Observable Sharing for Functional Circuit Description

Koen Claessen and David Sands

Department of Computing Sciences,
Chalmers University of Technology and Göteborg University, Sweden.
www.cs.chalmers.se/~{koen,dave}

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. 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 datastructures – 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.

P.S. Thiagarajan, R. Yap (Eds.): ASIAN'99, LNCS 1742, pp. 62–73, 1999.

A problem with *observable sharing* is 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 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].

2 Functional Hardware Description

We 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].

Describing Circuits The approach of modelling circuits as functions on streams 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    and :: Signal -> Signal -> Signal
latch :: 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 are two examples of circuits. One consists of just an and-gate and an xor-gate, which is used as a component in the other.

```
halfAdd a b   = (xor a b, and a b)
fullAdd a b c = let  (s1, c1) = halfAdd a b
                     (s2, c2) = halfAdd s1 c in  (s2, xor c1 c2)
```

We use local naming of results of subcomponents using a `let` expression. The types of these terms are:

```
halfAdd :: Signal -> Signal           -> (Signal, Signal)
fullAdd :: Signal -> Signal -> Signal -> (Signal, Signal)
```

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.

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       and as bs = zipWith (&&) as bs
latch bs = False : bs       xor as bs = zipWith (/=) as bs
```

We can simulate a circuit by applying it to inputs. The result of evaluating `fullAdd [False,True] [True,True] [True,True]` is `[(False,True),(True, True)]`, while the result of `toggle` is `[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.

Generating Netlists Simulating a circuit is not enough. If we want to implement it, for example on an FPGA, or prove properties about it, we need to

generate a *netlist* of the circuit. This is a description of the all components of the circuit, and how they are connected.

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.

```
type Signal = Var String | Comp String [Signal]
inv   b = Comp "inv"   [b]     and a b = Comp "and" [a, b]
latch b = Comp "latch" [b]     xor a b = Comp "xor" [a, b]
```

Now, we can for example symbolically evaluate `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 evaluate `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 a circuit which are themselves output wires of another circuit. The `Signal` type is a tree, which means that when a result is used twice, is has to be copied. This shows that trees are inappropriate for modelling circuits, because physically, circuits have a richer graph-like structure.

2.1 Previous Solutions

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, for example as follows.

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

When we define a circuit, we have to explicitly label every component with a unique tag. O'Donnell then introduces some syntactic sugar for making it easier for the programmer to do this.

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.

But if explicit tagging is not the desired solution, why not let some underlying machinery *guarantee* that all the tags are unique? *Monads* are a standard approach for such problems (see e.g., [Wad92]). In functional programming, a monad is a data structure that can abstract from an underlying computation model. 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. This monadic approach is taken in Lava [BCSS98].

Introducing a monad implies that the types of the primitive components and circuit descriptions become *monadic*, that is, their result type becomes monadic. A big disadvantage of this approach is not only that we must change the types, but also the syntax. We can no longer use normal function abstraction, local naming and recursion anymore, we have to express this using monadic operators. All this turns out to be very inconvenient for the programmer.

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.

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, one 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 introduce an abstract type `Ref`, with the following operators:

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

The following two examples show how we can use the new constructs to detect sharing: (i) `let x = undefined in (let r = ref x in r <=> r)`
(ii) `let x = undefined in ref x <=> ref x`

In (i) we create one reference, and compare it with itself, which yields `True`. In (ii), we create two *different* references to the same variable, and so the comparison yields `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*. We give a formal description of the semantics in section 4.

3.1 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 are identified, since `circ2` is just a recursive unfolding of `circ1`. 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.

We can now modify the signal datatype in such a way that the creation of identities happens transparently to the programmer.

```
data Signal = Var String  | Comp (Ref (String, [Signal]))
comp name args = Comp (ref (name, args))

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

In this way, a circuit like `toggle` still creates a cyclic structure, but it is now possible to define a function which *observes* this cyclicity and therefore terminates when generating a netlist for the circuit.

3.2 Other Possible Solutions

We briefly discuss two other solutions, both of which 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.

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. The other way around is not possible however, which shows that our extension is weaker.

Gensym The language is extended with a new type `Sym` of abstract symbols with equality, and an operator that generates fresh such symbols, `gensym`. It is possible to define `gensym` in terms of our `Ref`s, and also the other way around. With the reference approach however, by get an important law by *definition*, which is: `r1 <=> r2` $= True \Rightarrow$ `deref r1 = deref r2`

4 The Semantic Theory

In this section we formally define the operational semantics of observable sharing, and study the induced notion of operational equivalence. For the technical development we work with a de-sugared core language based on an untyped lambda calculus with recursive lets and structured data.

The language of terms, Λ_{ref} is given by the following grammar[1]:

$$L, M, N ::= x \mid \lambda x.M \mid M\,x \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mid \mathsf{ref}\,x \mid \mathsf{deref}\,M \mid M \rightleftharpoons N$$

Note that we work with a restricted syntax in which the arguments in function applications and the arguments to constructors are always variables (c.f, [PJPS96, PJS98, Lau93, Ses97]. It is trivial to translate programs into this syntax by the introduction of let bindings for all non-variable arguments.

The set of *values*, Val $\subseteq \Lambda_{\mathsf{ref}}$, ranged over by V and W are the lambda-expressions $\lambda x.M$. 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).

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).

4.1 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]. Following [MS99], we believe an abstract machine semantics is well suited as the basis for studying operational equivalence.

Transitions in this machine are defined over configurations consisting of (i) a *heap*, containing a set of bindings, (ii) the expression currently being evaluated,

[1] In the full version of the paper we also include constructors and a case expression, as well as a strict sequential composition operator.

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. We present an equivalent but syntactically more economical version. Instead of reference environment, references are represented by a new (abstract) 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 pending reference equality-test of the form $(\rightleftharpoons M)$, or $(\underline{\mathsf{ref}}\, x \rightleftharpoons)$,
- a dereference deref, indicating that the reference 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 $P \perp Q$ 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 4.1; we implicitly restrict the definition to well-formed configurations. The first collection of rules are standard. The second collection of rules concern 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.

4.2 Convergence, Approximation, and Equivalence

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 equiv-

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

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

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

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

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

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

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

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

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

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

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

Fig. 1. Abstract machine semantics

alence 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 exists 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.

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* $M \mathrel{\underset{\sim}{\sqsubseteq}} N$, *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 $M \mathrel{\underset{\sim}{\sqsubseteq}} N$ and $N \mathrel{\underset{\sim}{\sqsubseteq}} M$. Note that equivalence is a non-trivial equivalence relation. Below we present a sample of basic laws of equivalence. In the statement 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.

$$(\lambda x.M)\, y \cong M[{}^{y}/_{x}]$$
$$\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[V]\}\ \mathsf{in}\ \mathbb{C}[V]$$
$$\mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \cong \mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z]$$
$$\mathsf{let}\ \{x = z, \vec{y} = \vec{M}\}\ \mathsf{in}\ N \cong \mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{z}/_{x}]\}\ \mathsf{in}\ N[{}^{z}/_{x}]$$
$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \cong N, \quad \text{if } \vec{x} \perp \mathsf{FV}(N)$$
$$\mathbb{C}[\mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ M] \cong \mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[M]$$
$$M \rightleftharpoons N \cong N \rightleftharpoons M$$

Remark: The fact that the reference constructor 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 ref. This failure could be disastrous in some implementations, because in effect a configuration-level analogy of this law is applied by some garbage collectors.

4.3 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 1 (Context Lemma). *For all terms M and N, $M \sqsubseteq N$ 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 since laws are typically 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].

In the full paper we present some key technical properties and a proof that the compiler optimisation performed after so-called strictness analysis is still sound in the presence of observable sharing.

4.4 Relation to Other Calculi

Similar languages have been considered by Odersky [Ode94] (call-by-name semantics) and Pitts and Stark [PS93] (call-by-value semantics). A reduction-calculus approach to call-by-need was introduced in [AFM+95], and extended to deal with mutable state in recent work of Ariola and Sabry [AS98]. The reduction-calculi approach in general has been pioneered by Felleisen et al (e.g. [FH92]), and its advantage is that it builds on the idea of a core calculus of equivalences (generated by a confluent rewriting relation on terms); each language extension is presented as a conservative extension of the core theory. 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. Our starting point has been the call-by-need *improvement theory* introduced by Moran and Sands [MS99]. In improvement theory, the definition of operational equivalences includes an observation of the number of reduction steps to convergence. This makes sharing observable – although slightly more indirectly.

We have only scratched the surface of the existing theory. Induction principles would be useful – and also 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], is likely to be relevant.

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. 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.

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 full version of the paper.

The feature is likely to be useful for other embedded description languages, and we briefly consider two such applications in the full paper: writing parsers for left-recursive grammars, and an optimised representation of decision trees.

[2] www.haskell.org/implementations/

References

[AFM+95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Proc. POPL'95*, ACM Press, 1995.

[AS98] Z. M. Ariola and A. Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. POPL'98*, pages 62–74. ACM Press, 1998.

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP'98*. ACM Press, 1998.

[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. *TCS*, 103:235–271, 1992.

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

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, 1993.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *TCS* 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*, ACM Press, 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 Languages in Education*, LNCS vol 1125, pages 221–234. Springer Verlag, 1996.

[Ode94] Martin Odersky. A functional theory of local names. In *POPL'94*, pages 48–59, ACM Press, 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, 1996.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, 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 *MFCS'93*, LNCS vol 711, pages 122–141, Springer-Verlag, 1993.

[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 *FPCS'95*, LNCS vol 201, Springer Verlag, 1985.

[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.