Observable Sharing for Functional Circuit Description*

Koen Claessen David Sands
{koen,dave}@cs.chalmers.se†

Abstract

Pure functional programming languages have been proposed as a vehicle to describe, simulate and manipulate circuit specifications. We propose an extension to Haskell to solve a standard problem when manipulating data types representing circuits in a lazy functional language. The problem is that circuits are finite graphs – but viewing them as an algebraic (lazy) datatype makes them indistinguishable from potentially infinite regular trees. However, implementations of Haskell do indeed represent cyclic structures by graphs. The problem is that the sharing of nodes that creates such cycles is not observable by any function which traverses such a structure. In this paper we propose an extension to call-by-need languages which makes graph sharing observable. The extension is based on non updatable reference cells and an equality test (sharing detection) on this type. We show that this simple and practical extension has well-behaved semantic properties, which means that many typical source-to-source program transformations, such as might be performed by a compiler, are still valid in the presence of this extension.

1 Introduction

In this paper we investigate a particular problem of embedding a hardware description language in a lazy functional language – in this case Haskell. The "embedded language" approach to domain-specific languages typically involves the designing a set of combinators (higher-order reusable programs) for an application area, and by constructing individual applications by combining and coordinating individual combinators. There are a number of advantages in developing an embedded language rather than building a language from scratch. One advantage is that the embedded language can inherit desirable language properties and tools that the host language already provides. Examples of these which are relevant here are a strong type system, expressive syntax, higher order functions, compilers and interpreters. Further, if the host language enjoys a formal semantics and rich reasoning principles, then these can also be inherited by the the embedded language to formally reason about the embedded program. See [Hud96] for examples of domain-specific languages embedded in Haskell.

In the case of hardware design the objects constructed are descriptions of circuits; by providing different interpretations of these objects one can, for example, simulate, test, model-check or compile circuits to a lower-level description. For this application (and other embedded description languages) we motivate an extension to Haskell with a feature which we call *observable sharing*, that allows us to detect and manipulate cycles in data-structures – a particularly useful feature when describing circuits containing feedback. Observable sharing is added to the language by providing immutable reference cells, together with a reference equality test. In the first part of the paper we present the problem and motivate the addition of observable sharing.

A problem with *observable sharing* that it is not a conservative extension of a pure functional language. It is a "side effect" – albeit in a limited form – for which the semantic implications are not immediately apparent. This means that the addition of such a feature risks the loss of many of the desirable semantic features of the host language. O'Donnell [O'D93] considered a form of observable sharing (Lisp-style pointer equality `eq`) in precisely the same context (i.e., the manipulation of hardware descriptions) and dismissed the idea thus:

> " This ⟨pointer equality predicate⟩ is a hack that breaks referential transparency, destroying much of the advantages of using a functional language in the first place."

But how much is actually "destroyed" by this construct? In the second part of this paper we show – for our more constrained version of pointer equality – that in practice almost nothing is lost.

We formally define the semantics of the language extensions and investigate their semantic implications. The semantics is an extension to a call-by-need abstract machine which faithfully reflects the amount of sharing in typical Haskell implementations.

Not all the laws of pure functional programming are sound in this extension. The classic law of beta-reduction for lazy functional programs, which we could represent as:

$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N = N[M/x] \quad (x \notin M)$$

does *not* hold in the theory. However, since this law could duplicate an arbitrary amount of computation (via

*An extended abstract of this article appears in the proceedings of ASIAN'99, LNCS

†Department of Computing Sciences, Chalmers University of Technology and Göteborg University, Sweden

the duplication of the sub-expression M, it has been proposed that this law is not appropriate for a language like Haskell [AFM+95], and that more restrictive laws should be adopted. Indeed most Haskell compilers (and most Haskell programmers?) do not apply such arbitrary transformations – for efficiency reasons they are careful not to change the amount of sharing (the internal graph structure) in programs. This is because all Haskell implemetations use a *call-by-need* parameter passing mechanism, whereby the argument to a function in a given call is evaluated at most once.

We develop the theory of operational equivalence for our language, and demonstrate that the extended language has a rich equational theory, containing, for example, all the laws of Ariola et al's call-by-*need* lambda calculus [AFM+95]. We also show that the semantics satisfies evaluation-order independence properties meaning that compiler optimisations such as strictness analysis will not change the semantics of programs.

The ideas in this paper are not only relevant to manipulating circuit descriptions. They appear useful for programming with other embedded description languages in Haskell. In the conclusion of the paper we mention two other possible applications.

2 Functional Hardware Description

We will deal with the description of synchronous hardware circuits in which the behaviour of a circuit and also its components can be modelled as *functions* from streams of inputs to streams of outputs.

The description is realised using an embedded language in the pure functional language Haskell. There are good motivations in literature for being able to use higher-order functions, polymorphism and laziness to describe hardware [She85, O'D96, CLM98, BCSS98]. In this paper however, we are concerned with some specific details related to the realisation of such an embedded language.

2.1 Describing Circuits

Viewing circuits as functions provides us with a way of embedding them in a functional language: as functions from in signals to out signals. This approach was taken as early as in the days of μFP [She85], and later modernised in systems like Hydra [O'D96] and Hawk [CLM98]. The following introduction to functional circuit description owes much to the description in [O'D93].

Here are some examples of primitive circuit components modelled as functions. We assume the existence of a datatype `Signal`, which represents an input, output or internal wire in a circuit.

```
inv   :: Signal -> Signal
latch :: Signal -> Signal
and   :: Signal -> Signal -> Signal
xor   :: Signal -> Signal -> Signal
```

We can put these components together in the normal way we compose functions; by abstraction, application, and local naming. Here is an example of a circuit consisting of an and-gate and an xor-gate. It takes in two inputs and has two outputs.

```
halfAdd :: Signal -> Signal -> (Signal, Signal)
halfAdd a b = (xor a b, and a b)
```

Here is an example of a more complicated circuit. We use local naming of results of subcomponents using a `let` expression.

```
fullAdd :: Signal -> Signal -> Signal
        -> (Signal, Signal)
fullAdd a b c =
  let  (s1, c1) = halfAdd a b
       (s2, c2) = halfAdd s1 c
   in  (s2, xor c1 c2)
```

Here is a third example of a circuit. It consists of an inverter and a latch, put together with a loop, also called *feedback*. The result is a circuit that toggles its output.

```
toggle :: Signal
toggle =
  let  output = inv (latch output)
   in  output
```

Note how we express the loop; by naming the wire and using it recursively.

2.2 Simulating Circuits

By interpreting the type `Signal` as streams of bits, and the primitive components as functions on these streams, we can run, or *simulate* circuit descriptions with concrete input.

Here is a possible instantiation, where we model streams by Haskell's lazy lists.

```
type Signal = [Bool] -- possibly infinite

inv    bs = map not bs
latch  bs = False : bs
and as bs = zipWith (&&) as bs
xor as bs = zipWith (/=) as bs
```

We can simulate a circuit by applying it to inputs. Here are the results of evaluating some of the circuits we have defined so far:

```
> halfAdd [False, True] [True, True]
[(True, False), (False, True)]

> fullAdd [False, True] [True, True] [True, True]
[(False, True), (True, True)]

> toggle
[True, False, True, False, True, ...
```

As parameters we provide lists or streams of inputs and as result we get a stream of outputs. Note that the toggle circuit does not take any parameter and results in an infinite stream of outputs. The ability to both specify and execute (and perform other operations) hardware as a functional program is a claimed strength of the approach.

2.3 Generating Netlists

Suppose we have described a circuit component-wise in this way, and perhaps tested its behaviour. Now we want to realize the circuit – for example for implementation on an FPGA. Or perhaps we want to use a theorem prover or model checker to prove properties about the description. In all of these cases, we want to know what components the circuit are, and how they are connected. Such a description is usually called a *netlist*. We can reach this goal by *symbolic evaluation*. This means that we supply variables as inputs to a circuit rather than concrete values, and construct an expression representing the circuit.

In order to do this, we have to reinterpret the `Signal` type and its operations. A first try might be along the following lines. A signal is either a variable name (a wire), or the result of a component which has been supplied with its input signals.

```
data Signal
  = Var String
  | Comp String [Signal]

inv b   = Comp "inv"   [b]
latch b = Comp "latch" [b]
and a b = Comp "and"   [a, b]
xor a b = Comp "xor"   [a, b]
```

Now, we can for example symbolically evaluate a half adder:

```
> halfAdd (Var "a") (Var "b")
(Comp "xor" [Var "a", Var "b"],
Comp "and" [Var "a", Var "b"])
```

And, similarly a full adder. But what happens when we try to look at the toggle circuit?

```
> toggle
Comp "inv" [Comp "latch" [Comp "inv" [Comp "latch" ...
```

Since the `Signal` datatype is essentially a tree, and the toggle circuit contains a cycle, the result is an infinite structure. This is of course not usable as a symbolic description in an implementation. We get an infinite data structure representing a finite circuit.

We encounter a similar problem when we provide inputs to the half adder that are not simple variables, but the result of another component, for example an xor gate.

```
> halfAdd (xor (Var "x") (Var "y")) (Var "b")
(Comp "xor" [Comp "xor" [Var "x", Var "y"], Var "b"],
Comp "and" [Comp "xor" [Var "x", Var "y"], Var "b"])
```

The desired description here is *one* xor gate, whose output is a wire which is used twice in the half adder. Instead, because our signals are trees, the whole component is copied because sharing of subtrees cannot be expressed in the datatype. We have basically hit a wall because we have used trees (algebraic data types) to represent circuits, where as physically, circuits have a richer graph-like structure.

2.4 A Previous solution: Explicit Tagging

One possible solution, proposed by O'Donnell [O'D93], is to give every use of component a unique tag, explicitly. The signal datatype is then still a tree, but when we then traverse that tree, we can keep track of what tags we have already encountered, and thus avoid cycles and detect sharing.

In order to do this, we have to change the signal datatype slightly by adding a *tag* to every use of a component. To make it easier to decorate every component in the circuit with a unique tag, an operator `(!)` is introduced that can create a new tag using an old tag and an integer.

```
data Signal = Var String
            | Comp Tag String [Signal]

type Tag = ...
(!) :: Tag -> Int -> Tag
```

Here are the definitions of the primitive components.

```
inv b   t = Comp t "inv"   [b]
latch b t = Comp t "latch" [b]
and a b t = Comp t "and"   [a, b]
xor a b t = Comp t "xor"   [a, b]
```

This is how the toggle circuit would look.

```
toggle :: Tag -> Signal
toggle t =
  let  wir = latch out (t!1)
       out = inv wir   (t!2)
   in  out
```

Though presented as "the first real solution to the problem of generating netlists from executable circuit specifications [...] in a functional language", it is awkward to use. A particular weakness of the abstraction is that it does not enforce that two components with the same tag are actually identical; there is nothing to stop the programmer from mistakenly introducing the same tag on different components.

2.5 Another Solution: the Monadic Approach

If explicit tagging is not the desired solution, why not let some underlying machinery take care of it? *Monads* are a standard approach for such problems (see e.g., [Wad92]). The monadic approach is taken in Lava [BCSS98]. A monad `M` is a data structure that can abstract from an underlying computation model. A monadic computation resulting in something of type `a` has type `M a`.

A very common monad is the *state monad*, which threads a changing piece of state through a computation. We can use such a state monad to generate fresh tags for the signal datatype. Here is a simple implementation of the monad. Readers not familiar with the monadic style of programming in Haskell may safely skim through to the next section.

```
type Tag = Int
type M a = Tag -> (a, Tag)

return :: a -> M a
return a = \t -> (a, t)
```

```
(>>=) :: M a -> (a -> M b) -> M b
m >>= k = \t ->
  let (a, t') = m t  in  k a t'
```

There are two basic operations; `return` inserts a value into the computation type, and `>>=`, also called *bind*, sequences two computations, where the second can depend on the result of the first. Introducing a monad implies that the types of the primitive components and circuit descriptions become *monadic*, that is, their result is wrapped up in the monad type `M`.

```
inv   :: Signal -> M Signal
latch :: Signal -> M Signal
and   :: Signal -> Signal -> M Signal
xor   :: Signal -> Signal -> M Signal

inv b   = comp "inv"   [b]
latch b = comp "latch" [b]
and a b = comp "and"   [a, b]
xor a b = comp "xor"   [a, b]

comp name args =
  \t -> (Comp t name args, t+1)
```

A big disadvantage of this approach is not only that we must change of types, but also that the syntax must change. We can no longer use normal function abstraction and local naming anymore, we have to express this using the monadic bind `(>>=)`. This means that, just as in the previous solution, we have to change the definitions of the circuits. Here is what the half adder looks like in monadic style.

```
halfAdd :: Signal -> Signal -> M (Signal, Signal)
halfAdd a b =
  xor a b >>= \sum ->
  and a b >>= \carry ->
  return (sum, carry)
```

Another unwanted consequence of not being able to use local naming, is that we cannot use recursion anymore to express feedback in circuits. We have to define an explicit *monadic fixpoint* combinator to express loops.

```
loop :: (a -> M a) -> M a
loop f = \t -> let (a, t') = f a t in (a, t')
```

The definition of the toggle circuit serves as an example of how to introduce loops in this style:

```
toggle :: M Signal
toggle = loop (\out ->
  latch out >>= \wir ->
  inv wir   >>= \out' ->
  return out'
  )
```

All this turns out to be very inconvenient for the programmer. Furthermore, the monadic approach forces us to specify the order in which we create components in a circuit. This sequentiality is unnatural when specifying the components of circuits, which are more naturally thought of as executing in parallel. [1]

What we are looking for is a solution that does *not* require a change in the natural circuit description style of using local naming and recursion, but allows us to detect sharing and loops in a description from *within* the language.

3 Proposed Solution

The core of the problem is: a description of a circuit is basically a graph, but we cannot observe the sharing of the nodes from within the program.

The solution we propose is to make the graph structure of a program *observable*, by adding a new language construct. This can be done in several ways. In the beginning of this section we explain and motivate the particular constructs we chose to enrich the language. At the end, we will discuss and compare our choice with other possible solutions.

3.1 Objects with Identity

The idea is that we want the weakest extension that is still powerful enough to observe if two given objects have actually previously been created as one and the same object. The reason for wanting as weak an extension as possible is that we want to retain as many semantic properties from the original language as possible. This is not just for the benefit of the programmer – it is important because compilers make use of semantic properties of programs to perform program transformations, and because we do not want to write our own compiler to implement this extension.

Since we know in advance what kind of objects we will compare in this way, we choose to be explicit about this at *creation* time of the object that we might end up comparing. In fact, you can view the objects as *non-updatable references*. We can create them, compare them for equality, and dereference them.

Here is the interface we provide to the references. We give a formal description of the semantics in section 4.

```
type Ref a = ...

ref   :: a -> Ref a
deref :: Ref a -> a
(<=>) :: Ref a -> Ref a -> Bool
```

The following two examples show how we can use the new constructs to detect sharing. In the first example, we create one reference, and compare it with itself, which yields `True`.

```
> let  x = undefined in  let  r = ref x in  r <=> r
True
```

In the second example, we create two *different* references to the same variable, and so the comparison yields `False`.

[1] Also, a possible problem that we noticed in practice when modelling larger circuits (e.g. multipliers) is that the linearisation of component creation seemed (in our programs) to have a disastrous effect on run-time memory behaviour of the Haskell program. Although these space-leaks could probably be fixed within the monadic program, it is gratifying to note that these problems evaporated when we used the solution described in the next section.

```
> let  x = undefined in  ref x <=> ref x
False
```

Thus, we have made a *non conservative extension* to the language; previously it was not possible to distinguish between a shared expression and two different instances of the same expression. We call the extension *observable sharing*.

In appendix A, we will present a possible implementation of references as presented here, suitable for most Haskell compilers. The extension makes use of a standard "unsafe" (side-effecting) operation that is included with most Haskell implementations, but is not part of the language proper.

3.2 Back to Circuits

How can we use this extension to help us to symbolically evaluate circuits? Let us take a look at the following two circuits.

```
circ1 =
  let  output = latch output
  in  output

circ2 =
  let  output = latch (latch output)
  in  output
```

In Haskell's denotational semantics, these two circuits would be identified, since `circ2` is just a recursive unfolding of `circ2`. But we would like these descriptions to represent different circuits; `circ1` has one latch and a loop, where as `circ2` has two latches and a loop. If the signal type includes a reference, we could compare the identities of the latch components and conclude that in `circ1` all latches are identical, where as in `circ2` we have two *different* latches.

3.3 A New Signal Type

We can now modify the signal datatype in such a way that the creation of identities happens transparently to the programmer. We play a similar trick as with the tagging, but instead we use references.

```
data Signal = Var String
            | Comp (Ref (String, [Signal]))

inv b   = comp "inv"   [b]
latch b = comp "latch" [b]
and a b = comp "and"   [a, b]
xor a b = comp "xor"   [a, b]

comp name args =  Comp (ref (name, args))
```

3.4 Subtleties of Sharing

Using the references we gain the ability to express different degrees of sharing of subcomponents. The subtlety is that the programmer must have a clear understanding of the sharing properties of the semantics. Here are two different definitions of the `toggle` function.

```
toggle1 =
  let  output = inv (latch output) in  output

toggle2 () =
  let  output = inv (latch output) in  output
```

Without references `toggle1` would be indistinguishable from `toggle2 ()`. But the circuit `toggle1` is a *constant* circuit. This means that it will only occur at most *once* as a component; every use of `toggle1` refers to the same *wire*. Using `toggle2 ()` (applied to the dummy argument) creates a *new* component every time you apply it.

Both views are relevant in circuit descriptions – but the programmer needs to be aware of such differences. To do this the programmer must understand the basics of the *call-by-need* execution mechanism. In `toggle1` the output is executed (constructed) exactly once, the first time that it is needed. In the output is constructed once *for every application of the `toggle2` function*. Most reasonably experienced Haskell programmers are already aware of this difference; with observable sharing it becomes essential knowledge.

3.5 Other Possible Solutions

We present two other possible solutions, both are more or less well known extensions to functional programming languages.

Pointer Equality. The language is extended with an operator `(>=<) :: a -> a -> Bool` that investigates if two expressions are *pointer equal*, that is, they refer to the same bindings. There are a number of different semantics we can give to the extension; they involve how much evaluation of the arguments is done before comparing the pointers.

In our extension, we basically provide pointer equality in a more controlled way; you can only perform it on references, not on expressions of any type. This means we can implement our references using a certain kind of pointer equality:

```
type Ref a = a
ref a      = a
deref a    = a
r1 <=> r2  = r1 >=< r2
```

The other way around is not possible however, which shows that our extension is weaker. This corresponds to our goal, as explained at the beginning of this section.

Gensym. The language is extended with a new type `Sym` of abstract symbols with equality, and an operator that generates fresh such symbols, `gensym`.

```
type Sym = ...
gensym :: (Sym -> a) -> a
(==)   :: Sym -> Sym -> a
```

The symbol type and its operators can be implemented as a reference to the unit type:

```
type Sym = Ref ()
gensym f = f (ref ())
s1 == s2 = s1 <=> s2
```

The other way around is also possible; we implement a reference as a pair of a symbol and the value that the reference points to.

```
type Ref a           = (Sym, a)
ref a                = gensym (\s -> (s, a))
deref (_, a)         = a
(s1, _) <=> (s2, _) = s1 == s2
```

Since the approaches can be implemented in terms of each other, it is not clear which one is preferable. With the reference approach however, by get an important law by *definition*, which is:

$$\texttt{r1 <=> r2} = \mathit{True} \;\Rightarrow\; \texttt{deref r1} = \texttt{deref r2}$$

In the gensym approach, this becomes a proof obligation as part of the implementation.

4 The Semantic Theory

In this section we formally define the operational semantics of observable sharing, and study the induced notion of operational equivalence.

4.1 Language

For the technical development we work with a de-sugared core language based on an untyped lambda calculus with recursive lets, structured data, and case expressions.

The language of terms, Λ_{ref} is given by the following grammar:

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid M\,x \mid c\,\vec{x} \\ & \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \\ & \mid \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \\ & \mid M\,;N \\ & \mid \mathsf{ref}\ x \mid \mathsf{deref}\ M \mid M \rightleftharpoons N \end{aligned}$$

where the term $M\,;N$ is *strict sequential composition*, corresponding to Haskell's `'seq'` operator. We include it here since it is useful for describing the compiler optimisations which follow *strictness analysis*. Note that we work with a restricted syntax in which the arguments in function applications and the arguments to constructors are always variables. It is trivial to translate programs into this syntax by the introduction of let bindings for all non-variable arguments. Such syntactic restrictions are common in compilation schemes. In this particular case we follow its use in the core language of the Glasgow Haskell compiler, *e.g.*, [PJPS96, PJS98], and in [Lau93, Ses97]. Indeed, our language is essentially an untyped core of the intermediate language of the Glasgow Haskell Compiler, extended with immutable references and equality testing on references.

Throughout, x, y, z etc. will range over variables, and c will range over *constructors*. The set of *values*, $\mathrm{Val} \subseteq \Lambda_{\mathsf{ref}}$, ranged over by V and W are the constructor-expressions $c\,\vec{x}$ and the lambda-expressions $\lambda x.M$. The constructors are assumed to include the nullary constructors true and false. Constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1 \cdots x_n$. We will write $\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N$ as a shorthand for

$$\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N$$

where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (i.e., all lets are potentially recursive). Similarly we write $\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}$ for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\,\vec{x}_1 \to N_1 \mid \cdots \mid c_m\,\vec{x}_m \to N_m\}.$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i \to N_i\}$.

The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

4.2 The Abstract Machine

The semantics for the standard part of the language presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [Ses97]. Transitions in this machine are defined over configurations consisting of (i) a *heap*, containing a set of bindings, (ii) the expression currently being evaluated, and (iii) a *stack*, representing the actions that will be performed on the result of the current expression.

There are a number of possible ways to represent references in such a machine. One straightforward possibility is to use a global reference-environment, in which evaluation of the ref operation creates a fresh reference to its argument. The representation we present here is essentially equivalent, but syntactically more economical. Instead of reference environment, references are represented by a "hidden" constructor (i.e. a constructor which is not part of Λ_{ref}), which we denote by $\underline{\mathsf{ref}}$.

Let $\Lambda_{\underline{\mathsf{ref}}} \stackrel{\text{def}}{=} \Lambda_{\mathsf{ref}} \cup \{\underline{\mathsf{ref}}\,x \mid x \in \mathit{Var}\}$, and $\mathrm{Val}_{\underline{\mathsf{ref}}} \stackrel{\text{def}}{=} \mathrm{Val} \cup \{\underline{\mathsf{ref}}\,x \mid x \in \mathit{Var}\}$.

We write $\langle\, \Gamma,\ M,\ S \,\rangle$ for the abstract machine configuration with heap Γ, expression $M \in \Lambda_{\underline{\mathsf{ref}}}$, and stack S. A *heap* is a set of bindings from variables to terms of $\Lambda_{\underline{\mathsf{ref}}}$; we denote the empty heap by $\emptyset$, and the addition of a group of bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$.

A stack is a list of stack elements. The stack written $b : S$ will denote the a stack S with b pushed on the top. The empty stack is denoted by ϵ, and the concatenation of two stacks S and T by ST (where S is on top of T). Stack elements are either:

- a variable x, representing the argument to a function,
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap,
- a group of case-alternatives, one of which will be chosen according to the outcome of the current evaluation,
- the second argument of a strict sequence, denoted $(;M)$, or
- a pending reference equality-test of the form $(\rightleftharpoons M)$, or $(\underline{\mathsf{ref}}\,x \rightleftharpoons)$,
- a dereference deref, indicating that the location which is produced by the current computation should be dereferenced.

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. Since we cannot have more than one binding occurrence of a variable, a configuration is deemed *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle\, \Gamma,\ M,\ S \,\rangle$ to be closed, any free variables in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$.

For sets of variables P and Q we will write $P \natural Q$ to mean that P and Q are disjoint, *i.e.*, $P \cap Q = \emptyset$. The free variables of a term M will be denoted $\mathsf{FV}(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}(\vec{M})$.

The abstract machine semantics is presented in figure 4.2; we implicitly restrict the definition to well-formed configurations.

The first group of rules are the standard call-by-need rules. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this evaluation is finished, and we may update the heap with the new binding for x.

Rules (*Unwind*) and (*Subst*) concern function application: rule (*Unwind*) pushes an argument onto the stack while the function is being evaluated; once a lambda expression has been obtained, rule (*Subst*) retrieves the argument from the stack and substitutes it into the body of that lambda expression.

Rules (*Case*) and (*Branch*) govern the evaluation of case expressions. Rule (*Case*) initiates evaluation of the case expression, with the case alternatives pushed onto the stack. Rule (*Branch*) uses the result of this evaluation to choose one of the branches of the case, performing substitution of the constructor's arguments for the branch's pattern variables.

Rule (*Letrec*) adds a set of bindings to the heap. The side condition ensures that no inadvertent name capture occurs, and can always be satisfied by a local α-conversion.

Rules (*Seq1*) and (*Seq2*) implement the Haskell-style strict sequential evaluation, by first evaluating the left argument, then discarding the result in favour of the right argument.

The second collection of rules concern the observable sharing. Rule (*RefEq*) first forces the evaluation of the left argument, and (*Ref1*) switches evaluation to the right argument; once both have been evaluated to ref constructors, variable-equality is used to implement the pointer-equality test.

4.3 Convergence

Two terms will be considered equal if they exhibit the same behaviours when used in any program context. The behaviour that we use as our test of equivalence is simply termination. Termination behaviour is formalised by a convergence predicate:

Definition 4.1 (Convergence) *A closed configuration* $\langle\, \Gamma,\ M,\ S \,\rangle$ converges, *written* $\langle\, \Gamma,\ M,\ S \,\rangle\Downarrow$, *if there exits heap* Δ *and value* V *such that*

$$\langle\, \Gamma,\ M,\ S \,\rangle \to^* \langle\, \Delta,\ V,\ \epsilon \,\rangle.$$

We will also write $M\Downarrow$, identifying closed M with the initial configuration $\langle\, \emptyset,\ M,\ \epsilon \,\rangle$.

Closed configurations which do not converge are of four types: they either (i) reduce indefinitely, or get stuck because of (ii) a type error, (iii) a case expression with an incomplete set of alternatives, or (iv) a *black-hole* (a self-dependent expression as in $\mathsf{let}\ x = x\ \mathsf{in}\ x$). All non-converging closed configurations will be semantically identified.

4.4 Approximation and Equivalence

To define equivalence we take the standard approach of defining a contextual preorder which says that M approximates (is less than) N in the ordering if whenever a program containing M terminates, then replacing M by N will not worsen the termination behaviour. Let $\mathbb{C}$, $\mathbb{D}$ range over *contexts* – terms containing zero or more occurrences of a *hole*, $[\cdot]$ in the place where an arbitrary subterm might occur. Let $\mathbb{C}[M]$ denote the result of filling all the holes in $\mathbb{C}$ with the term M, possibly causing free variables in M to become bound.

Definition 4.2 (Operational Approximation) *We say that* M operationally approximates N, *written* $M \lesssim N$, *if for all* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are closed,*

$$\mathbb{C}[M]\Downarrow \implies \mathbb{C}[N]\Downarrow.$$

We say that M and N are *operationally equivalent*, written $M \cong N$, when $M \lesssim N$ and $N \lesssim M$. Note that equivalence is a non-trivial equivalence relation; In Figures 2 and 3 we present a collection of basic laws of equivalence. As usual with a "semantic" definition of equivalence, it is not a recursively enumerable relation.

Remark: The fact that the reference constructor $\underline{\mathsf{ref}}$ is abstract (not available directly in the language) is crucial to the variable-inlining properties. For example a (derivable) law like

$$\mathsf{let}\ \{x = z\}\ \mathsf{in}\ N \cong N[z/x]$$

would fail if terms could contain $\underline{\mathsf{ref}}$. This failure could be disasterous in some implementations, because in effect a configuration-level analogy of this law is applied by some garbage collectors.

4.5 Laws for Reduction Contexts

A *reduction context* $\mathbb{R}$ is a context in which the hole is the target of evaluation; in other words, evaluation cannot proceed until the hole is filled. We use the following simple grammar for reduction contexts; more complex definitions are also possible, but are not needed here.

$$\begin{array}{l}\mathbb{R} ::= [\cdot] \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{R} \mid \mathbb{R}\,x \mid \mathsf{case}\ \mathbb{R}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to M_i\} \\ \quad \mid\ \mathbb{R}\,;M \mid \mathbb{R} \rightleftharpoons M \mid \mathsf{deref}\ \mathbb{R}\end{array}$$

Figure 4 contains a collection of laws relating to reduction contexts.

$$\langle\, \Gamma\{x = M\},\ x,\ S\,\rangle \to \langle\, \Gamma,\ M,\ \#x : S\,\rangle \qquad (Lookup)$$

$$\langle\, \Gamma,\ V,\ \#x : S\,\rangle \to \langle\, \Gamma\{x = V\},\ V,\ S\,\rangle \qquad (Update)$$

$$\langle\, \Gamma,\ M\,x,\ S\,\rangle \to \langle\, \Gamma,\ M,\ x : S\,\rangle \qquad (Unwind)$$

$$\langle\, \Gamma,\ \lambda x.M,\ y : S\,\rangle \to \langle\, \Gamma,\ M[y/x],\ S\,\rangle \qquad (Subst)$$

$$\langle\, \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ alts,\ S\,\rangle \to \langle\, \Gamma,\ M,\ alts : S\,\rangle \qquad (Case)$$

$$\langle\, \Gamma,\ c_j\,\vec{y},\ \{c_i\,\vec{x}_i \to N_i\} : S\,\rangle \to \langle\, \Gamma,\ N_j[\vec{y}/\vec{x}_j],\ S\,\rangle \qquad (Branch)$$

$$\langle\, \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\ S\,\rangle \to \langle\, \Gamma\{\vec{x} = \vec{M}\},\ N,\ S\,\rangle \quad \vec{x} \notin \mathrm{dom}(\Gamma, S) \qquad (Letrec)$$

$$\langle\, \Gamma,\ M\,;N,\ S\,\rangle \to \langle\, \Gamma,\ M,\ (;N) : S\,\rangle \qquad (Seq1)$$

$$\langle\, \Gamma,\ V,\ (;N) : S\,\rangle \to \langle\, \Gamma,\ N,\ S\,\rangle \qquad (Seq2)$$

$$\langle\, \Gamma,\ \mathsf{ref}\ M,\ S\,\rangle \to \langle\, \Gamma\{x = M\},\ \underline{\mathsf{ref}}\,x,\ S\,\rangle \quad x \notin \mathrm{dom}(\Gamma, S) \qquad (Ref)$$

$$\langle\, \Gamma,\ \mathsf{deref}\ M,\ S\,\rangle \to \langle\, \Gamma,\ M,\ \mathsf{deref}\ : S\,\rangle \qquad (Deref1)$$

$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,x,\ \mathsf{deref}\ : S\,\rangle \to \langle\, \Gamma,\ x,\ S\,\rangle \qquad (Deref2)$$

$$\langle\, \Gamma,\ M \rightleftharpoons N,\ S\,\rangle \to \langle\, \Gamma,\ M,\ (\rightleftharpoons N) : S\,\rangle \qquad (RefEq)$$

$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,x,\ (\rightleftharpoons N) : S\,\rangle \to \langle\, \Gamma,\ N,\ (\underline{\mathsf{ref}}\,x \rightleftharpoons) : S\,\rangle \qquad (Ref1)$$

$$\langle\, \Gamma,\ \underline{\mathsf{ref}}\,y,\ (\underline{\mathsf{ref}}\,x \rightleftharpoons) : S\,\rangle \to \langle\, \Gamma,\ b,\ S\,\rangle \quad b = \begin{cases} \mathsf{true} & \text{if } x = y \\ \mathsf{false} & \text{otherwise} \end{cases} \qquad (Ref2)$$

Figure 1: Abstract machine semantics

4.6 Laws for Strictness

In Figure 5 we present a collection of laws for refs, and strictness. The term Ω denote any closed term which does not converge. For example, the "black-hole" term, $\mathsf{let}\ x = x\ \mathsf{in}\ x$, would suffice as a definition for Ω. Ω is the bottom element of the operational approximation ordering. The last of this collection of laws represents the transformation induced by *strictness analysis*. Operationally, a term M is strict in a free variable x if it is equivalent to Ω whenever x is bound to Ω. This corresponds to the usual denotational definition of strictness for the function $\lambda x.M$. The rule expresses that if M is strict in x then x can safely be evaluated in advance. This repressents the typical compiler optimisation that follows after performing strictness analysis. We sketch the proof of this property in the next section.

4.7 Proof Techniques for Equivalence

We have presented a collection of laws for approximation and equivalence – but how are they established? The definition of operational equivalence suffers from the standard problem: to prove that two terms are related requires one to examine their behaviour in *all* contexts. For this reason, it is common to seek to prove a *context lemma* [Mil77] for an operational semantics: one tries to show that to prove M operationally approximates N, one only need compare their immediate behaviour. The following context lemma simplifies the proof of many laws:

Lemma 4.1 (Context Lemma) *For all terms M and N, $M \lesssim N$ if and only if for all Γ, S and substitutions σ,*

$$\langle\, \Gamma,\ M\sigma,\ S\,\rangle\Downarrow \implies \langle\, \Gamma,\ N\sigma,\ S\,\rangle\Downarrow$$

It says that we need only consider configuration contexts of the form $\langle\, \Gamma,\ [\cdot],\ S\,\rangle$ where the hole $[\cdot]$ appears only once. The substitution σ from variables to variables is necessary here, but all the laws are closed under such substitutions, so there is no noticeable proof burden.

The proof of the context lemma follows the same lines as the corresponding proof for the *improvement theory* for call-by-need [MS99], and it involves uniform computation arguments which are similar to the proofs of related properties for call-by-value languages with state [MT91].

Here we give a flavour of such proofs by stating a few key properties, and outline the proof of the inference rule which says that if a term strict in a given variable, then the it is safe to evaluate the variable in advance.

It is useful and meaningful to allow computation over open configurations; this is a handy way to express certain properties of computations.

Proposition 4.2

1. *(Open Computation) If $\langle\, \Gamma,\ M,\ S\,\rangle \to^* \langle\, \Gamma',\ N,\ S'\,\rangle$ then $\langle\, \Gamma\Delta,\ M,\ ST\,\rangle \to^* \langle\, \Gamma'\Delta,\ N,\ S'T\,\rangle$ for any bindings Δ and stack T such that the corresponding configurations are well-formed.*

2. *(Subcomputation) $\langle\, \Gamma,\ M,\ S\,\rangle\Downarrow \Leftrightarrow$ $\exists \Delta, V.\, \langle\, \Gamma,\ M,\ \epsilon\,\rangle \to^* \langle\, \Delta,\ V,\ \epsilon\,\rangle\ \&\ \langle\, \Delta,\ V,\ S\,\rangle\Downarrow$*

3. *(Value Stability) If $\langle\, \Gamma\{x = V\},\ M,\ S\,\rangle \to^* \langle\, \Delta,\ W,\ \epsilon\,\rangle$ then $\{x = V\} \subseteq \Delta$*

4. *(Reordering) If $\langle\, \Gamma,\ M,\ \epsilon\,\rangle \to^* \langle\, \Delta,\ V,\ \epsilon\,\rangle$ and $\langle\, \Delta,\ N,\ \epsilon\,\rangle \to^* \langle\, \Delta',\ V',\ \epsilon\,\rangle$ then there exists some Γ' such that $\langle\, \Gamma,\ N,\ \epsilon\,\rangle \to^* \langle\, \Gamma',\ V',\ \epsilon\,\rangle$ and $\langle\, \Gamma',\ M,\ \epsilon\,\rangle \to^* \langle\, \Delta',\ V,\ \epsilon\,\rangle$.*

In the satement of all laws, we follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables.

$$(\lambda x.M)\, y \cong M[y/x]$$
$$\mathsf{case}\ c_j\ \vec{y}\ \mathsf{of}\ \{c_i\ \vec{x}_i \to M_i\} \cong M_j[\vec{y}/\vec{x}_j]$$
$$\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[V]\}\ \mathsf{in}\ \mathbb{C}[V]$$
$$\mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z]$$
$$\mathsf{let}\ \{x = z, \vec{y} = \vec{M}\}\ \mathsf{in}\ N \cong \mathsf{let}\ \{x = z, \vec{y} = \vec{M}[z/x]\}\ \mathsf{in}\ N[z/x]$$

Figure 2: Beta laws for call-by-need.

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \cong N, \quad \text{if } \vec{x} \not\downarrow \mathsf{FV}(N)$$
$$\mathsf{let}\ \{\vec{x} = \vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{y} = \vec{M}\}\ \mathsf{in}\ N \cong \mathsf{let}\ \{\vec{x} = \vec{L}, \vec{y} = \vec{M}\}\ \mathsf{in}\ N$$
$$\mathsf{let}\ \{x = \mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \cong \mathsf{let}\ \{x = \mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N, \vec{y} = \vec{L}\}\ \mathsf{in}\ N'$$
$$\mathbb{C}[\mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ M] \cong \mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[M]$$

Figure 3: Laws for dealing with lets.

$$\mathbb{R}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \cong \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{R}[N_i]\}$$
$$\mathbb{R}[\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N] \cong \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{R}[N]$$
$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ \mathbb{R}[x] \cong \mathbb{R}[M], \quad \text{if } x \notin \mathsf{FV}(M, \mathbb{R})$$

Figure 4: Laws for Reduction Contexts.

$$M \rightleftharpoons N \cong N \rightleftharpoons M$$
$$x \rightleftharpoons x \cong \mathsf{deref}\ x\,;\mathsf{true}$$
$$\mathsf{ref}\ M \rightleftharpoons N \cong \mathsf{deref}\ N\,;\mathsf{false}$$
$$\mathsf{ref}\ M\,; N \cong N$$
$$V\,; N \cong N \quad M\,; N \precsim N \quad M\,; M \precsim M$$
$$L\,;(M\,; N) \cong (L\,; M)\,; N$$
$$L\,; M\,; N \cong M\,; L\,; N$$
$$\Omega \precsim M$$

$$\frac{\mathsf{let}\ \{x = \Omega\}\ \mathsf{in}\ M \cong \Omega}{M \precsim x\,; M}$$

Figure 5: Laws for Refs, Ω and Strictness.

The properties are established in a reasonably straightforward way by induction on the length of the computations. The last property of the list, reordering, can be established along the lines of Theorem 3.5.1 of [SPJ97]. Note that the reordering property would not hold if we had updatable references.

Let us illustrate the use of the context lemma and some of the properties above in a sketch proof of the strictness inference rule from figure 5.

Proposition 4.3 *If* $\mathsf{let}\ \{x = \Omega\}\ \mathsf{in}\ M \cong \Omega$ *then* $M \precsim x\,; M$

Proof. Under the assumption we will show that $M \precsim x\,; M$. By the context lemma, it is sufficient to show, for arbitrary Γ and S ($x \in \mathrm{dom}\,\Gamma, S$), that if $\langle \Gamma,\ M,\ S\rangle\Downarrow$ then $\langle \Gamma,\ x\,; M,\ S\rangle\Downarrow$. Assume $\langle \Gamma,\ M,\ S\rangle\Downarrow$. By (Subcomputation) and (Uniform Computation) we know that

$$\langle \Gamma,\ M,\ \epsilon\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle \tag{1}$$
$$\langle \Gamma,\ M,\ S\rangle \to^* \langle \Delta,\ V,\ S\rangle\Downarrow \tag{2}$$

Now, by the assumption that $\mathsf{let}\ \{x = \Omega\}\ \mathsf{in}\ M \cong \Omega$ we argue that there must be an intermediate state:

$$\langle \Gamma,\ M,\ \epsilon\rangle \to^* \langle \Gamma',\ x,\ T\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle$$

Applying the subcomputation argument again, we can see that $\langle \Gamma',\ x,\ \epsilon\rangle \to^* \langle \Delta'\{x = W\},\ W,\ \epsilon\rangle$ for some Δ'. By value stability we know that $\{x = W\} \subseteq \Delta$, so

$$\langle \Delta,\ x,\ \epsilon\rangle \to^2 \langle \Delta,\ W,\ \epsilon\rangle \tag{3}$$

Thus by (Reordering) (1) and (3), we know that

$$\langle \Gamma,\ x,\ \epsilon\rangle \to^* \langle \Gamma'',\ W,\ \epsilon\rangle \tag{4}$$
$$\langle \Gamma'',\ M,\ \epsilon\rangle \to^* \langle \Delta,\ V,\ \epsilon\rangle \tag{5}$$

And thus, by open extension we can construct the following computation sequence:

$$\begin{aligned}\langle \Gamma,\ x\,; M,\ S\rangle &\to \langle \Gamma,\ x,\ (\,;M) : S\rangle \\ &\to \langle \Gamma'',\ W,\ (\,;M) : S\rangle && \text{(by 4)} \\ &\to \langle \Gamma'',\ M,\ S\rangle \\ &\to \langle \Delta,\ V,\ S\rangle\Downarrow && \text{(by 5, 2)}\end{aligned}$$

□

4.8 Relation to Other Calculi

Many authors have considered the semantics of functional languages extended with with various forms of state. The approach pioneered by Felleisen et al (e.g. [FH92]) has been to study term-based reduction-calculi. The advantage of this approach is that it builds on the idea of a core calculus of equivalences (generated by a confluent rewriting relation on terms) which is conservatively extended with each additional language feature. The price paid for this modularity is that the theory of equality is rather limited. The approach we have taken – studying operational equivalence – is exemplified by Mason and Talcott's work on call-by-value lambda

calculi and state [MT91]. An advantage of the operational-equivalence approach is that it is a much richer theory, in which induction principles may be derived that are inexpressible in reduction calculi.

A reduction-calculi approach to call-by-need was introduced in [AFM+95]. An operational theory subsuming this calculus, the call-by-need *improvement theory*, was introduced by Moran and Sands [MS99]. In improvement theory, the operational equivalences allow the observation of the number of reduction steps to convergence. This makes sharing observable indirectly. The approach in this paper is based closely on the development of [MS99]. Interestingly all the laws which do not involve refs are also "cost equivalences within a constant factor" in the improvement theory. We have not been able to find a cost equivalence which is not an equivalence in Λ_{ref}, and it would be interesting and useful if it were possible to prove that equivalence in Λ_{ref} was an extension of cost equivalence.

There is also some work on the theory of side-effects to non-strict languages. Odersky [Ode94] considered a minimal extension to the pure lambda calculus with binding constructs for local names, and with equality of names as their only primitive. (Pitts and Stark considered a similar extension for call-by-value [PS93] lambda calculus.) Because of the call-by-name operational model underlying this work, it is not directly relevant to the applications we have in mind, and the operational theory is somewhat simpler to develop. More relevant is the recent work of Ariola and Sabry [AS98], who consider the call-by-need lambda calculus extended with mutable state. Had we taken a reduction-calculus approach, rather than developing operational equivalence, we could have cut down their language and reduction theory. It would not, however, have been possible to treat e.g. strictness properties in such a reduction-theory. Their work could be very useful to prove the correctness of an implementation of our language.

We have only scratched the surface of the existing theory. Induction principles would be useful – and seem straightforward to adapt from [MS99]. For techniques more specific to the subtleties of references, work on parametricity properties of local names e.g., [Pit96], might also be adaptable to the current setting.

5 Conclusions

We have motivated a small extension to Haskell which provides a practical solution to a common problem when manipulating data structures representing circuits. The feature is likely to be useful for other embedded description languages, and we briefly consider two such applications below.

The extension we propose is small, and turns out to be easy to add to existing Haskell compilers/interpreters in the form of an abstract data-type (a module with hidden data constructors). In fact similar functionality is already hidden away in the nonstandard libraries of many implementations.[2] A simple implementation using the Hugs-GHC library extensions is given in the appendix. The `hbc` compiler contains a module with essentially the same signature and functionality (plus a reference-update operation) in the `UnsafeDirty` library.

[2] www.haskell.org/implementations/

An important contribution of our work is to show that, in the absence of an update operation, these features are neither "unsafe" nor "dirty"! We have presented a precise operational semantics for this extension, and investigated laws of operational approximation. We have shown that the extended language has a rich equational theory, which means that the semantics is robust with respect to program transformations which respect sharing properties. For example, we have shown that standard compiler transformations which use strictness analysis to turn call-by-need in to call-by-value are still sound in this extension.

5.1 Other Applications of Observable Sharing

Here we mention two other potential applications of observable sharing to other embedded description languages.

Grammars and Parsers

A popular example of an embedded description language is a language for *grammars*. This is usually realised as a library with parsing combinators [Hut92] for building more complex parsers from more basic ones. Parsing combinators are higher-order functions corresponding to grammatical constructs such as sequencing, alternation and repetition. Such parsers are simple to construct and easy to understand, since their form follows the grammar.

However, not all grammars are immediately executable when interpreted as parsers in this style. For example, the following grammar is *left recursive*:

$$T ::= T + T \mid n$$

Executing the grammar as a parser as it stands will result in an infinite loop, because a T will first parse a T, and so on. This is a known problem with parser combinators (and other top-down parsing methods), and usually one must transform a grammar so that it is not left recursive.

An alternative idea (due to Magnus Carlsson) is the following: if every parser had its own identity, and knew the identity of all its parents, it could detect the fact that it was used left recursively (if it occurs as one of its own parents). In that case, it could avoid infinite looping. We implemented a prototype parser based on this idea, using references to parsers.

Decision Trees

Another possible application area is decision trees. A decision tree is a binary tree with yes/no questions at its nodes and results at the leaves. We can obtain a result by walking down the tree and answering each question by yes or no, taking the left or right tree accordingly.

We can make such a decision tree more efficient by removing those nodes for which both subtrees are the same. In that case, the given answer does not matter. We can implement this using references, so that the comparison of the two subtrees only takes constant time.

```
type DT = Ref DecTree

data DecTree = Leaf Result
             | Node Quest DT DT
```

We introduce a special node creation function that checks if the two subtrees are identical.

```
node :: Quest -> DT -> DT -> DT
node quest yes no
  | yes <=> no = yes
  | otherwise  = ref (Node quest yes no)
```

One problem is that two subtrees might be equal, but not identical. This happens when we create two equal trees separately. We can leave it up to the programmer to make sure that if two trees are equal, they are shared. Another possibility is to use *memo functions* [CL97, PJME99] to make sure that equal subtrees are identical. Using this idea, we have made a simple functional implementation of Binary Decision Diagrams (BDD's) [Bry86]. Exploring the semantic theory of extensions such as memo-tables or hashable references remains as a topic for future work.

Acknowledgements We thank John Hughes for many helpful discussions in the initial stages of this work. Andrew Pitts suggested that we might be able to do without an environment in the abstract machine – as turned out to be the case.

References

[AFM+95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proc. POPL'95, the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM Press, January 1995.

[AS98] Z. M. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. POPL'98, the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–74. ACM Press, January 1998.

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

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[CL97] B. Cook and J. Launchbury. Disposable memo functions. In *Haskell Workshop*. Amsterdam, ACM SigPlan, 1997.

[CLM98] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Formal Techniques for Hardware and Hardware-like Systems*. Marstrand, Sweden, 1998.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992. Presents call-by-value lambda-calculi extended with assignment and control constructs. Proves conservative-extension theorems. Improves on Felleisen's earlier calculi by reducing the need for rewrite rules that apply only at the root of terms.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, December 1996.

[Hut92] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93, the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM Press, January 1993.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56. ACM Press, January 1999.

[MT91] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[O'D93] J. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Springer-Verlag Workshops in Computing, pages 178–194, 1993.

[O'D96] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languagues in Education*, volume 1125 of *Lecture Notes In Computer Science*, pages 221–234. Springer Verlag, 1996.

[Ode94] Martin Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994.

[Pit96] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *11th Annual Symposium on Logic in Computer Science*, pages 152–163. IEEE Computer Society Press, Washington, 1996.

[PJME99] S. Peyton Jones, S. Marlow, and C. Elliot. Stretching the storage manager: weak pointers and stable names in Haskell. In *Implementation of Functional Languages*. Nijmegen, 1999. Submitted.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96, the 1st ACM SIGPLAN International Conference on Functional*

Programming, pages 1–12. ACM Press, May 1996.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[PS93] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Intl. Symp, Gdansk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[She85] M. Sheeran. Designing regular array architectures using higher order functions. In *ACM Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes In Computer Science*. Springer Verlag, 1985.

[SPJ97] P. Sansom and S. Peyton Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):334–385, January 1997.

[Wad92] P. Wadler. Monads for Functional Programming. In *Lecture notes for Marktoberdorf Summer School on Program Design Calculi*, NATO ASI Series F: Computer and systems sciences. Springer Verlag, August 1992.

A Appendix

Here is one way we can implement our proposed extension in a Haskell system. Since we are making a non conservative extension to Haskell, we cannot express it using standard Haskell contructs. Therefore, we have to use some "unsafe" operation, called `unsafePerformIO`. This operation, which is not part of the Haskell standard but supported by all major compilers, allows the execution of impure actions by a function that looks pure from the outside.

The only way to perform side-effecting actions in Haskell is to embed them in the *IO monad*, an abstract datatype that allows an encapsulated treatment of imperative actions. So `IO a` is the type of a computation that performs some side-effect and produces a result of type `a`. The type of the "unsafe" operation simply hides the side-effect: `unsafePerformIO :: IO a -> a`.

In the implementation in Figure 6 we use the IO references of nonstandard library *IOExts*, which is part of the Hugs-GHC extension libraries.[3] We implement our references by creating an abstract datatype `Ref`, which only supports creating, reading and the comparison of such references.

[3] www.haskell.org/libraries/

```
module Ref
  ( Ref    -- type
  , ref    --:: a -> Ref a
  , deref  --:: Ref a -> a
  , (<=>)  --:: Ref a -> Ref a -> Bool
  )
 where

import IOExts -- The Hugs-GHC Extension Libraries
 (IORef, newIORef, readIORef, unsafePerformIO)

-- Ref: the type of references
newtype Ref a = MkRef (IORef a)

-- operations

ref :: a -> Ref a
ref x = MkRef (unsafePerformIO (newIORef x))

deref :: Ref a -> a
deref (MkRef ref) = unsafePerformIO (readIORef ref)

(<=>) :: Ref a -> Ref a -> Bool
(MkRef ref1) <=> (MkRef ref2) = ref1 == ref2
```

Figure 6: A Possible Implementation