Symbolic Simulation of Microprocessor Models using Type Classes in Haskell

Nancy A. Day, Jeffrey R. Lewis, and Byron Cook

Technical Report CSE-99-005
Department of Computer Science
Oregon Graduate Institute

{nday, jlewis, byron}@cse.ogi.edu

June 25, 1999

Abstract

We present a technique for doing symbolic simulation of microprocessor models in the functional programming language Haskell. We use polymorphism and the type class system, a unique feature of Haskell, to write models that work over both concrete and symbolic data. We offer this approach as an alternative to using uninterpreted constants. When the full generality of rewriting is not needed, the performance of symbolic simulation by evaluation is much faster than previously reported symbolic simulation efforts in theorem provers. We illustrate our work with both a simple state-based example and a complex, superscalar, out-of-order, stream-based microprocessor model. This technical report is a companion report to Day, Lewis, and Cook [DLC].

1 Introduction

Symbolic simulation of microprocessor models can be used for validation of microcode [Gre98] and it is a key ingredient to verification techniques such as symbolic trajectory evaluation [SB95] and Burch-Dill style microprocessor verification [BD94, JDB95]. Symbolic simulation executes a model for multiple data values in a single simulation run. For example, a symbolic program that we discuss in this paper takes the input data x and calculates x^4 (or $x*x*x*x$).

Symbolic simulation of microprocessor models written in the Haskell programming language [PH97] is possible without extending the language or its compilers and interpreters. When symbolically simulating a simple microprocessor model, we achieved performance of approximately 58 300 instructions per second. We describe how Haskell's type class system allows a symbolic domain to be substituted for a concrete one without changing the model or explicitly passing the operations on the domain as parameters. Algebraic manipulations of values in the symbolic domain carry out simplifications similar to what is accomplished by rewriting in theorem provers to reduce the size of terms in the output.

The infrastructure required for using symbolic values and maintaining a symbolic state set is reusable for simulation of different models. We believe the approach presented in this paper may be applied in other languages with user-defined data types, polymorphism, and overloading. However, a key requirement is that overloading work over polymorphic types. Few programming languages support this, although a different approach using parameterized modules, as in SML, might also work well. Haskell's elegant integration of overloading with type inference and the clear semantics of the language make it amenable to formal verification.

2 Example

To illustrate our technique, we use the simple, non-pipelined, state-based processor model given in Moore's paper on symbolic simulation [Moo98]. First, we explain the model and demonstrate concrete simulation. Next, we show how using more general types for the data in the model makes it possible to simulate interchangeably concrete and symbolic values. The full source code for this example in Haskell is found in the appendix.

2.1 Model

The opcodes of the simple machine are described using a data type:

```
data Op = MOVE Addr Addr
        | MOVI Addr Data
        | ADD Addr Addr
        | SUBI Addr Data
        | JUMPZ Addr Loc
        | JUMP Loc
        | CALL String
        | RET
```

For now, interpret the type names `Addr` (memory address), `Loc` (location), and `Data` as integers.

The machine's visible state is captured by five values: the program counter, the stack pointer, the data memory (modeled as a list, and indexed by integers), the halt signal, and the program. The program is indexed by a name and location because separate routines are stored in distinct memory. Thus, the program counter and elements of the stack consist of both a name and a location. The program consists of names with associated lists of instructions. The machine's state is captured using the following data type[1]:

```
data MachState = ST ((String,Loc),
                     [(String,Loc)],
                     [Data],
                     Bool,
                     Program)
```

The meaning of each instruction is described by individual functions that take a machine state and return a machine state, such as:

```
add a b (ST ((name,loc),stk, mem,halt,code)) =
   mkState ((name,loc+1), stk,
            put a (mem ‘at‘ a + mem ‘at‘ b) mem, halt, code)

subi a b (ST ((name,loc),stk, mem,halt,code)) =
   mkState ((name,loc+1), stk,
            put a ((mem ‘at‘ a) - b) mem, halt, code)

jumpz a b (ST ((name,loc),stk, mem,halt,code)) =
   if’ ((mem ‘at‘ a) === 0)
   (mkState ((name,b),stk, mem,halt,code))
   (mkState ((name, loc + 1), stk, mem, halt, code))
```

The semantics of the ADD instruction increase the program counter by one and put the result of the "+" operation on the values in memory locations a and b in memory location a. The function at is an indexing function. In Haskell, to use a regular identifier as an infix operator, you surround it with backquotes, as we did above. The SUBI instruction subtracts the immediate value b from the memory location a. The JUMPZ instruction sets the program counter to the value b if memory location a has the value 0. The operator === is defined to be equality over integers and if’ is if-then-else. The function mkState turns a tuple into a state.

The function execute matches opcodes to the semantic functions. For example, execute calls the semantic function for ADD as follows, where s is a state:

```
execute (ADD a b) s = add a b s
```

2.2 Concrete Simulation

We can execute the model on particular concrete programs. One of the example programs given in Moore's paper multiplies the value in mem[0] by mem[1] using repeated addition, leaving the result in mem[2], and clearing mem[0]:

[1] In Haskell, list types are represented using square brackets ("[...]").

```
prog = [ MOVI 2 0,        -- 0, mem[2] <- 0
         JUMPZ 0 5,       -- 1, if mem[0]=0 goto 5
         ADD 2 1,         -- 2, mem[2] <- mem[1] + mem[2]
         SUBI 0 1,        -- 3, mem[0] <- mem[0] -1
         JUMP 1,          -- 4, goto 1
         RET ]            -- 5, return to caller
```

Comments (prefixed by `--`) on the right describe the meaning of each instruction. Beginning with memory containing the values `[7,11,3,4,5]` (i.e., `mem[0]` containing 7 and `mem[1]` containing 11), and executing the machine for 31 cycles, results in the following memory state: `[0,11,77,4,5]`. Memory location 2 contains the result of multiplying 7 by 11.

2.3 Overloading: Type Classes

We now use the type class system of Haskell to make all the operations that manipulate data be overloaded on both concrete and symbolic data.

In the previous concrete simulation, the type of the function `subi` is[2]:

```
subi :: Addr -> Data -> MachState -> MachState
```

To simulate symbolic values, we will make it so that the type `Data` can be interpreted at other types than `Int`. We cannot allow `Data` to be any type (i.e., make `subi` polymorphic) because numeric operations are not defined for all types. Alternatively, we could parameterize `subi` by numeric and other operations that are type-specific to the type of data in memory (i.e., symbolic or concrete). This is the approach of Joyce-style representation variables [Joy90], where all the semantic functions are parameterized by what could become a long list of any operations that are type-specific for any opcode.

Our solution is to take advantage of the overloading of operators provided by type classes. A type class groups a set of operations by the type they operate over. The typechecker is able to determine which instance of the operation is being invoked based on the type of its arguments.

The existing Haskell type class `Num` has almost all the operators that we require for data values for this example. In Haskell, a type class definition declares the name of the class and the operations on members of the class. The `Num` class has the following definition:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  fromInt :: Int -> a
```

Following the first line in this class definition are operators defined on types within this class. Parentheses indicate that the operation is infix. The parameter

[2] In Haskell, a type expression is preceded by a "::" .

a after the name of the class is a placeholder for types belonging in this class. The type signatures of the operations are described in terms of this type. The simple machine only requires the use of "+" and "-". The function fromInt turns integers into values of type a. This capability is very useful when moving to the symbolic domain because it means existing uses of constant integers do not have to be converted by hand into their representation in the symbolic domain – fromInt is automatically applied to them.

In Haskell, the type Int is declared to be an instance of the Num class.

For the JUMPZ opcode, the equality operation on data values is also needed. Therefore, we create a new class called Word that inherits all the operations of Num and includes the operation === for semantic equality.

```
class Num a => Word a where
  (===) :: a -> a -> a
```

The use of the operator => in Haskell indicates that the type a must be a member of the type class Num and therefore the types in Word inherit all of Num's operations. The type Int is an instance of the type Word where the equals operator returns true (1) if the two integer operands are equal, and false (0) otherwise. Boolean values are treated as integers.

The type of values in memory now must be elements of the type class Word. The types MachState and Program are parameterized by the type of the memory elements, as in:

```
data MachState a = ST ((String,Loc),
                       [(String,Loc)],
                       [a],
                       Bool,
                       Program a)
```

Opcodes are also adjusted to take immediate values of types in the Word class rather than just integers. For example, the type of the subi instruction becomes:

```
subi :: Word a => Addr -> a -> MachState a -> MachState a
```

The definition of subi does not change.

Concrete simulation of prog results in the same state.

2.4 Symbolic Simulation of Data Flow

Once the model has been set up to accept memory values of types within the Word class rather than just integers, we can consider an appropriate symbolic domain. Our symbolic domain must include representations of all operations that the model performs on integers. The values of this domain represent syntactic versions of the expressions performed by the machine. An appropriate symbolic domain for this example includes representations for constants (Const), symbols (Var), and the results of addition and subtraction operations. Using a recursive data type, we describe the values in the symbolic domain as:

```
instance Num Symbo where
  Const x + Const y = Const (x + y)
  Const 0 + y       = y
  x + Const 0       = x
  x + y             = x ‘Plus‘ y

  Const x - Const y = Const (x - y)
  x - Const 0       = x
  x - y             = x ‘Minus‘ y

  Const x * Const y = Const (x * y)
  Const 0 * y       = Const 0
  x * Const 0       = Const 0
  Const 1 * y       = y
  x * Const 1       = x
  x * y             = x ‘Times‘ y

  fromInt           = Const

instance Word Symbo where
  (Const x) === (Const y)  =  if (x == y) then (Const 1) else (Const 0)
```

Figure 1: Instance declarations for “Symbo”

```
data Symbo =
      Const Int
    | Var String
    | Plus Symbo Symbo
    | Minus Symbo Symbo
    | Times Symbo Symbo
```

Plus and Minus will be used to represent the results of addition and subtraction operations on numbers. We include a representation of multiplication (Times) because using algebraic laws we can simplify expressions involving addition and subtraction to expressions involving multiplication (Section 2.5).

Next, we create an instance of the Num and Word type classes providing witnesses showing how the required operations of Num and Word are implemented for Symbo. Fig. 1 shows the instance declarations for Symbo that include function definitions (using pattern-matching) for these operations. The last case in the pattern-matching is the default case. We assume for the moment that the operands to the equality operation will only be concrete values therefore we define === in terms of Haskell's syntactic equality operator ==.

After providing these instance declarations, all that is necessary to simulate symbolically the program prog is to provide symbolic inputs. To calculate $7 * j$, we begin with memory having the values,

```
[7,Var "j",Var "x",Var "y",Var "z"]
```

The result of the program after 31 steps is[3]:

```
[0,j,j + j + j + j + j + j + j,y,z]
```

This result shows that the sequence of opcodes in the program performs repeated addition resulting in seven additions of 7 being left in memory position 2.

In this example, we only made one input symbolic. If we had made all of memory symbolic, we would not have been able to execute the program because the value in memory location 1 is used to determine if a branch is taken. Because we have not yet defined equality on symbolic values, checking whether a value like Var "i" is 0 would cause a run-time error. We extend our example with symbolic branching in Section 2.6.

Symbolic values in memory are used interchangeably with concrete values in memory (e.g., 7) and in the immediate values within the programs (e.g., 0 in MOVI 2 0). The function fromInt in the Num class turns concrete values into symbolic values making this interchangeability possible. Programs running on concrete values and producing concrete output can still be run on the model with the more general types.

2.5 Algebraic Simplifications

The symbolic domain must have the same behavior as the concrete domain. For the case of numbers, there are algebraic laws that hold for the concrete domain that can be used to simplify the output of symbolic simulation. For example, Var x + Var x is equivalent to Const 2 * Var x. These rules can be implemented for the symbolic domain by augmenting the instance declaration for Symbo with cases that describe the algebraic rules. Two algebraic rules useful for the multiplication program are:

```
 Var x + Var y = if (x == y) then Const 2 * Var x
                             else Var x ‘Plus‘ Var y

 ((Const x) ‘Times‘ (Var y)) + (Var z) =
       if (y == z) then (Const (x+1)) *  (Var y)
                   else (Const x ‘Times‘ Var y) ‘Plus‘ Var z
```

Using these algebraic simplifications, the result of the multiplication program calculating $7 * j$ is [0,j,7 * j,y,z].

These algebraic simplification rules perform a similar task to rewriting in a theorem prover.

[3] This output is pretty printed to remove the “Var” and “Const” prefixes.

2.6 Symbolic Simulation of Control Flow

When control values in a program are symbolic, the output of symbolic simulation captures the multiple execution paths that the program could have followed. Memory location 1 is a control value in the program prog, because its value is used to determine whether to take a branch or not. To deal with symbolic simulation of control values, we have to extend our idea of a state to include branches representing multiple execution paths. We build this infrastructure on top of the model.

The branching structure will have states at its leaves. The following is a data type for capturing trees of states:

```
data State f a =
       CondS a (State f a) (State f a) |
       Term (f a)
```

The type variable a describes the type of the expression that is used to decide which branch to follow. In our symbolic simulation, this type variable is instantiated to Symbo. The type variable f describes the form of the leaf states. For the simple machine, this will be the type MachState. Because MachState is parameterized by the type of data in its memory, we use the type expression f a, providing the parameter Symbo to MachState. The data constructor CondS represents multiple execution paths that are conditional on the first argument to CondS.

To take a step in this symbolic machine, each leaf state must take a step. This may result in new branches in the tree. The function step_state is defined over leaf states and invokes the function execute described in Section 2.1. Using step_state, we can define a function to take steps over our symbolic state:

```
step (Term s) = step_state s
step (CondS a b c) = CondS a (step b) (step c)
```

Next, we need to extend our symbolic domain to include the result of checking for semantic equality over symbolic values. We add one new symbolic value:

```
data Symbo =
    ...
    | Equals Symbo Symbo
```

The definition of equality in the instantiation of Symbo as a member of the Word type class is now extended to:

```
instance Word Symbo where
    (Const x) === (Const y)  =  if (x==y) then (Const 1) (Const 0)
    a === b                  =  Equals a b
```

Finally, we need to create branches in the state data structure when conditional jumps are encountered in the program and symbolic data determines which branch to take. The operator if' used in the semantics of JUMPZ must be able to sometimes return a terminal state and sometimes return a branch state. We use a multi-parameter type class to capture the behavior of `if'`. A multi-parameter type class allows you to constrain multiple types in a class instantiation. In the case of `if'`, we parameterize the type of the first argument (the deciding value), separately from the type of the other arguments. The result of the function has the same type as the second and third arguments.

```
class Conditional a b where
    if' :: a -> b -> b -> b
```

For working with concrete states, we need an instantiation that uses the regular if-then-else for concrete values. Since we are treating Booleans as numbers, it checks if its first argument is 1.

```
instance Conditional Int (State f Int) where
        if' a b c = if (a==1) then b else c
```

When the first argument is symbolic, we have a different definition of `if'` that returns a branched state if the argument is symbolic.

```
instance Conditional Symbo (State f Symbo) where
        if' (Const 1) b c = b
        if' (Const 0) b c = c
        if' a b c         = CondS a b c
```

Now without having changed our model, we have the necessary ingredients to simulate symbolic control values[4]. For example, if we run the program `prog` for 20 steps, with all symbolic values in memory, calculating $i * j$ produces the output found in Fig. 2. In this output, we have included the value of the halt flag for each state. If i is 0, then the result in memory location 2 is 0 and the program has stopped. If i is 1, the result is j and the program has stopped. The last line of the figure is for the case where $i > 4$, so the result will be at least $5 * j$.

3 Symbolic Simulation of a Superscalar, Out-of-order Microarchitecture

We are modifying an existing Hawk model for a Pentium II-like microarchitecture [CLM98] to use the type class facilities of Haskell for symbolic simulation. This design is a superscalar, out-of-order, with exceptions, pipelined architecture. We are now able to simulate symbolic data flow for programs running on the model.

Hawk is a Haskell-based hardware description language for expressing microarchitecture designs [CLM98, MCL98]. The value of Haskell's higher-order

[4] The types of the semantic functions change to return a symbolic state but these type changes can be inferred by the typechecker.

```
CondS (i == 0)
  ([i,j,0,y,z],True)
  CondS ((i - 1) == 0)
    ([i - 1,j,j,y,z],True)
    CondS ((i - 2) == 0)
      ([i - 2,j,2 * j,y,z],True)
      CondS ((i - 3) == 0)
        ([i - 3,j,3 * j,y,z],True)
        CondS ((i - 4) == 0)
          ([i - 4,j,4 * j,y,z],True)
          ([i - 5,j,5 * j,y,z],False)
```

Figure 2: Output of prog after 20 steps with inputs "i" and "j"

functions and polymorphism are illustrated in this Hawk model although we do not have space to describe them in this paper.

Hawk models usually process transactions. A transaction captures the state of an instruction as it progresses through the pipeline. A transaction contains the address of the instruction, its opcode, and the addresses and values of its operands. The transaction may also contain a speculative PC. As the transaction moves through the pipeline, values for input operands and result operands get filled in. The speculative PC is compared to the calculated result of a branch instruction to determine if the pipeline needs to be flushed.

The essential change necessary to use type classes in this design was to modify the values in registers and memory to be of a type belonging to the type class `Num` rather than only integers. This modification also affects the type of addresses because calculations unite the address and value space. Various Hawk library devices that manipulate transactions were changed to the more general type.

The `Symbo` data type was used to execute a symbolic program calculating x^4 on this design. Fig. 3 shows our representation of the symbolic DLX [HP96] program. The comments beside each instruction indicate the address where the instruction is placed in memory. The output of simulating a Hawk model is a stream of transactions describing the instructions that have been executed. Fig. 4 shows the output of the symbolic x^4 program for 48 cycles. The number on the left is the cycle that the transaction leaves the pipeline. Because this processor is superscalar, multiple instructions may leave the pipeline in one cycle. The program counter is after the cycle number on an output line. The values of the registers used in computation are given in parentheses. If the instruction is a branch, a speculative program counter is included in the transaction.

We are currently extending the Hawk library to handle symbolic control paths as well. The key to making this work is to have trees of transactions flowing along the wires instead of just simple transactions. This is similar to

```
prog_x_4 =
 [ImmIns (ALUImm (Add Signed)) R3 R0 (Var "x")),-- 64: R3 <- R0 + x
  ImmIns (ALUImm (Add Signed)) R4 R0 4),        -- 65: R4 <- R0 + 4
  ImmIns (ALUImm (Add Signed)) R6 R0 1,         -- 66: R6 <- R0 + 1
  ImmIns (ALUImm (Add Signed)) R5 R0 0,         -- 67: R5 <- R0 + 0
                                                -- loop begins here
  RegReg ALU (S GreaterEqual) R1 R5 R4,         -- 68: R4 <- R1 >= R5
  ImmIns BNEZ R0 R1 32,                         -- 69: if (R1==0) then
                                                     goto (70+32/4=78)
  Nop,                                          -- 70: No_op
  RegReg ALU Input1 F2 R6 R0,                   -- 71: F2 <- R6
  RegReg ALU Input1 F3 R3 R0,                   -- 72: F3 <- R3
  RegReg ALU (Mult Signed) F2 F2 F3,            -- 73: F2 <- F2 * F3
  RegReg ALU Input1 R6 F2 R0,                   -- 74: R6 <- F2
  ImmIns (ALUImm (Add Signed)) R5 R5 1,         -- 75: R5 <- R5 + 1
  Jmp J ((-36)),                                -- 76: goto (77-36/4=68)
                                                -- end of loop
  Nop,                                          -- 77: No_op
  RegReg ALU (Add Signed) R1 R0 R6,             -- 78: R1 <- R0 + R6
 ]
```

Figure 3: Symbolic DLX program for x^4

how the state in the earlier example became trees of states. However, a Hawk model is stream-based and therefore, does not have explicit access to its state like the earlier example does. Instead of simply having a top-level branching of state, the branching of state must be threaded through the entire model, just as transactions are. This means that most components will need to understand how to handle trees of transactions. We are exploring how to best use a transaction type class to define easily a new instance of transactions that are trees.

Once these modifications to the Hawk library have been made, all future models will be able to simulate both concrete and symbolic programs. The symbolic domain presented in this paper is sufficient for many microarchitectures.

4 Performance

In this section, we consider the performance of our "symbolic simulator". We used the Glasgow Haskell Compiler Version 4.02 [Ghc] for running our tests. Moore provided timing numbers for doing symbolic simulation of the simple machine in the theorem prover ACL2 on a 200 MHz Sun Ultra 2 with 512 MB [Moo98]. Unfortunately, we did not have an equivalent platform available and ran our test cases on a 450 MHz Intel Pentium II with 512 MB memory. Based on SPEC CPU95 integer benchmarks, our platform is roughly two and half times faster than Moore's [SPE].

For concrete simulation, the multiplication program calculating $10\,000 * 1000$ for 40 007 cycles took 0.53 seconds with ACL2 at best and 0.54 seconds for us. Here we are comparing Lisp execution to Haskell execution. On a larger concrete test case taking 400 000 cycles for $100\,000 * 1000$, we achieved approximately 62 200 instructions per second.

In ACL2, the multiplication program with symbolic data flow calculating $1000 * j$ for 4005 cycles took at best 17 seconds with hints and at worst 55 seconds (72 and 235 instructions per second respectively). Running the same symbolic program took 0.04 seconds for us. When running a much larger test case of $100\,000 * j$ for 400 000 instructions (no branches) we achieved 58 300 instructions per second.

The multiplication program with symbolic control flow calculating $i * j$ for 2000 cycles took 1.55 seconds, which is approximately 1290 instructions per second. With branching symbolic programs, printing time is significant.

ACL2 must use its rewrite engine for symbolic simulation, whereas our approach involves executing a functional program. Therefore, we do not suffer a performance penalty for symbolic simulation. Rewriting requires searching a database of rewrite rules and potentially following unused simplifications [Moo98].

```
1:
2:
3:
4:   256:    R3(x) <- R0(0) + x
     260:    R4(4) <- R0(0) + 4
5:   264:    R6(1) <- R0(0) + 1
     268:    R5(0) <- R0(0) + 0
6:   272:    R1(0) <- R5(0) >= R4(4)
7:   276:    PC(280) <- if R1(0) then PC(280) + 32 else PC(280)
                                                   (SpecPC(256))
8:
...
16:
17:  292:    F2(x) <- F2(1) * F3(x)
18:  296:    R6(x) <- F2(x)
     300:    R5(1) <- R5(0) + 1
     304:    PC(272) <- PC(308) + -36              (SpecPC(256))
19:
...
28:
29:  292:    F2(x * x) <- F2(x) * F3(x)
30:  296:    R6(x * x) <- F2(x * x)
     300:    R5(2) <- R5(1) + 1
     304:    PC(272) <- PC(308) + -36              (SpecPC(272))
     272:    R1(0) <- R5(2) >= R4(4)
     276:    PC(280) <- if R1(0) then PC(280) + 32 else PC(280)
...
42:
43:  292:    F2(x * x * x * x) <- F2(x * x * x) * F3(x)
44:  296:    R6(x * x * x * x) <- F2(x * x * x * x)
     300:    R5(4) <- R5(3) + 1
     304:    PC(272) <- PC(308) + -36              (SpecPC(272))
     272:    R1(1) <- R5(4) >= R4(4)
     276:    PC(312) <- if R1(1) then PC(280) + 32 else PC(280)
                                                   (SpecPC(280))
45:
46:
47:
48:  312:    R1(x * x * x * x) <- R0(0) + R6(x * x * x * x)
```

Figure 4: Stream of transactions resulting from execution of x^4 program

5 Related Work

The approach described in this paper is closely related to work on Lava [BCSS98], another Haskell-based hardware description language. Lava has explored using Haskell features, such as monads, to provide alternative interpretations of circuit descriptions for simulation, verification, and generation of code from the same model. A predominant use of a symbolic circuit interpretation in Lava is to produce output for theorem provers. Consequently, their symbolic simulation assigns labels to all subterms and produces a sequence of assertions relating symbolic inputs to outputs. Our emphasis has been more on building symbolic simulation on top of the simulation provided by the execution of a model as a functional program. In our descriptions of microprocessors, we rely on the standard meaning of function application to connect components of circuits. We use type classes extensively to choose between a symbolic interpretation or a non-symbolic interpretation of an operation. Both interpretations can be used within the same simulation run. To achieve this flexibility, we build the branching structure into the symbolic domain and use type classes to capture the symbolic operations. The branching structure is threaded through the model. This threading relies on multi-parameter type classes – a recent extension to Haskell.

Symbols in Lisp can be used for symbolic simulation. For example, to generate expressions for input to the Stanford Validity Checker [JDB95], a simple HDL based on Common Lisp is used [BD94]. In this paper, we show how this approach can be done in a strongly-typed, higher-order, functional programming language.

Symbolic simulation can be carried out with uninterpreted constants using rewriting techniques in a theorem prover (e.g., [Joy89, Win90, Moo98, Gre98]) or using more specialized techniques such as symbolic functional evaluation [DJ]. In this form of symbolic simulation, the model is executed over constants of unknown value but the same type as a concrete value. It does not require any changes to the model. However, uninterpreted constants are an element of logic and their use requires that the model be expressed in a logic. Simulation of a logical specification requires special-purpose infrastructure such as rewriting or a means of partial evaluation. Constructors in our symbolic domain play the same role as uninterpreted constants using a general-purpose programming language, when the full generality of rewriting is not required.

Type classes provide the infrastructure needed to support the way uninterpreted constants have been used in logical models of microprocessors. Taking advantage of polymorphism in higher-order logic, Joyce first used "representation variables" to bundle operations on data [Joy90]. These operations parameterize both a reference machine and a model of the implementation. The verification effort is valid for any instantiation of these operations. Having an object-oriented flavor, a type class packages the functions of a representation variable in one location. It is not necessary to parameterize all components of the model by type-specific operations. The type classes keep track of the higher-order function parameters. We provide instantiations of the operations of the type class for both concrete and symbolic simulation.

Graph structures such as BDDs and MDGs represent symbolic formulae. Binary decision diagrams (BDDs) [Bry86] are a canonical form for propositional logic. Multiway decision diagrams (MDGs) [CZS$^+$94] are a canonical representation of formulae in many sorted, first-order logic (including uninterpreted functions). In both cases, by iterating a next state relation, these representations can be used to carry out symbolic simulation. BDDs and MDGs are used in decision procedures because of their canonical form. Our form of symbolic simulation for higher-order expressions only calculates terms and does not produce a canonical form. We have not yet characterized the "decidability" of verification efforts involving the symbolic terms we produce. Also our terms are currently not reduced for sharing of subterms and therefore the representation can become quite large. In more recent work, we have investigated linking symbolic simulation in Haskell directly with decision procedures for verification to take advantage of the reduced size of representations in these packages [DLL].

6 Limitations

Our approach is limited to models expressed as functions, although they may be either state-based as in Moore's simple example or stream-based as in the Hawk Pentium II-like model.

Compared to carrying out symbolic simulation in a logic, in our approach it is necessary to introduce a term structure for the symbolic domain. Our symbols differ from uninterpreted constants in logic in that a programming language has a built-in assumption that elements of user-defined types are distinct. Creating the symbolic term structure requires care because the symbolic domain must have the same properties of the concrete domain. Therefore, the usual equality operation is only defined for the symbolic domain in special cases such as `Var x = Var x`. In this paper, we do not address the issues of how one ensures the symbolic domain has the same properties as the concrete domain.

Our symbolic simulation cannot determine when multiple symbolic decision points conflict and therefore prune impossible execution paths.

Finally, type classes can make fixing type errors a more difficult process. For example, type errors are often masked as missing class instantiations.

7 Conclusion

The most important conclusion of this work is that facilities can be found within some existing programming languages to carry out symbolic simulation of microprocessor models. Using a programming language means symbolic simulation is accomplished by simply running a program. The speed of our method compares well with using rewriting techniques to carry out symbolic simulation. The output of symbolic simulation produced by a model written in a programming language or executable hardware description language can be used as input to verification tools.

Type classes in Haskell make it possible to simulate interchangeably concrete and symbolic values without changing the model. Type classes provide a way to exchange domains of values without requiring explicit parameterization. The class definition specifies the operations on both the symbolic and concrete domains. Algebraic manipulations of values in the symbolic domain reduce the size of the symbolic terms in the output.

The symbolic infrastructure is likely to be reusable for future microprocessor models. Thus, the initial investment in setting up the type classes can be amortized over the ability to simulate symbolically many models.

We intend to continue this work by considering how this form of symbolic simulation can be used in verification techniques. For example, symbolic trajectory evaluation (STE) [SB95] is currently being applied at the bit-level using BDDs as a symbolic representation. To apply STE at a more abstract level a means of symbolic simulation of abstract values, such as the one we have presented, is needed. We intend to investigate the use of STE for microarchitecture verification leveraging off of this work on symbolic simulation.

8 Acknowledgments

For their contributions to this research, we thank Mark Aagaard of Intel; Dick Kieburtz, John Launchbury, and John Matthews of OGI; and Tim Leonard, and Abdel Mokkedem of Compaq. The authors are supported by Intel, U.S. Air Force Material Command (F19628-93-C-0069), NSF (EIA-98005542) and the Natural Science and Engineering Research Council of Canada (NSERC).

References

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–79. Springer-Verlag, 1994.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CLM98] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, 1998.

[CZS+94] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. Technical Report RC19676, IBM, 1994. Also *Formal Methods in Systems Design*, 10(1), pages 7-46, 1997.

[DJ] N. A. Day and J. J. Joyce. Symbolic functional evaluation. To appear in TPHOLs'99.

[DLC] N. A. Day, J. R. Lewis, and B. Cook. Symbolic simulation of microprocessor models using type classes in Haskell. Accepted to CHARME'99 poster session.

[DLL] N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. Submitted for publication.

[Ghc] Glasgow Haskell compiler.
http://research.microsoft.com/users/t-simonm/ghc/.

[Gre98] D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD*, volume 1522 of *LNCS*, pages 321–333. Springer, 1998.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, 1996.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD*, 1995.

[Joy89] J. Joyce. *Multi-Level Verification of Microprocessor Based Systems*. PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.

[Joy90] J. J. Joyce. Generic specification of digital hardware. In *Designing Correct Circuits*, pages 68–91. Springer-Verlag, 1990.

[MCL98] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.

[Moo98] J. Moore. Symbolic simulation: An ACL2 approach. In *FMCAD*, volume 1522 of *LNCS*, pages 334–350. Springer, 1998.

[PH97] J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell*. Yale University, Department of Computer Science, RR-1106, 1997.

[SB95] C.-J. H. Seger and R. E. Bryant. Formal verification of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, March 1995.

[SPE] SPEC CPU95 results.
http://www.specbench.org/osg/cpu95/results/cpu95.html.

[Win90] P. J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, 1990.

A Haskell Code

A.1 Infrastructure.hs

```
module Infrastructure where

{-
  The class Num, which already exists in Haskell, has most
  of the operation that we need.  The one extra operation that we
  need is equality that takes two objects and returns an object of
  the same type (not Boolean).  This is so we can represent the
  result of equality symbolically.
-}

class Num a => Word a where
    (===) :: a -> a -> a

{-
  The following multi-parameter type class captures conditional
  operations.  It is multi-parameter because the type of the result
  depends on both the type of the first argument (the test condition)
  and the result type of the function.
-}

class Conditional a b where
    if' :: a -> b -> b -> b

{-
  An instance of the Num class for the type Int
  is built-in to Haskell.  We just have to provide an instance
  of the operations in the Word class.  Here equality
  returns 1 (true) or 0 (false).
-}

instance Word Int where
    a === b = if (a == b) then 1 else 0


{-
  Our symbolic domain contains numbers (Const), symbols
  (Var) and the syntactic representations of the operations
  that can be performed on numbers.
-}
data Symbo  =
      Const Int
    | Var String
    | Plus Symbo Symbo
    | Minus Symbo Symbo
    | Times Symbo Symbo
    | Equals Symbo Symbo

{-
  Unfortunately, Haskell’s built-in type class structure requires
  Num to also be part of the Eq class where normal
  equality is defined.  We never want to use normal equality over
  symbols because that would be comparing syntactic equality not
  semantic equality.  So we make Haskell happy by defining this
  operation to be an error if it’s ever called.
-}
instance Eq (Symbo) where
    a == b = error "shouldn’t be calling this: syntactic equality!"

{-
  As with the symbolic state, we have to show how to print elements
  of our symbolic domain.
-}

instance Show Symbo where
    showsPrec p (Const a) = showsPrec p a
    showsPrec p (Var s) = showString s
    showsPrec p (Plus x y) =
        showParen (p > 6) (showsPrec 6 x .
                           showString " + " .
                           showsPrec 6 y)
    showsPrec p (Minus x y) =
        showParen (p > 6) (showsPrec 6 x .
                           showString " - " .
                           showsPrec 6 y)
    showsPrec p (Times x y) =
       showParen (p > 7) (showsPrec 7 x .
                          showString " * " .
                          showsPrec 7 y)

    showsPrec p (Equals x y) =
       showParen (p > 7) (showsPrec 7 x .
                          showString " == " .
                          showsPrec 7 y)
    showsPrec p _ = error "show not defined for something in Symbo"

{-
  Next we define the meaning of the Num class operators
  on our symbolic domain.  The first four implement algebraic
  simplifications.
-}

instance Num Symbo where

    Var x + Var y = if (x == y)
                    then (Const 2) * (Var x)
                    else Var x ‘Plus‘ Var y
    ((Const x) ‘Times‘ (Var y)) + (Var z) =
          if (y == z)
          then (Const (x+1)) *  (Var y)
          else (Const x ‘Times‘ Var y) ‘Plus‘ Var z
    (Var x ‘Minus‘  Const y) + Const j = Var x ‘Minus‘ Const (y - j)
    (Var x ‘Plus‘  Const y) + Const j = Var x ‘Plus‘ Const (y + j)
    Const x + Const y = Const (x + y)
    Const 0 + y = y
    x + Const 0 = x
    x + y = x ‘Plus‘ y

    (Var x ‘Minus‘ Const y) - Const j = Var x ‘Minus‘ Const (y + j)
    (Var x ‘Plus‘  Const y) - Const j = Var x ‘Plus‘ Const (y - j)
    Const x - Const y = Const (x - y)
    x - Const 0 = x
    x - y = x ‘Minus‘ y

    Const x * Const y = Const (x * y)
    (Const 0) * y = Const 0
    x * (Const 0) = Const 0
    Const 1 * y = y
    x * Const 1 = x
    x * y = x ‘Times‘ y

    fromInt = Const
    fromInteger = Const . fromInteger

{-
  Finally, we need to show how Symbo implements the operations
  of the Word class, namely equality.
-}

instance Word Symbo where
    (Const x) === (Const y) =
           if (x == y) then (Const 1) else (Const 0)
    a === b = Equals a b

{-
```

```
  A symbolic state captures a tree of possible execution paths
  that the machine could take.
-}

data State f a =
       CondS a (State f a) (State f a) |
       Term (f a)

{-
  Taking a step in any symbolic state means taking a step at the
  leaves of each branch.  A step at the leaf of a branch is expected
  to be defined individually for particular machines in the function
  step_state.
-}

step (Term s) = step_state s
step (CondS a b c) = CondS a (step b) (step c)

{-
  The following is simple pretty printing functions for symbolic
  (tree-like) states.  It expects the function printSt to
  be defined for particular machines.  This pretty printing doesn't
  worry about line wrap, etc.
-}

indent :: Int -> [Char]
indent i = if (i==0) then "" else (" " ++  indent (i-1))

printBranchingState i (Term a) =
   showString (indent i) .
   showString (printSt a)
printBranchingState i (CondS a b c) =
   showString (indent i) .
   showString "CondS " .
   showsPrec 9 a .
   showString "\n" .
   printBranchingState (i+2) b .
   showString "\n" .
   printBranchingState (i+2) c

{-
  Next, we say that symbolic states can be printed (i.e., they are
  an instance of the Show class using the above function
  that produces a string.
  We also have to know that the type a in the Word class and
  in the StateComponents class used in the symbolic state
  can be printed.
-}

instance (Word a,StateComponents f a, Show a) => Show (State f a) where
        showsPrec p s = printBranchingState 0 s

{-
  The following function puts the string resulting from show
  on the screen using the IO monad.
-}
printState :: (Word a, StateComponents f a, Show a) => State f a -> IO()
printState s = putStr ((show s)++"\n")

{-
  The following function is used by a particular machine to produce
  a leaf state from its state elements.
-}
state s  = Term s

instance Conditional Int (State f Int) where
        if’ a b c = if (a==1) then b else c

{-
  Operations such as jumpz return different next states
  depending on the current state of the machine.  When memory can
  be symbolic, we may not be able to determine which branch is
  followed so we have to create a branching state.  The following
  instance of the Conditional allows us to do this.
  If its argument is concrete, i.e., evaluates to true (1) or
  false (0), then we proceed normally.  If not, then we return
  a branched state.
-}
instance Conditional Symbo (State f Symbo) where
        if’ (Const 1) b c = b
        if’ (Const 0) b c = c
        if’ a b c = CondS a b c

{-
  StateComponents is the class that captures the operations
  the symbolic state needs to rely on about the machine state.
  The values in memory in a state may have a symbolic or non-symbolic
  type.  This type is captured in the type parameter a.  Therefore
  a StateComponent is a constructor f that depends on
  the type a.
-}
class Word a => StateComponents f a where
```

```
        step_state :: (f a) -> State f a
        printSt ::  (f a) -> String
```

A.2 Model.hs

```
{-
  This simple example is based on the model found in
  J Strother Moore’s paper FMCAD’98 paper entitled ‘‘Symbolic Simulation:
  An ACL2 Approach’’.
  This is a state-based specification of a non-pipelined machine.

  We begin by giving the module a name and importing the definitions
  found in the Infrastructure module.
-}

module Model where

import Infrastructure

{-
  Programs running on this machine consist of a name, which is
  a string and a list of instructions; the name is used in function
  calls and the return pc is (name, loc).  The stack
  stores a list of (name, loc) for RET instructions.

  The machine state contains the program counter, the stack, memory
  as a list, a halt flag, and programs.
-}

data MachState a = ST
      ((String, Loc),    -- program counter
       [(String,Loc)],   -- stack
       [a],              -- memory as a list
       Bool,             -- halt flag
       Program a)        -- program

mkState s = state (ST s)

{-
  A program consists of a label and a list of opcodes.
-}

type Program a = [(String,[Op a])]

{-
  Next, we define a data type to represent the opcodes.
  The only possible symbolic values are immediate operands so the
  opcode type is parameterized by this type.
-}

type Reg = Int
type Loc = Int

data Op a =
          MOVE Reg Reg
        | MOVI Reg a
        | ADD Reg Reg
        | SUBI Reg a
        | JUMPZ Reg Loc
        | JUMP Loc
        | CALL String
        | RET
        deriving (Show)

{-
  We have to show how MachState is an element of the
  type class StateComponents defined in Infrastructure.
  We need to show that as a leaf node of a symbolic state,
  it can be printed, and we can execute the machine for one step.
-}

instance (Conditional a (State MachState a), Word a) =>
     StateComponents MachState a where
        printSt (ST ((name,loc),stk,mem,halt,program)) =
            show ((name,loc),stk,mem,halt)
        step_state (ST s@(pc, stk, mem, halt, code))  =
            if halt
                then (mkState s)
                else (execute (current_instruction pc code) (ST s))

{-
  The current instruction is found by looking in the program memory.
-}

current_instruction :: ([Char],Int) -> Program a -> Op a
current_instruction (name,loc)  ((x,code):_) | (name == x) =
        code ‘at‘ loc
current_instruction pc (_:b) =
        current_instruction pc b
current_instruction pc [] = error "code not found"
```

```
{-
  Memory is modeled as a list, so the memory operation put
  changes the value at a position in the list.
-}

put 0 value (a:b) = (value:b)
put x value (a:b) = (a: put (x-1) value b)

(at) = (!!)

{-
  Execute chooses which instruction to execute based on
  the opcode.  It also takes a statecomponent (the leaf of a symbolic
  state) as an argument and returns a state.
-}

execute (MOVE a b) s = move a b s
execute (MOVI a b) s = movi a b s
execute (ADD a b) s = add a b s
execute (SUBI a b) s = subi a b s
execute (JUMPZ a b) s = jumpz a b s
execute (JUMP a) s = jump a s
execute (CALL a) s = call a s
execute (RET) s = ret s

move a b (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((name,loc+1), stk, put a (mem ‘at‘ b) mem, halt, code)

movi a b (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((name,loc+1), stk, put a b mem, halt, code)

add a b (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((name,loc+1), stk,
              put a (mem ‘at‘ a + mem ‘at‘ b) mem, halt, code)

subi a b (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((name,loc+1), stk, put a (mem ‘at‘ a - b) mem, halt, code)

jumpz a b (ST ((name,loc),stk, mem,halt,code)) =
        if’ ((mem ‘at‘ a)===0)
            (mkState ((name,b),stk, mem,halt,code))
            (mkState ((name, loc+1), stk, mem, halt, code))

jump a (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((name,a), stk, mem, halt, code)
```

```
call a (ST ((name,loc),stk, mem,halt,code)) =
        mkState ((a,0), ((name,loc+1):stk), mem, halt, code)

ret (ST ((name,loc),(stk1:stkrest), mem,halt,code)) =
        mkState (stk1, stkrest, mem, halt, code)
ret (ST (pc,[], mem,halt,code)) =
        mkState (pc, [], mem, True, code)


{-
  The following function
  runs the machine for n steps starting from state s.
-}

sm s n =
        if (n == 0) then s else sm (step s) (n-1)

{-
  This is the program used as an example in the paper.
  If called with mem[0] = i and mem[1] = j,
  it leaves i * j in mem[2] and clears mem[0].
-}

prog :: Word a => [Op a]
prog = [
        MOVI 2 0,       -- 0, mem[2] <- 0
        JUMPZ 0 5,      -- 1, if mem[0]=0, goto 5
        ADD 2 1,        -- 2, mem[2] <- mem[1] + mem[2]
        SUBI 0 1,       -- 3, mem[0] <- mem[0] -1
        JUMP 1,         -- 4, goto 1
        RET             -- 5, return to caller
        ]

{-
  Here's an initial state for the machine to calculate 7 * 11.
-}

state1 :: State MachState Int
state1 = mkState
        (("TIMES",0),   -- program counter
         [],            -- empty stack
         [7,11,3,4,5],  -- memory
         False,         -- halt flag is currently false
        [("TIMES", prog)])

{-
```

```
  This functions gives us a shortcut, so we can just type ps1 x
  where x is the number of cycles we want to run prog
  on state1.
-}
ps1 :: Int -> IO()
ps1 x =
  printState (sm state1 x)

{-
  To calculate 7 * 11, execute ‘‘ps1 31’’
-}


{-
  In this example we have symbolic data in the data path but
  not in the control path.  For prog, this means j
  can be symbolic but not i.
  Here’s a state that runs the same program for 7 * j.
-}
state2 :: State MachState Symbo
state2 = mkState
        (("TIMES",0),   -- program counter
         [],            -- empty stack
         [7,Var "j",Var "x",Var "y",Var "z"],
         False,         -- halt flag is currently false
        [("TIMES", prog)])

ps2 x =
  printState (sm state2 x)

{-
  To calculate 7 * j, execute ‘‘ps2 31’’
-}


{-
  Next, we run the multiplication program with all values in
  memory being symbolic.
-}

state3 :: State MachState Symbo
state3 = mkState
        (("TIMES",0),      -- program counter
         [],               -- empty stack
         [Var "i",Var "j",Var "x",Var "y",Var "z"], -- memory
         False,            -- halt flag is currently false
         [("TIMES",prog)])

ps3 x =
  printState (sm state3 x)

{-
  To calculate i * j, execute ‘‘ps3 20’’
-}

{-
  Now let’s look at a program that takes longer to execute
  to get some timing figures on instructions per cycle for
  concrete and symbolic simulation.

  This program does 10000 repeated additions of 1000 and takes 40007
  cycles to complete.
-}

state4 :: State MachState Int
state4 = mkState
        (("MAIN",0),         -- program counter
         [],                 -- empty stack
         [0,0,0,0,0],            -- memory
         False,              -- halt flag
        [("TIMES", prog),   -- program memory
         ("MAIN", [MOVI 0 10000, -- 0, mem[0] <- 10000
                  MOVI 1 1000,  -- 0, mem[1] <- 1000
                  CALL "TIMES",
                  RET]
         )])

ps4 :: Int -> IO()
ps4 x = printState (sm state4 x)

{-
  To calculate 10 000 * 1000, execute ‘‘ps4 40007’’
-}

{-
  Moore’s benchmark for symbolic simulation

  This example runs 1000 * j in symbolic simulation. There
  are no branching states.  It take 4007 cycles to complete.
-}

state5 :: State MachState Symbo
```

```
state5 = mkState
        (("MAIN",0),
         [],
         [1000,Var "j",Var "x", Var "y", Var "z"],
         False,
        [("TIMES", prog),
         ("MAIN",[CALL "TIMES",
                  RET]
         )])

ps5 :: Int -> IO()
ps5 x = printState (sm state5 x)

{-
 execute ‘‘ps5 4005’’
-}

{-
  Bigger test case for concrete simulation
-}

state6 :: State MachState Int
state6 = mkState
        (("MAIN",0),
         [],
         [0,0,0,0,0],
         False,
        [("TIMES", prog),
         ("MAIN", [MOVI 0 100000, -- 0, mem[0] <- 100000
                  MOVI 1 1000,   -- 0, mem[1] <- 1000
                  CALL "TIMES",
                  RET]
         )])


ps6 :: Int -> IO()
ps6 x = printState (sm state6 x)

{-
  execute ‘‘ps6 400000’’
-}

{-
  Bigger test case for symbolic data flow
-}
```

```
state7 :: State MachState Symbo
state7 = mkState
        (("MAIN",0),
         [],
         [100000,Var "j",Var "x",Var "y", Var "z"],
         False,
        [("TIMES", prog),
         ("MAIN",[CALL "TIMES",RET])])

ps7 :: Int -> IO()
ps7 x = printState (sm state7 x)

{-
  execute ‘‘ps7 400000’’
-}

{-
  Test case for symbolic control flow
-}

state8 :: State MachState Symbo
state8 = mkState
        (("MAIN",0),
         [],
         [Var "i",Var "j",Var "x", Var "y", Var "z"],
         False,
        [("TIMES", prog),
         ("MAIN",[CALL "TIMES",
                  RET]
         )])

ps8 :: Int -> IO()
ps8 x = printState (sm state8 x)

{-
  execute ‘‘ps8 2000’’
-}
```