased the scalability of symbolic execution engines [6, 17, 21, 27, 30, 31].

Two notable examples of modern symbolic execution systems are the symbolic extension of Java PathFinder [21, 27], which has been used to analyze Java code used by NASA, and CUTE [31], a "concolic" testing tool for C that interleaves invocations of a symbolic and concrete execution.

The recent popularity of dynamic languages has lead to a corresponding interest in symbolic execution for these languages. Saxena et. al [30] perform symbolic execution on Javascript programs, for example, to discover malware; Rozzle [14] is a similar effort that uses symbolic execution along with other techniques to detect malicious Javascript.

Embedding symbolic values directly in the target language is not without precedent. The Jeeves system [36] is designed to enforce security policies, and is implemented as a library in Scala. Jeeves provides symbolic values with which to build policies, but policies are written in a subset of Scala. Köskal et. al [23] also

embed symbolic values in Scala; they use symbolic values in performing constraint programming.

Austin et. al [3] propose *virtual values*, and allow the programmer to provide definitions for primitive operations over these values. Such a mechanism provides the perfect platform on which to build library-based alternative execution models like our approach to symbolic execution.

Rubicon's specification language is based on the RSpec domain-specific language for testing [9], and its philosophy is most heavily influenced by QuickCheck [12], a random testing framework for Haskell. Rubicon's back-end solver is based on Alloy [20], which is itself based on the Kodkod relational model finder [34].

Existing work applying formal methods to web applications has focused on modeling navigation between pages [1, 4, 5, 7, 10, 15, 24, 33, 35]. A smaller but growing body of work (e.g. [2, 19, 32]) focuses on building formal models of the behavior of web applications.

Work whose goals are closest to those of Rubicon include Chlipala's Ur/Web [11], which statically verifies user-defined security properties of web applications, and Chaudhuri and Foster's work [8], which verifies the absence of some particular security vulnerabilities for Rails applications. Nijjar and Bultan [26] translate Rails data models into Alloy to find inconsistencies, but do not check behavior.

5. Conclusion

We have presented a new approach to symbolic execution in which the standard implementation of the target language is used both to propagate symbolic values, and to compute in the normal way with concrete values when available. We implement this approach as a library written in the target language itself; this library comprises an encoding of symbolic values and new symbolic definitions for the primitive operations of the language, and effectively transforms the standard (concrete) implementation of the target language into a symbolic executor.

We used this approach to build Rubicon, a scalable bounded verification system for Ruby on Rails web applications, in fewer than 1000 lines of Ruby code. Despite its small size, Rubicon has been used to find previously unknown bugs in open-source Rails applications.

We might never have considered this approach to symbolic execution if not for the rich extensibility offered by Ruby. Most mainstream languages now offer at least limited facilities for building embedded domain-specific languages, and many, including Ruby, go farther, allowing the programmer to redefine even the language's most primitive operations.

We hope the trend towards greater language extensibility continues. While it is possible to misuse the power that comes with a truly extensible language, surely that power is worth the risk if it enables as powerful a technique as symbolic execution to be implemented in as small a library as Rubicon.

References

[1] L. Alfaro. Model checking the world wide web. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 337–349. Springer-Verlag, 2001.

[2] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with fsms. *Software and Systems Modeling*, 4(3):326–345, 2005.

[3] T. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 921–938. ACM, 2011.

[4] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *In Proceedings of 11th International World Wide Web Conference (WW W2002*. Citeseer, 2002.

[5] B. Bordbar and K. Anastasakis. Mda and analysis of web applications. *Trends in Enterprise Application Architecture*, pages 44–55, 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[7] D. Castelluccia, M. Mongiello, M. Ruta, and R. Totaro. Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, 157(1):61–76, 2006.

[8] A. Chaudhuri and J. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.

[9] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. The rspec book: Behaviour driven development with rspec, cucumber, and friends. *Pragmatic Bookshelf*, 2010.

[10] J. Chen and X. Zhao. Formal models for web navigations with session control and browser cache. *Formal Methods and Software Engineering*, pages 46–60, 2004.

[11] A. Chlipala and L. Impredicative. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, page 1. USENIX Association, 2010.

[12] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 35(9):268–279, 2000. ISSN 0362-1340.

[13] L. Clarke. A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, (3):215–222, 1976.

[14] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*. USENIX Association, 2011.

[15] L. De Alfaro, T. Henzinger, and F. Mang. Mcweb: A model-checking tool for web site debugging. In *Poster presented at WWW*, volume 10. Citeseer, 2001.

[16] J. Field and M. Hicks, editors. *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, 2012. ACM. ISBN 978-1-4503-1083-3.

[17] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[18] D. Hansson. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[19] M. Haydar, A. Petrenko, and H. Sahraoui. Formal verification of web applications modeled by communicating automata. *Formal Techniques for Networked and Distributed Systems–FORTE 2004*, pages 115–132, 2004.

[20] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.

[21] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[22] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In Field and Hicks [16], pages 151–164. ISBN 978-1-4503-1083-3.

[24] D. Licata and S. Krishnamurthi. Verifying interactive web programs. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 164–173. IEEE.

[25] J. P. Near and D. Jackson. Rubicon: Bounded verification of web applications. In *Proceedings of the 20th ACM SIGSOFT Symposium on Foundations of software engineering*, page To appear. ACM, 2012.

[26] J. Nijjar and T. Bultan. Analyzing ruby on rails data models using alloy. *GSWC 2010*, page 39, 2010.

[27] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. *Model Checking Software*, pages 164–181, 2004.

[28] B. Pierce. *Types and programming languages*. The MIT Press, 2002.

[29] R. Rajagopalan. Rubygems virtual_keywords. `http://rubygems.org/gems/virtual_keywords`.

[30] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010.

[31] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005. ISBN 1-59593-014-0.

[32] J. Syriani and N. Mansour. Modeling web systems using sdl. *Computer and Information Sciences-ISCIS 2003*, pages 1019–1026, 2003.

[33] P. Tonella and F. Ricca. Dynamic model extraction and statistical analysis of web applications. 2002.

[34] E. Torlak and D. Jackson. Kodkod: A relational model finder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[35] M. Winckler and P. Palanque. Statewebcharts: A formal description technique dedicated to navigation modelling of web applications. *Interactive Systems. Design, Specification, and Verification*, pages 279–288, 2003.

[36] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In Field and Hicks [16], pages 85–96. ISBN 978-1-4503-1083-3.