Logical Abstractions in Haskell

Nancy A. Day John Launchbury Jeff Lewis

Oregon Graduate Institute of Science & Technology

Abstract

We describe a generalization of the Haskell Boolean type, which allows us to use existing decision procedures for reasoning about logical expressions. In particular, we have connected Haskell with a Binary Decision Diagram (BDD) package for propositional logic, and the Stanford Validity Checker for reasoning in quantifier-free, first-order logic. We have defined referentially transparent interfaces to these packages allowing the user to ignore the details of their imperative implementations. We found that having a tight connection between the provers and Haskell allows Haskell to serve as a meta-language enhancing the capabilities of the provers. We illustrate the use of these packages for reasoning about a sort algorithm and a simple microprocessor model. In the sort example, the parametric nature of Haskell's polymorphism is used to lift the result of the BDD analysis to arbitrary datatypes.

1 Introduction

Here's some old advice: go through life like a swimming duck — remain calm and unruffled on the surface, but paddle like fury underneath. This advice applies to datatypes in programming languages. Consider the `Integer` datatype in Haskell. As far as the Glasgow Haskell compiler is concerned, the operations of `Integer` are implemented with calls to the GNU multi-precision arithmetic library, with all its ancillary mess of manipulating storage and pointers. Above the surface, however, is quite another story. As far as the user is concerned, `Integer` is just another numeric datatype with the usual arithmetic operations defined on it. It is quite possible to use `Integer` without thinking about the implementation mechanism at all. Of course, the plain numeric interface provided by Haskell is far poorer than the rich variety of methods provided by the full library. However, experience suggests that, for most users, a simple interface to a complex implementation provides far more benefit than a complex interface used simply.

The goal of this paper is set out on the same program for logical types like booleans. Excellent packages are now available that implement decision procedures for different logics, and we wondered whether clean interfaces could be built to allow the details of the decision procedures to be hidden under the surface.

Haskell's type class system was designed to solve a thorny problem in language design: how to combine overloading and polymorphism of numeric operators. The problem was motivated by the variety of numeric types. The solution was general enough to also solve several similar problems involving equality and printing. But, the notion of overloading booleans just didn't arise. However, several recent examples have made it clear that it's useful to be able to overload even simple types like booleans.

The Fran work on reactive animations demonstrates this point nicely [9]. In Fran, datatypes are lifted over time. An integer, for example, is replaced by a function from time to integer, and the numeric operations are defined pointwise. The same is done for equality. Are two time-varying integers equal? The answer is a time-varying boolean. By defining the boolean operations pointwise, it is easy to see that functions from time to `Bool` are fully "boolean".

Another example, and one which is the direct inspiration for this work, is the Voss verification system [20], used extensively for hardware verification. Voss uses a lazy functional language called FL as its interface language. In FL, booleans are implemented using Binary Decision Diagrams (BDDs) [4]. In effect, a decision procedure for propositional logic is built into the language, allowing the user to combine simulation and verification in powerful ways.

In this paper, we introduce two new flavors of booleans for Haskell. The first one follows FL by defining booleans using Binary Decision Diagrams. The improvement over FL is that we're able to do this by a mixture of type classes, the foreign function interface, and a little `unsafePerformIO` magic, rather than by designing and implementing (and maintaining!) a new language. For the second flavor of booleans, we extend the logic to quantifier-free predicate logic by using the Stanford Validity Checker (SVC) [2].

The implementations of each flavor are complex and have a strong imperative feel to them, but for both we have defined referentially transparent interfaces, allowing the underlying tools to do their work while the user simply sees the corresponding values. To some extent, this choice was forced upon us: we found that a fairly tight integration with SVC was necessary in order to avoid overly large intermediate data structures and to exploit the data sharing provided by SVC.

Even though both implement decision procedures for logics, BDDs and SVC are quite different in their approach. BDDs represent propositional formulae maintained in a canonical form. The results of operations are simplified incrementally, so equivalence between propositions is determined immediately by structural equivalence. In contrast, because it handles a richer logic, the basic SVC operations construct the problem *statement*. Much of the work is contained in testing logical equivalence, which involves a call to the prover. What pleased us about the embedding in Haskell is that both approaches are implemented in the same framework, so the user has great freedom to decide which is appropriate for the task.

The tight connection between the various provers and Haskell allows Haskell to be used very naturally as a meta-language, in effect enhancing the capabilities of each of the logics. In the BDD case, we have an example where the parametric nature of Haskell's polymorphism can be used to lift the result of the BDD analysis to arbitrary datatypes. In the SVC case, we present an example where we introduce an uninterpreted function, but use the expressiveness of Haskell to generate a limited axiomatization of it.

In summary, the goal of this paper is to describe a new easy-to-use power tool for the Haskell programmer's workbench. Applications include verification of Haskell programs within Haskell. This suggestion immediately brings to mind visions of higher-order logics, but for now we'll forgo generality in favor of the automation of simpler logics.

The rest of the paper is organized as follows. Section 2 presents the Boolean class. In Sections 3 and 4 we describe BDDs, and provide an example leveraging the structure of Haskell. In Section 5 we do the same for SVC and in Section 6 we present a larger worked example using the power of SVC. The remaining sections present discussion.

2 Logical Type Classes

We now do for Bool what the Num class does for numeric types. That is, we define a type class signature for operations over booleans. It contains all the usual suspects, plus implication (==>), mutual implication (<=>), and if-then-else (ifb).

```
class Boolean b where
    true :: b
    false :: b
    (&&) :: b -> b -> b
    (||) :: b -> b -> b
    (==>) :: b -> b -> b
    (<=>) :: b -> b -> b
    not :: b -> b
    ifb :: b -> b -> b -> b
```

Bool is of course an instance of class Boolean.

We also need to refer to logical variables, which are not an aspect of the Bool datatype. Thus, we introduce a new class for logical variables.

```
class Var a where
    var :: String -> a
```

For example, here's a little proposition about distributing && over implication:

```
(var "a" ==> var "b") && (var "c" ==> var "d")
==>
(var "a" && var "c") ==> (var "b" && var "d")
```

Of course, Bool is used in many places in the prelude. One place that it shows up is in the definition of equality. We define a variant of the Eq class where the boolean result is abstract.

```
class Boolean b => Eql a b where
    (===) :: a -> a -> b
```

```
data BinTree a t =
      Terminal t
    | Branch a (BinTree a t) (BinTree a t)
    deriving Eq

cofactor a (Terminal x) =
    (Terminal x, Terminal x)
cofactor a c@(Branch b x y) =
    if a == b then (x, y) else (c, c)

top2 x y =
    a `min` b
    where
        a = index x
        b = index y
        index (Terminal _) = maxBound
        index (Branch a _ _) = a

norm b@(Branch a x y) = if x == y then x else b
norm x = x

bddBranch a x y =
    let a' = top2 x y
        (x1, x2) = cofactor a x
        (y1, y2) = cofactor a y
    in
        if a <= a' then
            norm (Branch a x1 y2)
        else
            norm (Branch a' (bddBranch a x1 y1)
                            (bddBranch a x2 y2))
```

Figure 1: Bdd normalization

3 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a representation of boolean functions as a binary tree. The nodes are labeled by boolean variables, and the leaves, usually referred to as *terminals*, are boolean values. A BDD represents a boolean formula in the form of a case analysis.

An Ordered BDD (OBDD) is a normal form for BDDs defined as follows:

- the variables along every path are in strictly increasing order, and
- all unnecessary nodes are eliminated. An unnecessary node represents a variable upon which the subtree doesn't depend, and is recognized as a node where both left and right children are equal.

An Ordered BDD is further a canonical form for boolean formulae, thus equality of two OBDDs is reduced to structural equality.

Construction of OBDDs can be easily described in Haskell, as we show in figure 1. Creating a new terminal is easy—just use the Terminal constructor. Creating a new branch (bddBranch) is a matter of pushing the branch down the tree until it's label is in order. Notice that as a node is moved down the tree, its branches get duplicated. This duplication may be countered by the pruning action of the second restriction (implemented by `norm`). But in general, the worst case size complexity of a BDD is exponential in the number of variables.

An important refinement that makes BDDs a practical tool for large scale verification is Reduced Ordered BDDs (ROBDDs)[1] [4]. An ROBDD is one in which the tree is reduced to a directed acyclic graph (DAG) by performing common subexpression elimination to capture all structure sharing. This has two benefits. The first is that equivalence of BDDs is reduced to pointer equality. The second is a reduction in space use, which can be considerable when there's a significant amount of regularity in the data. Despite the worst-case size complexity of BDDs, the worst case is often avoided in practice.

However, there's another aspect of sharing that isn't captured by just using a DAG. The regularity that leads to a lot of structural sharing also leads to performing the same boolean calculations on a given BDD over and over again—you may do a lot of work just to realize that you've already constructed this tree before. Thus, another important trick in a BDD implementor's bag is to keep a memo table for each basic boolean operation so that the operation is only performed once on any given BDD (or pair of BDDs).

It's an interesting question whether a Haskell implementation of BDDs would offer any improvement over a highly-tuned off-the-shelf BDD library written in C. We've pursued this question by doing prototypes that use the latest features of Hugs and GHC to implement structure sharing and memoizing. The results are promising, but it's clear that Haskell isn't really ready to beat C at its best game. So maybe we're trying the wrong strategy. All of these structure-sharing and memoizing mechanisms make BDDs strict, whereas the simple implementation of OBDDs sketched above is lazy. An even more interesting question may be whether there's some way to play off of Haskell's strengths and take advantage of laziness. This question is pursued further in Section 7.1.

3.1 The CMU BDD Library

The BDD library used in this paper is David Long's BDD library (which we'll refer to as the CMU library) [16], which is an ROBDD package written in C, and is one of several high-quality ROBDD packages that are available. We use the CMU BDD package as distributed with the VIS suite, version 1.3, from Berkeley [1]. As with other BDD packages, it comes with its own garbage collector that has been tuned to work well with BDDs.

We import the CMU BDD package into Haskell with the help of H/Direct [10]. Using the foreign function interface of either GHC or Hugs, you can interface to libraries written in various imperative languages, such as C, FORTRAN, Java, etc. H/Direct helps simplify that process by automatically generating the glue code that marshals and unmarshals datatypes from C/FORTRAN/Java to Haskell. The input to H/Direct is a specification of the external library's interface, written in IDL, an Interface Description Language used in industry. IDL interface descriptions are essentially annotated C declarations. In the case of the BDD library, it was a fairly easy matter to translate the header file from the library into an IDL description. For example, here's the snippet of IDL describing the BDD implementations for the boolean constant `true`, and boolean conjunction:

```
cmuBdd cmu_bdd_one ([in]cmuBddManager bddm);
cmuBdd cmu_bdd_and ([in]cmuBddManager bddm,
                    [in]cmuBdd x, [in]cmuBdd y);
```

The `[in]` annotations indicate that the argument is an input argument only (i.e. it doesn't return a result by side-effect).

The first parameter to both functions is common to all BDD calls and is the structure that holds all the context necessary for managing BDDs, such as hash tables, and garbage collection data.

The signatures that H/Direct generates for these two functions are as follows:

```
cmu_bdd_one :: CmuBddManager -> IO (CmuBdd)
cmu_bdd_and :: CmuBddManager ->
               CmuBdd -> CmuBdd -> IO (CmuBdd)
```

3.2 Managing BDDs

Of course, the raw imported interface to the BDD package is not exactly the svelte duck that we were after in the beginning. There are two things in the way. First, the `CmuBddManager` is an implementation artifact that we would rather hide from users. Second, the result lies in the `IO` monad. Our plan is to hide the fact that we are using an imperative implementation underneath by liberal use of `unsafePerformIO`. We will then to justify why it's really safe after all.

In the C world, the types `CmuBddManager` and `CmuBdd` are pointers to rich structures. In Haskell, all of that is hidden, and they appear as type synonyms for the type `Addr`, a type which only supports pointer addition and pointer equality. However, even this is more than we want to expose about BDDs, so we define a new abstract type for BDDs.

```
newtype BDD = Bdd CmuBdd
```

The BDD manager must ordinarily be explicitly allocated at the beginning of a program. We would rather it was allocated on demand without any muss or fuss from the programmer. We use `unsafePerformIO` to get this effect:

```
bdd_manager :: CmuBddManager
bdd_manager = unsafePerformIO (cmu_bdd_init)
```

In the instance declarations for `Boolean`, we distribute the `bdd_manager` to all the calls, and the extract the result from the `IO` monad with `unsafePerformIO`.

```
instance Boolean (Bdd Bool) where
  true = Bdd $ unsafePerformIO $
         bdd_one bdd_manager
  (Bdd x) && (Bdd y) =
      Bdd $ unsafePerformIO $
      cmu_bdd_and bdd_manager x y
  ...
```

The `$` operator is right-associated function application (`f $ g $ x = f (g x)`).

Dealing with variables brings up another detail of the machinery that we wish to hide. The BDD package uses integers as variable identifiers, but we'd rather provide the nicer interface of using strings as variable identifiers. Thus, we keep a table that assigns to each variable name encountered so far a unique integer.

[1] Subsequently, we will use the term BDD to mean ROBDD.

```
bdd_vars :: IORef [(String, Int32)]
bdd_vars = unsafePerformIO (newIORef [])
```

The instance of BDDs for the Var class is then defined in terms of a function that keeps track of the necessary bookkeeping. The function newvar takes a reference to the lookup table, and a variable identifier, and constructs a BDD representing that variable.

```
newvar :: CmuBddManager ->
          IORef [(String,Int32)] -> String ->
          IO CmuBdd
newvar m v a =
    do  vars <- readIORef v
        case lookup a vars of
          Just i -> bdd_var_with_id m i
          Nothing ->
            do  b <- bdd_new_var_last m
                i <- bdd_if_id m b
                writeIORef v ((a, i) : vars)
                return b

instance Var (Bdd Bool) where
    var a = Bdd $ unsafePerformIO $
            newvar bdd_manager bdd_vars a
```

3.3 Safe use of unsafePerformIO

All this use of unsafePerformIO may seem a bit blithe, so it's worth a few moments to informally justify why it doesn't harm referential transparency.

For starters, since the types CmuBddManager and CmuBdd are essentially abstract, we're in control of what can be observed about the imperative side. We simply do not import operations from the CMU package that reveal the imperative structure, such as a procedure that gives the size of internal hash tables. Those functions that we do import, despite all the operational goings-on with hash tables and heap allocation, are essentially functional. The imperative features affect the way the data is constructed, but not the data itself, and we give ourselves no way to observe the imperative details of how the data is constructed.

We do allow ourselves to do pointer equality on CmuBdds, which is how structural equality is tested. But since the pointers will be equal if-and-only-if the structures are equal, this is safe.

One point to question is the use of unsafePerformIO in the definition of bdd_vars to extract an IORef out of the IO monad. By doing this and then using the IORef elsewhere inside the IO monad, we're making the assumption that the store inside the IO monad is indeed persistent, and that IORefs are coherent between invocations of the IO monad. Fortunately, although this behavior is not, to the authors' knowledge, documented, it is at least the most reasonable assumption to make. After all, the IO monad is supposed to represent the "real world", which has the behavior of being persistent.

4 An Example: Sorting

BDDs give us the ability to show equivalence of boolean functions. This is useful of itself, but in this section we show how the structure of Haskell can be used to imply much richer results. The example we take is from sorting.

4.1 Comparison-Swap Sorting

Knuth has a famous theorem about sort algorithms that are based only on comparison-swaps (an operation that takes two elements and returns them in sorted order). The theorem states that if such an algorithm is able to sort booleans correctly, it will also sort integers correctly. Knuth's proof is based in decision-theory, and is specific to sorting. We have discovered that Knuth's result is a special case of the more general theorems coming from polymorphic parametricity.

This suggests the following proof technique. Take a polymorphic function, perform some verification using BDDs at the boolean instance, and then use the parametricity theorem to deduce a corresponding result for more complex types. In effect, the boolean instance acts as an abstract interpretation of the more general algorithm, and parametricity supplies the abstraction and concretization relationships.

To see how this works in practice, we first need to consider the type of a comparison-swap sort. We want to avoid the manual examination of the program text that an application of Knuth's theorem would require. What can we learn from the type alone? The type cannot tell us that the algorithm is a sorting algorithm, but it can ensure that it makes no assumptions about the contents of the list, except via the first parameter. Consider the type:

```
sort :: ((a,a)->(a,a)) -> [a] -> [a]
```

If the sort argument is indeed a comparison-swap function then, intuitively, the type of sort ensures that the only data-sensitive operation sort can use is comparison-swap.

Let's make this precise. Consider the parametricity theorem for functions of this type [19, 23] (f, g, and h are universally quantified, and we define $j \times k = \lambda x, y.(j\ x, k\ y)$):

$$\begin{tikzcd}
(a,a) \arrow[r, "f"] \arrow[d, "h \times h"'] & (a,a) \arrow[d, "h \times h"'] \arrow[rrd, phantom, "\Rightarrow"] & & {[a]} \arrow[r, "\texttt{sort}\ f"] \arrow[d, "\texttt{map}\ h"'] & {[a]} \arrow[d, "\texttt{map}\ h"'] \\
(b,b) \arrow[r, "g"] & (b,b) & & {[b]} \arrow[r, "\texttt{sort}\ g"] & {[b]}
\end{tikzcd}$$

In Haskell, the theorem applies only when h is strict and, since the introduction of seq in Haskell 98, bottom-reflecting as well.

Now, instantiate a to be Int, and b to be Bool, and chose f and g to be the standard comparison/swap over integers and booleans (where False $<$ True) respectively. If we can show that sort g sorts sequences of booleans correctly (using BDDs for example), then the parametricity theorem will allow us to conclude that sort f sorts sequences of integers correctly as well.

To see this, suppose the converse, and we will derive a contradiction. Suppose xs contains x and y, such that $x < y$. For sort f to be incorrect, there has to exist at least one x and y pair which appears out of order (y before x) in the list sort f xs. Let h be the function that is false for all inputs less than y, and true otherwise ($h(n) = y <= n$). This function commutes with f and g, and is strict and bottom-reflecting as well, thus it satisfies the precondition for the theorem. Therefore, the right-hand side of the theorem must hold.

Now, by assumption, `sort` g (`map` h xs) is sorted correctly. That is, the result is a list of booleans with all the occurrences of `False` preceding the occurrences of `True`. However, if y precedes x in the result of `sort` f xs then the result of `map` h(`sort` f xs) will contain an occurrence of `True` before the final occurrence of `False`. Thus, we have a contradiction, so the assumption that y preceded x in the result of `sort` f xs was incorrect.

In effect, parametricity has ensured that `sort` behaves coherently over all types, so that results at the boolean instance can be used to imply consequences at other types. Another perspective is that parametricity expresses the multi-faceted symmetry inherent in this problem Symmetry is vital in verification by model checking for reducing large problem spaces to manageable proportions, and that is what is achieved here [8]. Rather than model check on lists of 32-bit integers, we perform the check on single-bit integers.

4.2 Checking Comparison-Swap Sort on Booleans

The missing part of the story is using BDDs to show that `sort` g sorts lists of booleans correctly. We can only show that sorting is correct for an arbitrary, but fixed-length list. The logic of BDDs is simply not powerful enough to prove the result in general (which would require some kind of inductive argument—well beyond the scope of propositional logic). However, since the method is automated, it's no trouble to check it for a variety of lengths of list, leaving only very subtle bugs out of its reach.

The sorting algorithm we use is bitonic sort, an efficient algorithm that is particularly amenable to hardware realization. Also, for a sorting algorithm, it's fairly tricky, so it's a good candidate for verification. The code is given in Figure 2. Bitonic sort is designed to work on lists that have a length that is a power of two. It recursively divides the list into two parts and sorts each partition. To combine the two parts it swaps corresponding elements in each list. Because one list is sorted in ascending order and the other in descending order, the swapping results in all the elements in the first list being lower (or higher) than the elements in the second list. The two lists are in the correct form to repeat this swapping on the two sublists to arrive at a sorted list.

To verify the algorithm, we first need a simple predicate to indicate whether a list is sorted or not.

```
sorted test [] = true
sorted test [x] = true
sorted test (x : ys@(y : _)) =
    x ‘test‘ y && sorted test ys
```

Now, we will state the property that we want to show for a list with variable elements, but a fixed length of sixteen.

```
result = sorted lessEq (sort xs)
  where
    xs = [ var ("x" ++ show i) | i <- [0 .. 15] ]
    sort xs = bitonic_sort cmpSwap xs True
```

It remains to define the two BDD-specific functions `cmpSwap` and `lessEq`. These turn out to be particularly nice.

```
cmpSwap a b = (a && b, a || b)

lessEq a b = a ==> b
```

Now, when we query Haskell about `result`, it returns `true`.

```
bitonic_to_sorted cmpSwap [] up = []
bitonic_to_sorted cmpSwap [x] up = [x]
bitonic_to_sorted cmpSwap xs up =
  let k = length xs ‘div‘ 2
    (ys, zs) = pairwise cmpSwap (splitAt k xs)
    (ys’, zs’) =
      if up then (ys, zs) else (zs, ys)
  in
    bitonic_to_sorted cmpSwap ys’ up ++
    bitonic_to_sorted cmpSwap zs’ up

pairwise f ([], []) = ([], [])
pairwise f (x : xs, y : ys) =
  let (x’, y’) = f x y
    (xs’, ys’) = pairwise f (xs, ys)
  in
    (x’ : xs’, y’ : ys’)

bitonic_sort cmpSwap [] up = []
bitonic_sort cmpSwap [x] up = [x]
bitonic_sort cmpSwap xs up =
  let k = length xs ‘div‘ 2
    (ys, zs) = splitAt k xs
    ys’ = bitonic_sort cmpSwap ys True
    zs’ = bitonic_sort cmpSwap zs False
  in
    bitonic_to_sorted cmpSwap (ys’ ++ zs’) up

cmpSwap x y = if x < y then (x, y) else (y, x)
```

Figure 2: Bitonic Sort

4.3 Limitations to using Parametricity

We expect the verification technique outlined above to be useful in many cases, but it's not a panacea. Sometimes parametricity is not powerful enough to capture appropriate abstractions. In effect, some types are simply not expressive and/or constraining enough to enable the boolean instance to say much about the general case. Consider the following variation on the example above.

It might seem that the parametricity argument that we used to echo Knuth's sorting theorem would apply just as easily to a regular sort algorithm based on a comparison function, with type:

```
sort :: ((a, a) -> Bool) -> [a] -> [a]
```

However, it is fairly easy to construct a pseudo-sorting algorithm of this type that will correctly sort lists of booleans but fails to sort lists of integers correctly. Consider the following: take the first element of the list as a partition value. Next, do a one-pass sort into three buckets: one for elements less then the partition, one for those equal (neither less, nor greater), and one for those greater. Finally, stick the partition element in the equal bucket, and concatenate the buckets in the order: less, equal and greater. Partitioning based upon a single element will work for booleans, because there's only two values; however, it clearly won't work in general.

So, the parametricity-based approach must fail for comparison-based sorts. Where does it break down? First, examine the "free theorem" for a comparison-based sort.

$$\begin{tikzcd}
(a,a) \arrow[r, "f"] \arrow[d, "h \times h"'] & \texttt{Bool} \arrow[d, equal] & {} \arrow[d, phantom, "\Rightarrow"] & {[a]} \arrow[r, "\texttt{sort}\ f"] \arrow[d, "\texttt{map}\ h"'] & {[a]} \arrow[d, "\texttt{map}\ h"'] \\
(b,b) \arrow[r, "g"] & \texttt{Bool} & {} & {[b]} \arrow[r, "\texttt{sort}\ g"] & {[b]}
\end{tikzcd}$$

The conclusion is identical in each instance of the parametricity theorem, but the precondition of this instance is much more stringent than before. The key to proving the comparison-swap case was the ample supply of appropriate functions h to "detect" any incorrectly sorted list. However, the precondition on h in this case requires that the comparisons on the two sorts, integer and boolean say, are equivalent to one another. Thus for the case of comparison sort there are essentially no interesting choices for h relating the integer and boolean cases.

5 The Stanford Validity Checker

The Stanford Validity Checker (SVC) is an implementation of a decision procedure for a quantifier-free, first-order logic with equality [2, 7, 11]. It has been used extensively for microprocessor validation and verification [7, 11, 12, 22] and recently for requirements validation [18]. The logic allows models to include uninterpreted functions, which can be used to represent datapath operations in a pipelined architecture. SVC returns a counterexample if the formula is not valid.

$$\begin{array}{rcl}
\mathit{formula} & ::= & \text{ite}\ (\mathit{formula}, \mathit{formula}, \mathit{formula}) \\
& | & (\mathit{term} = \mathit{term}) \\
& | & \mathit{predicate\ symbol}\ (\mathit{term}, ..., \mathit{term}) \\
& | & \text{true} \\
& | & \text{false} \\
\\
\mathit{term} & ::= & \text{ite}\ (\mathit{formula}, \mathit{term}, \mathit{term}) \\
& | & \mathit{function\ symbol}\ (\mathit{term}, ..., \mathit{term}) \\
& | & \text{read}\ (\mathit{term}, \mathit{term}) \\
& | & \text{write}\ (\mathit{term}, \mathit{term}, \mathit{term}) \\
& | & \mathit{distinct\ constant} \\
& | & \mathit{formula}
\end{array}$$

Figure 3: The SVC logic

5.1 Connecting SVC with Haskell

Our initial interface with this tool was file-based. We had a representation of expressions in the logic as a datatype in Haskell and wrote expressions of this form to a file that was later read by SVC. As we worked on larger examples, this approach became unmanageable. The size of the structure was extremely large and did not take advantage of possible sharing of subexpressions. While SVC's internal data structure is not canonical as is the case for BDDs, it is optimized and shares common subexpressions. Thus it quickly became apparent that a much better approach is to have a tight link between the process of generating the term and building the term in SVC. Using H/Direct we were able to create an abstract interface to the SVC C++ functions that build the expressions. We used version 1.1 of SVC.

As it is a richer logic, SVC expressions include more than just boolean-valued terms. Figure 3 contains a description of the SVC logic. The predicate and function symbols introduce uninterpreted predicates and functions. The functions `ite` and `=` are interpreted functions representing "if-then-else" and equality. The functions `read` and `write` are interpreted as acting on stores; an axiom of the logic relating these functions is, `read (write (store, index, data), index) = data`. Other logical, numeric, bit vector and record operations also have an interpreted meaning.

Using H/Direct we created an interface to SVC that has functions for building each of the kinds of terms and formulae. These functions return elements of the type `PExpr`, which are pointers to SVC expressions. The interface functions to SVC that build expressions in the logic do not distinguish between terms and formulae.

As with the BDD package, the calls to the SVC functions are wrapped in `unsafePerformIO` to extract the value from the IO monad. Because the only way to observe the SVC expressions is to check their validity, the meaning of an expression is the same regardless of its order of construction. Therefore we can use the term building functions as if they are referentially transparent.

Only a subset of the SVC expressions, the formulae, can be used to instantiate the Boolean class. Even though the underlying package doesn't distinguish between terms and formulae, we want Haskell to make this distinction so that the Boolean class is only instantiated for formulae. We create the datatypes `SvcFormula` and `SvcTerm` to wrap around the pointers to expressions that SVC returns to make them distinct types.

```
newtype SvcFormula = SvcF PExpr
newtype SvcTerm = SvcT PExpr
```

We wrap the output of the SVC functions with `SvcF` or `SvcT` as appropriate. The arguments to the function must be unwrapped.

Using Haskell's type system to distinguish between terms and formulae in SVC's logic, we instantiated the Boolean class using only the formulae of SVC.

```
instance Boolean SvcFormula where
  true = SvcF $ unsafePerformIO $ Svc.makeTrue
  (SvcF a) && (SvcF b) =
    SvcF $ unsafePerformIO $ Svc.makeAnd a b
  ...
```

The functions `Svc.makeTrue` and `Svc.makeAnd` are calls to the SVC package.

In SVC the equality operator is also used to create formulae. This operator is an instance of the generalized equality class:

```
instance Eql SvcTerm SvcFormula where
  (SvcT a) === (SvcT b) =
    SvcF $ unsafePerformIO $ Svc.makeEquals a b
```

Because SVC has both terms and formulae, there are functions that create terms. We provide wrappers for these functions as well. For example, `fcn` creates an uninterpreted function application, where the first string argument is the name of the function, and the arguments to the function are provided in a list:

```
fcn a bs =
  SvcT $ unsafePerformIO $
    (if (bs==[]) then Svc.makeSymbol a
                 else Svc.makeUninterpretedFcn a
                        (args bs))
```

The function `args` turns the Haskell list of terms into the SVC form.

The SVC package has two instantiations of the `Var` class – one for formulae, and one for terms.

```
instance Var SvcFormula where
  var a = SvcF $ unsafePerformIO $ Svc.makeSymbol a

instance Var SvcTerm where
  var a = SvcT $ unsafePerformIO $ Svc.makeSymbol a
```

Type annotations are sometimes necessary to distinguish which instance of `var` is being used in a Haskell program.

The interface includes the SVC function `checkValid` to call the prover on the constructed expression. Calls to `checkValid` are referentially transparent because our interface tells SVC to treat each check independently from any other calls to the prover. We pop its stack of knowledge about a particular proof session (context), but retain its data about the expressions that have been built.

5.2 Sort Example

SVC is able to check the sort algorithm presented in Section 4 for a fixed length list of elements without the parametricity meta reasoning because SVC can reason over arbitrary types. To do this, we provided different definitions for `lessEq` and `cmpSwap`. We made the "less than" operator an uninterpreted predicate replacing its use with `pred "lt" [a,b]`. We also used the SVC "if-then-else", namely `itet`.

```
INVALID
Falsifying Assumptions
=====================
Assert:

  $92:(lt $7:a1 $11:a3)
Deny:

  $85:(lt $8:a2 $7:a1)
Deny:

  $55:(lt $8:a2 $11:a3)
Deny:

  $57:(lt $7:a1 $12:a4)
Deny:

  $13:(lt $11:a3 $12:a4)
Deny:

  $9:(lt $7:a1 $8:a2)

INVALID
Case_Splits:    7
Exprs_Generated:        57
```

Figure 4: SVC counterexample

```
cmpSwap x y =
  let test = pred "lt" [x,y] in
  (itet test x y, itet test y x)

lessEq x y = not (pred "lt" [y,x])
```

The prover was invoked to determine if a fixed length list of symbolic elements is sorted, as in:

```
result = checkValid
           (sorted lessEq (sort xs)) where
  sort xs = bitonic_sort cmpSwap xs True
  xs = [var "a1", var "a2", var "a3", var "a4"]
```

SVC returned with a counterexample found in Figure 4. The counterexample is in the form of a series of assertions and denials of subformulae. The "$" variables refer to internal subexpression names. The case provided in Figure 4 has both $\neg(a2 < a1)$ and $\neg(a1 < a2)$, which means $a2$ must equal $a1$. The case also says that $a1 < a3$ and $\neg(a2 < a3)$, which is impossible when $a1$ and $a2$ are equal, and $<$ has its intended meaning. From this counterexample, we learned that we cannot achieve our verification result without providing more information about the behavior of the "less than" operator.

SVC has an interpreted "less than" function for rational expressions that we could use. But we wished to check the sort algorithm for all types of ordered elements without any meta reasoning. SVC needed the information that the "less than" operator is irreflexive, transitive, and that $x \neq y \Rightarrow (x < y = \neg(y < x))$. We provided these in the form of antecedents to the consequent that we wanted to check. This is a limited axiomatization of the "less than" operator.

The SVC logic has no quantifiers so it was necessary to generate all the possible instantiations of these properties

$$\begin{tikzcd}
pipeState \arrow[r, "\texttt{pipe input}"] \arrow[d, "\texttt{pipe flush}"'] & pipeState' \arrow[d, "\texttt{pipe flush}"] \\
{} \arrow[d, "\texttt{pipe flush}"'] & {} \arrow[d, "\texttt{pipe flush}"] \\
{} \arrow[d, "\texttt{pipe flush}"'] & {} \arrow[d, "\texttt{pipe flush}"] \\
\ldots & \ldots \\
{} \arrow[d, "\texttt{proj}"'] & {} \arrow[d, "\texttt{proj}"] \\
refState \arrow[r, "\texttt{refMachine input}"] & refState'
\end{tikzcd}$$

Figure 5: Burch and Dill Commuting Diagram (found in [7])

for the symbolic elements in the input list. Haskell's list comprehension syntax was very convenient for stating, in a compact form, all the antecedents that were needed. For example, transitivity of a relation `r` for a list of elements is expressed as:

```
trans r x y z = (r x y && r y z) ==> r x z

genTrans list r =
  foldr1 (&&) [trans r x y z |
                   x <- list,
                   y <- list,
                   z <- list]
```

With these antecedents, SVC returns instantly for the bitonic sort of 4 elements saying that the sort algorithm is correct.

6 Example 2: Microprocessor Verification

As a second example of the use of SVC in Haskell, we present the verification of a simple pipelined ALU used in Burch and Dill [7] (originally found in Burch et al. [6]). In their presentation, they use a simple hardware description language based on Lisp as input to their verification process. This section describes how this example can be verified in Haskell using SVC and uninterpreted functions for the datapath operations.

The Burch and Dill approach to verification automatically calculates an abstraction function relating a microprocessor pipeline to a reference machine. The calculation is done using symbolic simulation. The pipeline is equivalent to the reference machine if the diagram in Figure 5 commutes. The abstraction function consists of flushing the intermediate results of the pipeline and projecting from the pipeline only the parts of the state visible in the reference machine (`proj`).

Figure 6 is a pipeline and reference machine modeled in Haskell, translated from the descriptions in Lisp found in the appendix of Burch and Dill's paper. The state of the reference machine is simply the register file. The reference machine and the pipeline are compared on the value of the register file only. The register file element of the pipeline is projected from its state as part of the correctness statement.

Because we leave the ALU operation as an uninterpreted function, our verification using SVC does not depend on the datapath width and operations. For the operations on the register file, the interpreted write and read SVC functions are used. The SVC instantiations of the Boolean operators and equality (`===`) are chosen automatically.

To verify the pipeline, we stall it to flush its state by setting the stall signal of the input high for a certain number of steps. The flush input contains symbolic values for every input other than the stall signal.

```
flush = (true,
         var "flushDestReg",
         var "flushOpcode",
         var "flushSrc1Reg",
         var "flushSrc2Reg")

flushPipe initialState n =
  if (n == 0) then initialState
  else flushPipe (pipe flush initialState) (n-1)
```

The `initialState` also assigns symbolic values to all the internal latches of the pipe:

```
initialState = (var "registers",
                var "arg1",
                var "arg2",
                var "bubble_wb",
                var "dest_wb",
                var "result",
                var "bubble_ex",
                var "dest_ex",
                var "op_ex")
```

To calculate the left and bottom route of the commuting diagram, we flush the pipeline, project out a state for the reference machine, and run the reference machine on this initial state with symbolic input:

```
proj (registers, _,_,_,_,_,_,_,_) = registers

input = (var "stall",
         var "dest",
         var "opcode",
         var "src1",
         var "src2")

path1 n = refMachine input
            (proj (flushPipe initialState n))
```

We compare `path1` with the other side of the commuting diagram. In `path2`, we run the pipeline on the symbolic input, starting from the symbolic initial state, and then flush the pipeline:

```
path2 n = proj
        (flushPipe (pipe input initialState ) n)
```

These two paths are computed by executing the Haskell models. On any path in the pipe there are at most two latches, therefore the pipe should agree with the reference machine after two flushes. The verification condition that we pass to the prover to be checked is:

```
pipeTest = (path1 2) === (path2 2)
```

```
type Input =
    (SvcFormula,  -- stall
     SvcTerm,     -- dest
     SvcTerm,     -- opcode
     SvcTerm,     -- source1
     SvcTerm)     -- source2

type PipeState  =
    (SvcTerm,     -- register file
     SvcTerm,     -- arg1
     SvcTerm,     -- arg2
     SvcFormula,  -- bubble-writeback
     SvcTerm,     -- dest-writeback
     SvcTerm,     -- result
     SvcFormula,  -- bubble-ex,
     SvcTerm,     -- dest-ex,
     SvcTerm)     -- opcode

pipe :: Input -> PipeState -> PipeState
pipe (stall, dest,opcode,src1,src2)
  (registers,arg1,arg2,bubble_wb,dest_wb,
         result,bubble_ex,dest_ex,op_ex) =
  (registers',arg1',arg2',bubble_wb',dest_wb',
         result', bubble_ex',dest_ex',op_ex')
    where
    registers' = itet bubble_wb
          registers
          (write registers dest_wb result)
    bubble_wb' = bubble_ex
    dest_wb' = dest_ex
    result' = fcn "alu" [op_ex, arg1, arg2]
    bubble_ex' = stall
    dest_ex' = dest
    op_ex' = opcode
    arg1' = itet ((not bubble_ex) &&
                 (dest_ex === src1))
            result'
            (read registers' src1)
    arg2' = itet ((not bubble_ex) &&
                 (dest_ex === src2))
            result'
            (read registers' src2)

type RefState = SvcTerm   -- register file

refMachine :: Input -> RefState -> RefState
refMachine (stall, dest, opcode, src1, src2)
           registers =
    itet stall
    registers
    (write registers dest
              (fcn "alu" [opcode,
                          read registers src1,
                          read registers src2]))
```

Figure 6: Microprocessor models

SVC verifies pipeTest instantly.

Integrating SVC with Haskell creates a very convenient debugging loop when there are errors in the model. Using the information in a counterexample, concrete values can be input to the model to illustrate the error.

7 Discussion

The section discusses some interesting points that have been raised in creating these logical abstractions.

7.1 Shallow versus Deep Embedding

In order to look most like a duck, we've taken the approach of doing a shallow embedding of both BDDs and SVC. This means we directly interpret the logical operators as operations on the internal data structures of the BDD package and SVC. An alternate approach is a deep embedding, where we construct an intermediate data structure that is exactly (or close to) the term structure.

One benefit of the deep embedding is that it gives us the opportunity to tackle the normalization process in different ways that may be more efficient. By analogy, when constructing BDDs incrementally, we must use essentially a bubble sort to put the nodes in sorted order. The incremental approach is necessarily based on local decisions, but we know that sorting is suboptimal when it is restricted to making local decisions. Thus, we can imagine being able to do something analogous to mergesort to put a BDD in normal form much more efficiently.

This approach is not considered feasible in the strict setting of a C implementation, because the intermediate data structure would be huge, and space is more of a limiting factor with BDDs than speed. The intermediate data structure would not be able to take advantage of any sharing. Thus, the only feasible approach is to calculate the sharing as you go.

However, in the setting of a lazy functional programming language, we have more options. Because the intermediate data structure doesn't necessarily get built, we may be able to take advantage of laziness to process BDDs more efficiently, while not taking a hit in space usage.

Another argument in favor of a deep embedding is that we could let Haskell control decomposition or simplification before calling the decision procedure. Haskell could become a platform for building "minimal proof assistants" [17] combining evaluation for term generation, decision procedures, and theorem proving techniques.

7.2 Types

So far, we are not making too much use of Haskell's rich type system. The only type distinction that we make in SVC expressions is between booleans and any other kind of term. We are working on building a typed layer on top of SVC logical terms where we regain the typechecking benefits of Haskell. This layer will make extensive use of type classes letting Haskell do the work of choosing the correct instances of functions rather than the user.

7.3 Ambiguity

One unfortunate consequence of generalizing booleans to a type class is that ambiguity problems can arise left and right. Booleans are used all over the place as intermediate values, especially in if-then-else expressions. Intermediate types in expressions don't show up in the type of the overall expression, and thus the type class system has no basis upon which to chose which instance to use. The same scenario holds for the Num class, but Haskell resolves this by the default mechanism. It would be helpful if the default mechanism could be made more general, such that we could talk about defaults for Boolean as well.

7.4 BDD Variable Order

As was pointed out in the introduction, in our zeal to put a pretty face on complex implementation packages, we give up a good deal of control.

For the sake of simplicity the interface that we provide to the BDD package leaves the user unaware of the details of variable order when building a BDD. The variables are ordered by the time of their creation. Since Haskell is free to change the order of evaluation, the variable order is not even predictable. This can have serious drawbacks, since the size of a BDD can vary greatly depending on the variable order. Figure 7 gives two BDDs for `(a1 && b1) || (a2 && b2) || (a3 && b3)` with different variable orderings. The dashed lines are false branches and the solid lines true branches.

However, in practice, trying to control variable ordering is a bit of a black art, and the problem in undecidable in general. But this situation is analogous to space allocation in Haskell, which is similarly out of the programmers hand, and has similar bad worst-case scenarios. One option, when variable order really needs to be controlled, is to use explicit sequencing via seq.

7.5 Applications in Verification

Why would this connection between Haskell and decision procedures be of interest to the verification community ? First, properties can be proven about Haskell programs. Free theorems from the parametricity of Haskell programs that model microprocessors may provide symmetry-like arguments for reducing the size of the state space.

Second, using Haskell allows models to be written in a strongly-typed language. Typechecking has its own benefits for a specification language, and now we are providing a link directly to verification tools for this language.

Third, Haskell works well as a meta-language for generating terms for input to the verification process. Can laziness in Haskell be exploited to avoid full generation of a term while a proof is in progress ? Laziness could be particularly important for defect-finding verification efforts.

8 Related Work

This work extends the brief description of linking BDDs with Haskell found in Launchbury, Lewis, and Cook [14].

Lava [3] is a Haskell-based hardware description language. They provide multiple interpretations of circuit descriptions for simulation, verification, and code generation. For verification, Lava interfaces to the propositional logic checker Prover [21], and two first-order logic theorem provers. The interface is file-based, breaking an expression into component subexpressions. Lava used reinterpretations of monads to create output for the different provers.

Individually decision procedures have been connected to other functional languages. For example SVC has been connected to Lisp. Voss uses BDDs for all boolean manipulations. And BDD packages such as Buddy [15] have been connected to ML and as a decision procedure in the HOL theorem prover [13]. Compared to these approaches, we use a generalized version of the Bool datatype through the class system to allow the packages to be used somewhat interchangeably. Furthermore, using Haskell we are able to provide this link in a pure functional language while preserving referential transparency.

9 Conclusion

It seems that our logical ducks swim quite well as abstract datatypes in Haskell. By generalizing the boolean and equality classes, it is possible to use the different decision procedures somewhat interchangeably. We have defined referentially transparent interfaces, allowing the underlying tools to do their work while the user simply sees the corresponding values. Having a tight connection between Haskell and the decision procedure allowed us to avoid space limitations in building the unreduced expression. The integration with Haskell also allowed us to leverage parametricity arguments in proofs.

10 Acknowledgements

The use of parametricity to redo Knuth's result was discovered in conjunction with John Matthews and Mircea Draghicescu. We thank Clark Barrett of Stanford for help with the Stanford Validity Checker. The authors are supported by Intel, U.S. Air Force Materiel Command (F19628-96-C-0161), NSF (EIA-98005542) and the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] VIS home page.
http://www-cad.eecs.berkeley.edu/~vis/.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, volume 1166 of *LNCS*, pages 187–201. Springer-Verlag, 1996.

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

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

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *DAC*, 1990.

[7] 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.

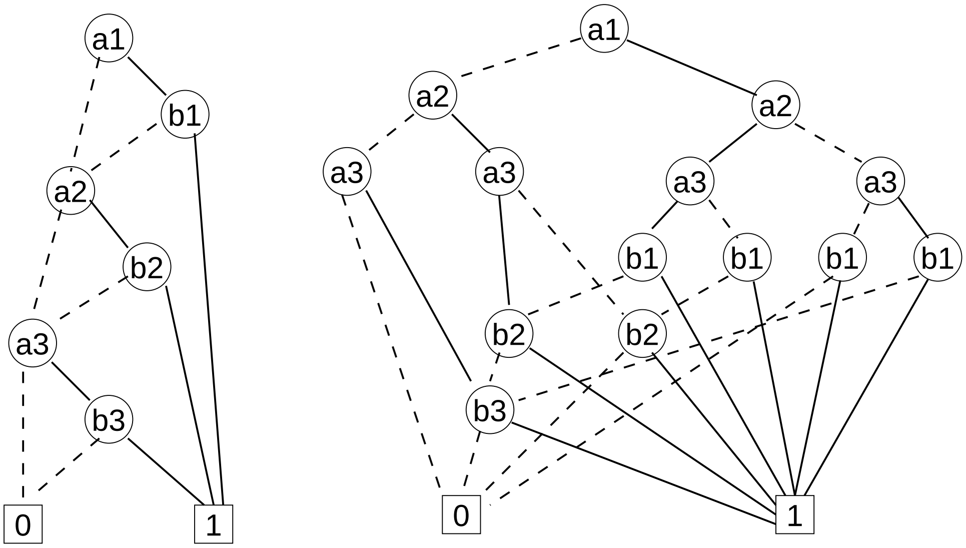

Figure 7: The formula (a1 && b1) || (a2 && b2) || (a3 && b3) with different variable orders (found in [5])

[8] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, pages 450–462, 1993.

[9] C. Elliot and P. Hudak. Functional reactive animation. In *ACM Int. Conf. on Functional Programming*, 1997.

[10] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.

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

[12] R. B. Jones, J. U. Skakkebæk, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 2–17. Springer-Verlag, 1998.

[13] K. Larsen and J. Lichtenberg. MuDDy. http://www.itu.dk/research/muddy/.

[14] J. Launchbury, J. Lewis, and B. Cook. On embedding a microarchitectural design language within Haskell. In *ACM Int. Conf. on Functional Programming*, 1999. To appear.

[15] J. Lind-Nielsen. BuDDy: Binary decision diagram package, release 1.6, 1998. Department of Information Technology, Technical University of Denmark.

[16] D. E. Long. bdd - a binary decision diagram (BDD) package. Man page.

[17] K. L. McMillan. Minimalist proof assistants. In *FMCAD*, volume 1522 of *LNCS*, page 1. Springer, 1998.

[18] D. Y. Park, J. U. Skakkebæk, M. P. Heimdahl, B. J. Czerny, , and D. L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *FMSP'98*, 1998.

[19] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In R. Mason, editor, *Information Processing 83*, Proceedings of the IFIP 9th World Computer Conference, 1983.

[20] C.-J. H. Seger. Voss - a formal hardware verification system: User's guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, December 1993.

[21] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In *FMCAD*, number 1522 in LNCS, pages 82–99, 1998.

[22] J. U. Skakkebæk, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV*, volume 1427 of *LNCS*, pages 98–109. Springer-Verlag, 1998.

[23] P. Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.