Symbolic Simulation of Microprocessor Models using Type Classes in Haskell

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

Oregon Graduate Institute, Portland, OR, USA
{nday, jlewis, byron}@cse.ogi.edu

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.

Symbolic simulation of microprocessor models written in the Haskell programming language [13] is possible without extending the language or its compilers and interpreters. Compared to rewriting in a theorem prover, symbolically simulating via evaluation of a program is generally much faster. 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.

Symbolic simulation in Haskell involves constructing a symbolic domain to represent the operations of the model. The values of this domain are syntactic representations of the machine's behavior. For example, using a recursive data type, an appropriate symbolic domain for numeric computations is:

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

Haskell's type class system allows us to make the operations that manipulate data be overloaded on both concrete and symbolic data. A type class groups a set of operations by the common type they operate over. For example, part of the `Num` class definition is:

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

Parentheses indicate that the operation is infix. The parameter `a` after the name of the class is a placeholder for types belonging in this class. 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 by the parser.

In Haskell, the type `Int` is declared to be an instance of the `Num` class. We also declare our symbolic data as an instance of `Num` using:

```
instance Num Symbo where
  x + y             = x `Plus` y
  x - y             = x `Minus` y
  x * y             = x `Times` y
  fromInt x          = Const x
```

Now without changing the microprocessor model, we can execute it for either concrete data or symbolic data.

The symbolic domain must behave consistently with 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. These rules can be implemented for the symbolic domain by augmenting the instance declaration for `Symbo` with cases that describe the algebraic rules. For example, instead of just having the rule `x + y = x `Plus` y`, we have:

```
Var x + Var y = if (x == y) then Const 2 * Var x
                else Var x `Plus` Var y
Const x + Const y = Const (x + y)
Const 0 + y       = y
x + Const 0       = x
x + y             = x `Plus` y
```

When control values in a program are symbolic, the output of symbolic simulation captures the multiple execution paths that the program could have followed. To deal with symbolic control values, we extend the idea of a state to include branches representing multiple execution paths. This leads us to have a symbolic representation of Boolean terms that are used to decide the branches. We introduce an abstraction of the "if-then-else" operation because the first argument to the if operator may be symbolic. A multi-parameter type class captures the behavior of our new `ifc`. A multi-parameter type class constrains multiple types in a single class instantiation. In the case of `ifc`, 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
    ifc :: a -> b -> b -> b
```

The normal "if-then-else" operator is an instance of this class with the type parameter `a` being `Bool` (concrete Booleans).

Further details and examples can be found in Day, Lewis, and Cook [7]. One of these examples walks through symbolic simulation in Haskell of the simple, non-pipelined, state-based processor model found in Moore [12].

We have also worked on symbolic simulation of a superscalar, out-of-order with exceptions, pipelined microprocessor model in the Haskell-based hardware description language Hawk [3, 11]. We are now able to simulate symbolic data flow for programs running on the model. We are currently extending the Hawk library to handle symbolic control paths as well. Because it is stream-based, the model does not have explicit access to its state. Hawk models usually process transactions, which capture the state of an instruction as it progresses through the pipeline. The key to having symbolic control flow is to have trees of transactions flowing along the wires rather than just simple transactions. Instead of simply having a top-level branching of state, the branching 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.

Symbolic simulation can be carried out with uninterpreted constants using rewriting in a theorem prover (e.g., [8, 9]) or using more specialized techniques such as symbolic functional evaluation [5]. Rewriting requires searching a database of rewrite rules and potentially following unused simplifications [12]. Constructors in our symbolic domains play the same role as uninterpreted constants in a logical model. Because our approach simply involves executing a functional program, we do not suffer a performance penalty for symbolic simulation compared with concrete simulation. Running on a platform roughly two and half times faster than Moore [12], we achieved performance of 58 300 instructions per second compared to ACL2's performance of 235 instructions per second (with hints) for the same non-pipelined, state-based processor model.

Type classes avoid the need to pass the operations to all the components of the model as in Joyce [10]. The type classes keep track of the higher-order function parameters that Joyce grouped in "representation variables".

The approach described in this paper is closely related to work on Lava [1], 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. 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.

Graph structures such as Binary decision diagrams (BDDs) [2] and Multiway decision diagrams (MDGs) [4] are canonical representations of symbolic formu-

lae. 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 [6].

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 makes symbolic simulation easy.

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

1. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.
2. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
3. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, 1998.
4. 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.
5. N. A. Day and J. J. Joyce. Symbolic functional evaluation. To appear in TPHOLs'99.
6. N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. Submitted for publication.
7. N. A. Day, J. R. Lewis, and B. Cook. Symbolic simulation of microprocessor models using type classes in Haskell. Technical Report CSE-99-005, Oregon Graduate Institute, 1999.
8. D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD*, volume 1522 of *LNCS*, pages 321–333. Springer, 1998.
9. J. Joyce. *Multi-Level Verification of Microprocessor Based Systems.* PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.
10. J. J. Joyce. Generic specification of digital hardware. In *Designing Correct Circuits*, pages 68–91. Springer-Verlag, 1990.
11. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.
12. J. Moore. Symbolic simulation: An ACL2 approach. In *FMCAD*, volume 1522 of *LNCS*, pages 334–350. Springer, 1998.
13. J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell.* Yale University, Department of Computer Science, RR-1106, 1997.