A Functional-Logic Library for Wired

Matthew Naylor
University of York
mfn@cs.york.ac.uk

Emil Axelsson
Chalmers University
emax@cs.chalmers.se

Colin Runciman
University of York
colin@cs.york.ac.uk

Abstract

We develop a Haskell library for functional-logic programming, motivated by the implementation of Wired, a relational embedded domain-specific language for describing and analysing digital circuits at the VLSI-layout level. Compared to a previous library for logic programming by Claessen and Ljunglöf, we support residuation, easier creation of logical data types, and pattern matching. We discuss other applications of our library, including test-data generation, and various extensions, including lazy narrowing.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; B.8.2 [*Performance And Reliability*]: Performance Analysis and Design Aids

General Terms Design, Languages

1. Introduction

Functions are great for describing the structure and layout of circuits [4]. They are simple, reusable, high-level, and amenable to mathematical reasoning. But *relations* are more general. By abstracting over the direction of data-flow, a single relation can capture the behaviours of many functions. Research suggests that the use of relations over functions can have a profound effect on circuit design languages. For instance, Ruby [8] – a combinator-based language for describing and reasoning about circuits – is pleasingly simple because, without loss of expressitivity, relations reduce the numbers of circuit combinators and associated algebraic laws.

More recently, relations have been found to be beneficial in Wired [1], a language for describing and analysing circuits at the VLSI-layout level. Relations are again used to simplify the combinator set, but also to support *bi-directional evaluation* and a basic form of *layout inference*. With bi-directional evaluation, accurate analyses can be expressed as non-standard interpretations: for example, in RC-delay analysis, drive resistances flow forwards and load capacitances backwards. With layout inference, circuit tiles can adapt automatically to fit within the context they are placed, simplifying the programmer's task.

A promising approach to implementing Wired is to embed it in a language that supports both functional and logic programming features. Here, we choose Haskell and provide the necessary logic programming features in the form of a Haskell library. This lets us keep Wired in a language that is relatively mature compared to dedicated functional-logic languages, such as Curry [5], and also to integrate Wired with Lava [4], an existing Haskell library which sits a higher level of circuit abstraction.

Our library is similar to the logic programming library by Claessen and Ljunglöf [2] (which we refer to as "Claessen's library"), but there are several important differences. Let us illustrate them with an example. Suppose that we wish to define a list concatenation predicate, `append`. First we must introduce a logical data type for lists, so we write:

$$(\texttt{nil} ::: (\triangleright)) \;=\; \texttt{datatype}\;(\texttt{cons}_0\;[\,] \vee \texttt{cons}_2\;(:))$$

This definition automatically introduces two list constructors for logical terms with the following types:

$$\begin{array}{lcl} \texttt{nil} & :: & \texttt{Term}\;[a] \\ (\triangleright) & :: & \texttt{Term}\;a \to \texttt{Term}\;[a] \to \texttt{Term}\;[a] \end{array}$$

So we have a clear type-level correspondence between logical terms, of type $\texttt{Term}\;a$, and normal Haskell values of type a. Introducing a new data type in Claessen's library is a lot less satisfying. Firstly, as Claessen and Ljunglöf admit, "it takes some work to add a data type to be used in the embedding". This is because the data type must be instantiated under a type class with three methods: one for variable creation, one for variable detection and one for unification. Secondly, and perhaps more seriously, the internal representation of logical variables is directly exposed to the programmer who has to define and deal with an explicit "Var" constructor for the data type.

Now, to define the `append` predicate using our library, we write:

$$\begin{array}{l} \texttt{append} \;::\; \texttt{Term}\;[a] \to \texttt{Term}\;[a] \to \texttt{Term}\;[a] \to \texttt{LP}\;() \\ \texttt{append}\;as\;bs\;cs \;=\; \texttt{caseOf}\;(as, cs)\;\texttt{alts} \\ \quad \texttt{where} \\ \quad\quad \texttt{alts}\;(a, as, cs) \;= \\ \quad\quad\quad\quad (\texttt{nil}\;\;\;, cs\;\;\;\;) \twoheadrightarrow (bs \doteq cs) \\ \quad\quad\quad \oplus\; (a \triangleright as, a \triangleright cs) \twoheadrightarrow \texttt{append}\;as\;bs\;cs \end{array}$$

Here we make use of a function `caseOf` that allows pattern matching. The pattern variables in the case alternatives `alts` are explicitly quantified by introducing a function for them. This basic support for pattern matching greatly improves the clarity of predicates. Claessen and Ljunglöf suggest that "syntactic sugar" is needed to support pattern matching, but as we see, this is not necessary.

Finally, Claessen and Ljunglöf do not discuss how to deal with primitive types and functions such as `Int` and $+$ in a logical way. We address this issue by supporting *residuation*, an evaluation strategy from functional logic programming. Our Wired implementation in particular makes heavy use of residuation.

All three of these improvements combine to greatly increase the practicality of our library. Indeed, all three are used to good effect

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'07, September 30, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-674-5/07/0009…$5.00

in our application to Wired, and also in our application to test-data generation. Furthermore, we also discuss how another important improvement, *lazy narrowing*, can be supported.

1.1 Road map

This paper is completely self-contained in that we do not assume prior knowledge of Claessen's library or Wired. In fact, most of section 2 introduces monads, so experienced Haskell programmers may wish to jump straight to Section 2.4. Following on from our discussion on monads, Section 3 presents our logic programming library. Section 4 uses our library to implement a small but useful version of Wired called MiniWired, and demonstrates MiniWired's relational features by defining and analysing a parallel prefix circuit. Section 5 discusses our library in comparison to Claessen's library and dedicated logic programming languages. It also discusses another application of the library, to test-data generation, and how the functional-logic technique of lazy narrowing can be supported. Section 6 concludes and discusses future work.

2. Preliminaries

Logic programs describe *backtracking* computations with *state*, where the state partially maps *logical variables* to values. In a pure functional language such as Haskell, both backtracking and state are computational effects that can be conveniently structured using *monads* [16]. In this section we introduce a backtracking state monad that we will later use as a basis for functional-logic programming in Haskell. The ideas presented are well established, but we summarise them so that this paper is self-contained. Readers familiar with monads and their uses may wish to skip to section 3.

2.1 Monads

Sometimes the flow of data in a pure functional program is, in Wadler's words, "painfully explicit". The problem is that the meaning of a program can become "buried under the plumbing required to carry data from its point of creation to its point of use". This plumbing is particularly annoying when the data is frequently accessed and passed on in the same ways, over and over again.

As is often the case when programming, the problem is one of finding the right *abstraction*. Wadler's solution is to use a particularly general abstraction called a *monad*. Whereas a pure computation is, in general, a function of type $a \to b$, a monadic computation is one of type $a \to m\ b$, where m is a monad that captures some *implicit side-effect*. Wadler shows that many of the side-effects found in impure languages, such as state, exceptions and non-determinism, can be simulated using monads.

More specifically, a monad is an abstract data type, parametrised by some other type, that provides the two functions of the following type class.

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

An expression of the form `return` a denotes a computation that simply returns a without any side-effect. And one of the form $m \gg\!= f$ denotes a computation that sequences the two computations m and $f\ a$, where a is the value returned by m.

2.2 Monadic Exception Handling

One kind of side-effect that is useful for some computations to have is *exception handling*. In exception handling, a computation can either return no value, if it *fails* (i.e. raises an exception), or a single value otherwise. Such a computation can be represented using the following data type.

```
data Maybe a = Nothing | Just a
```

To illustrate exception handling, suppose that a computation c, of type `Maybe` a, is sequentially composed of two smaller ones, c_0 and c_1. If c_0 fails then c should fail without ever executing c_1. Otherwise, c should fail if and only if c_1 fails. This behaviour can be captured as a monadic side-effect, freeing the programmer from continuously checking for, and propagating, failure.

```
instance Monad Maybe where
  return a       = Just a
  Nothing >>= f  = Nothing
  Just a  >>= f  = f a
```

Combining computations in this way can be thought of as "and" combination, because a computation succeeds only if both its constituents do. "Or" combination is also useful. It allows a computation to detect and recover from failure. A general interface to "or" combination of monadic computations is provided by the following type class.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  (⊕)   :: m a → m a → m a
```

In exception handling, `mzero` denotes a computation that fails, and $c_0 \oplus c_1$ denotes a computation that executes c_0, and if that fails, then c_1.

```
instance MonadPlus Maybe where
  mzero       = Nothing
  Nothing ⊕ m = m
  Just a  ⊕ m = Just a
```

Example 1 (*Failing Lookups*). Consider a function, `lookup`, that takes a key, and a list of key/value pairs, and returns the value that is paired with the given key. If the given key is not paired with any value in the list, then `lookup` should fail. With the help of two useful abstractions,

```
m0 >> m1 = m0 >>= (λ_ → m1)
guard c  = if c then return ()
           else mzero
```

the `lookup` function can be defined as follows.

```
lookup x []         = mzero
lookup x ((k,v):ps) =
  (guard (x == k) >> return v)
    ⊕
  lookup x ps
```

If the given key is paired with more than one element in the list, then the value of the *first* matching pair is returned. For example, the following sample evaluations hold.

```
lookup 'a' [('a',1),('b',3),('a',6)]  ⇝  Just 1
lookup 'c' [('a',1),('b',3),('a',6)]  ⇝  Nothing
```

Using the exception handling monad, it is straightforward to define a function that performs two lookups and adds the results.

```
add k0 k1 l =
  lookup k0 l >>= λa →
  lookup k1 l >>= λb →
  return (a+b)

add k0 k1 l =
  do a ← lookup k0 l
     b ← lookup k1 l
     return (a+b)
```

(The two definitions are equivalent. The one on the right simply makes use of syntactic sugar for monads, known as do-notation.)

The possibility that the first lookup may fail does not need to be considered explicitly by the second, as failure is propagated implicitly as a monadic side-effect. For example, the following evaluations hold.

```
add 'a' 'b' [('a',1),('b',3),('a',6)]  ⤳  Just 4
add 'c' 'a' [('a',1),('b',3),('a',6)]  ⤳  Nothing
```

□

2.3 Monadic Backtracking

A natural generalisation of exception handling is *backtracking*. In backtracking, a computation can yield zero or more results, not just zero or one. Therefore, when a computation fails, it may be possible to backtrack to an earlier computation, pick a different result, and then try executing the original computation again. Haskell's list monad provides such behaviour.

```
instance Monad [] where
  return a       =  [a]
  []      >>= f  =  []
  (a : as) >>= f =  f a ++ (as >>= f)

instance MonadPlus [] where
  mzero          =  []
  (⊕)            =  (++)
```

Example 2 (*Backtracking Lookups*). In exception handling, the `lookup` function returns only the first value that is paired with the given key in the list. In backtracking, all associated values are returned as a lazily evaluated list.

```
lookup 'a' [('a',1),('b',3),('a',6)]  ⤳  [1,6]
lookup 'c' [('a',1),('b',3),('a',6)]  ⤳  []
```

Note that the definition of `lookup` has not changed. The `add` function now returns the results of all combinations of lookups.

```
add 'a' 'a' [('a',1),('b',3),('a',6)]  ⤳  [2,7,7,12]
add 'a' 'c' [('a',1),('b',3),('a',6)]  ⤳  []
```

□

2.4 Adding State

Another kind of side-effect that is useful for some computations to have is *state passing*, whereby state is implicitly threaded through a sequence of computations, and each individual computation can read, modify or ignore it. A state passing computation can be represented as a transition function from the current state to a pair containing the next state and a return value. To support both state *and* backtracking, we use a transition function to a *list* of such pairs.

```
newtype BS s a  =  BS { run :: s → [(a, s)] }
```

Here `BS` stands for *backtracking state*. For any type of state s, `BS` s can be made a monad as follows.

```
instance Monad (BS s) where
  return a  =  BS (λs → [(a, s)])
  m >>= f   =  BS (λs → run m s
                  >>= (λ(a, s) → run (f a) s))
```

```
newVar       ::  a → BV a VarID
newVar a     =   do (env, i) ← get
                    put (insert i a env, i+1)
                    return i

readVar      ::  VarID → BV a a
readVar v    =   do (env, i) ← get ; return (env ! v)

writeVar     ::  VarID → a → BV a ()
writeVar v a =   do (env, i) ← get
                    put (insert v a env, i)
```

Figure 1. Operations of the "backtracking variables" monad

The occurrence of (>>=) on the right-hand side of the second equation refers to the (>>=) of the list monad instance. So "and" combination passes the current state to the first computation which yields a list of next-states, each of which is passed in turn to the second computation. "Or" combination passes the same current state to each alternative.

```
instance MonadPlus (BS s) where
  mzero    =  BS (λs → [])
  m0 ⊕ m1  =  BS (λs → run m0 s ++ run m1 s)
```

To abstract away from the internal representation of the `BS` monad, we define a `get` computation that returns the current state, and a `put` computation that replaces the current state with the given state.

```
get    ::  BS s s
get    =   BS (λs → [(s, s)])
put    ::  s → BS s ()
put s  =   BS (λ_ → [((), s)])
```

We chose to present this backtracking state monad here because of its simplicity, but in practice we use the more efficient variant, shown in Appendix 1, based on Hinze's two-continuation monad transformer [6].

2.5 Adding Variables

The implicit state of a logic program is a partial mapping from logical variables to values, often referred to as the current *substitution*. We specialise the backtracking state monad to support such a substitution as follows.

```
type BV v a   =  BS (IntMap v, VarID) a
type VarID    =  Int
```

The state of the `BS` monad is hard-coded to be a pair containing the substitution and its size. Variable identifiers are represented as integers, so the substitution is a mapping (`IntMap`) from integers to values of any type v. An efficient implementation of the `IntMap` data structure is provided in Haskell's hierarchical libraries, but for completeness we specify the functionality that we expect of it in Appendix 2. Functions to create, read, and write variables are defined in Figure 1.

Claessen describes an alternative way to implement logical variables using the support for *mutable references* provided by Haskell's `ST` monad. The advantage of using `ST` references is that they are polymorphic, automatically garbage collected, and can be accessed in constant time. Indeed, we believe that Claessen's approach will outperforms ours, but here we have chosen to present a simple implementation that performs reasonably well in practice.

```
unboundVar    :: LP VarID
unboundVar    =  newVar Nothing

bindVar       :: VarID → Uni → LP ()
bindVar v a   =  writeVar v (Just a)

ifBound       :: VarID → (Uni → LP b) → LP b → LP b
ifBound v t f =  readVar v ≫= decons
  where
    decons (Just a) = t a
    decons Nothing  = f
```

Figure 2. An implementation of logical variables

```
unify      :: Uni → Uni → LP ()
unify a b  =  do a_r ← root a ; b_r ← root b ; unif a_r b_r
  where
    unif (Var v)    (Var w)    | v == w  = return ()
    unif (Var v)    b                    = bindVar v b
    unif a          (Var w)              = bindVar w a
    unif (Int a)    (Int b)    | a == b  = return ()
    unif (Ctr n as) (Ctr m bs) | n == m  = unif′ as bs
    unif _          _                    = mzero

    unif′ []       []       = return ()
    unif′ (a : as) (b : bs) = unify a b ≫ unif′ as bs
```

Figure 3. Unification algorithm

3. A Library for Logic Programming

In this section we build a layer of logic programming support on top of the BV monad, including functions for creating *logical data types*, creating *logical variables*, *unification*, *residuation*, and *pattern matching*. We demonstrate the resulting library by describing some useful arithmetic and list-based relations.

3.1 Logical Terms

A *logical term* is a value, similar to the value of an algebraic data type in Haskell, that can be constructed using one of a number of data constructors. However, every logical term has the additional possibility of being a *logical variable*. For example, a logical list term could be constructed by a "nil" constructor, a "cons" constructor, *or a variable constructor*. Because such data types may be recursively defined, it is not, in general, possible to treat the values of existing Haskell data types as logical terms.

Instead, we define a *universal* data type with a specific constructor for variables, in which any algebraic data type can be encoded.

```
data Uni = Var VarID | Ctr Int [Uni] | Int Int
```

The universal data type Uni provides constructors for logical variables, compound terms, and primitive Haskell data types. (For presentation purposes, we support only one primitive type, namely Int.)

3.2 Logical Variables

The value of a logical variable is either *bound* to a logical term or *unbound*, so it can be represented using Haskell's Maybe type.

```
type Val = Maybe Uni
```

A monad for logic programming, LP, can be defined by hard-coding the values of variables to be of type Val.

```
type LP a = BV Val a
```

VarID is an abstract data type that provides the following three operations, as defined in Figure 2: unboundVar for creating a new, unbound, variable; bindVar for binding a value to an unbound variable; and ifBound for calling one of two given functions depending on whether a given variable is bound or not. The reason for making Val abstract is because it will later be redefined to support residuation.

3.3 Unification

Two logical terms can be successfully *unified* if they are equal, or if they contain unbound variables that can be instantiated so as to make them equal. However, to obtain maximum generality, unification must instantiate as *few* variables as possible. This can be achieved by allowing unbound variables to be unified with each other without having to instantiate them.

Two unbound variables can be unified simply by letting one be bound to the other. Since a variable will never be bound twice, a set of bindings like $\{a \to b, b \to c, d \to b, e \to f\}$ forms a set of trees, and no variable will appear in two trees. We take the variables at the roots of the trees as the representatives of the equivalence sets. Cycles can be easily avoided by checking that the roots of two variables are not the same variable. An important operation in the unification algorithm is therefore to find the root of a logical term.

```
root         :: Uni → LP Uni
root (Var v) =  ifBound v root (return (Var v))
root a       =  return a
```

The unification function takes two terms that are to be unified and performs a case analysis on their roots, as shown in Figure 3. If both roots are the same variable, then unification succeeds. If at least one of the roots is a variable, then that variable is bound to the other root. If both roots are instantiated to the same term constructor, then the arguments of those constructors are unified. In any other case, unification fails.

3.4 Static Typing

A problem with the universal data type representation is that every logical term has the *same* type, namely Uni. Leijen and Meijer propose *phantom types* as an elegant solution to this problem [10]. Their idea is to create a data type with a type parameter that does no occur in any construction. This type parameter is referred to as a phantom type.

```
newtype Term a = Term { uni :: Uni }
```

Now terms have the type Term a for some type a that can be controlled by an explicit type signature. This explicit typing allows data constructors for terms to be defined with the desired type. For example, the following functions define the list constructors.

```
nil   :: Term [a]
nil   =  Term (Ctr 0 [])

(▷)   :: Term a → Term [a] → Term [a]
a ▷ b =  Term (Ctr 1 [uni a, uni b])
```

Only well-typed logical terms can be constructed using these functions. It is slightly awkward that each constructor must *manually* be given a type signature, a unique identifier, and a representation in the universal type. Thankfully, these three tasks can be completely automated. Using the combinators of Figure 4, the list constructors are simply defined as:

```
cons0     ::  a → Int → Term a
cons0 f   =   λn → Term (Ctr n [ ])
cons1     ::  (a → b) → Int → Term a → Term b
cons1 f   =   λn a → Term (Ctr n [uni a])
cons2     ::  (a → b → c) → Int
          →   Term a → Term b → Term c
cons2 f   =   λn a b → Term (Ctr n [uni a, uni b])
data Pair a b  =  a ::: b
a ∨ b          =  λn → a n ::: b (n+1)
datatype d     =  d 0
```

Figure 4. Combinators for creating logical data types

```
(nil ::: (▷))  =  datatype (cons0 [ ] ∨ cons2 (:))
```

It is straightforward to support any algebraic type in this way. However, primitive types such as `Int` must be defined specially.

```
int    ::  Int → Term Int
int n  =   Term (Int n)
```

3.5 Logical Interface

It is useful to overload unification and free-variable creation so that they can be used on a number of different types. Therefore, we introduce the following type class for logical terms.

```
class Logical a where
  free   ::  LP a
  (≐)    ::  a → a → LP ()
  match  ::  Logical b ⇒ a → a → LP b → LP b
```

For now, the `match` member is not important – it will be used in the implementation of residuation in Section 3.8. To instantiate `Term` a for all a under the `Logical` class, we just need to convert between the typed and universal representations.

```
instance Logical (Term a) where
  free         =  do v ← unboundVar
                     return (Term (Var v))
  a ≐ b        =  unify (uni a) (uni b)
  match a b k  =  match′ (uni a) (uni b) k
```

The main motivation for having the `Logical` class is to support *tuple-terms*. Tuple-terms allow many variables to be created in one go, and collections of terms to be treated as if they were a single compound term. For example, the following instance defines tuple-terms of size two.

```
instance (Logical a, Logical b) ⇒ Logical (a, b) where
  free                          =  do a ← free ; b ← free
                                      return (a, b)
  (a0, a1) ≐ (b0, b1)           =  a0 ≐ b0 ≫ a1 ≐ b1
  match (a0, a1) (b0, b1) k     =  match a0 b0 (match a1 b1 k)
```

Tuple-terms of any size can be defined similarly.

Example 3 (*List Concatenation*). Given three lists, as, bs, and cs – that may be variables or contain variables – the list concatenation relation non-deterministically generates all variable instantiations such that cs is the concatenation of as and bs.

```
app           ::  Term [a] → Term [a] → Term [a] → LP ()
app as bs cs  =   do as ≐ nil ≫ bs ≐ cs
              ⊕   do (a, as′, cs′) ← free
                     as ≐ (a ▷ as′)
                     cs ≐ (a ▷ cs′)
                     app as′ bs cs′
```

This definition of `app` is said to be *polymodal*, meaning it has "many modes" of operation. For example, if any two of as, bs and cs are known then the other can be inferred. □

3.6 Pattern Matching

Term deconstruction is achieved using free variables and unification, as illustrated by the above example. However, it is often clearer and more concise to use pattern matching to deconstruct terms. Although we cannot use Haskell's `case` construct for this purpose, we can redefine `app` as follows.

```
app as bs cs  =  caseOf (as, cs) alts
  where
    alts (a, as, cs)  =
         (nil    , cs     )  ↠  (bs ≐ cs)
       ⊕ (a ▷ as, a ▷ cs)   ↠  app as bs cs
```

The `caseOf` function can be thought of as a replacement for Haskell's `case` construct in which logical terms can be matched. The variable `alts` stands for "case alternatives" and is parameterised by the free variables that appear in the patterns of all the alternatives.

Now it remains to define `caseOf` and the ↠ operator.

```
pat ↠ rhs     =  return (pat, rhs)
caseOf a as   =  do (pat, rhs) ← free ≫= as
                    pat ≐ a
                    rhs
```

3.7 Residuation

When the value of an uninstantiated logical variable is needed for a computation to proceed, one strategy, demonstrated above, is to non-deterministically instantiate it with appropriate values. An alternative strategy, known as *residuation*, is to *suspend* the computation until the variable becomes bound, and to proceed by evaluating the next, sequentially composed, computation. The suspended computation (also called *residual*) is *resumed* as soon as the variable that it is waiting on is instantiated.

We implement residuation by associating every unbound variable with a list of computations that are suspended on it. To do this, we need to alter our model of logical variables, and redefine the `Val` type as follows.

```
data Val        =  Bound Uni | Unbound [Residual]
type Residual   =  Uni → LP ()
```

A logical variable is now either bound to a value, or unbound, with a list of residuals waiting for it to become bound.

The new implementation of the `VarID` abstract data type is shown in Figure 5. The important difference is in the definition of `bindVar`: when an unbound variable becomes bound, each of its associated residuals is resumed by calling the `resumeOn` function, as defined in Figure 6. An expression of the form rs ‘`resumeOn`‘ v

```
unboundVar    =  newVar (Unbound [ ])
bindVar v a   =  do Unbound rs ← readVar v
                    writeVar v (Bound a)
                    rs `resumeOn` a
ifBound v f g =  readVar v >>= decons
  where
    decons (Bound a)      = f a
    decons (Unbound rs)   = g
```

Figure 5. An implementation of logical variables that supports residuation

```
resumeOn             :: [Residual] → Uni → LP ()
resumeOn rs (Var v)  =  do Unbound ss ← readVar v
                           writeVar v
                             (Unbound (rs ++ ss))
resumeOn rs a        =  mapM_ (resume a) rs
  where
    resume a g       =  g a
```

Figure 6. Resuming a list of residuals

applies each residual in rs to the value v, provided that v is instantiated. If it isn't, then v must be an unbound variable, so the list of residuals associated with v is updated to include rs.

To create a function that residuates, we introduce a special application operator called rigid′.

```
rigid′      :: Logical b ⇒ (Uni → LP b)
                          → (Uni → LP b)
rigid′ f a  =  do a_r ← root a
                  b   ← free
                  let g x  =  do c ← f x
                                 b ≐ c
                  [g]`resumeOn`a_r
                  return b
```

It takes a logical function as argument and returns a new version of it, which automatically suspends when applied to an unbound variable. If a function suspends, we still want to have access to its (future) result. In rigid′, the result is represented by the variable b, so we define the residual g to unify the result of f with b. This way, whenever g is resumed, the value of b is immediately updated.

The typed version of rigid′ is rigid, and is defined as:

```
rigid      :: Logical b ⇒ (Term a → LP b)
                         → (Term a → LP b)
rigid f a  =  rigid′ (f ∘ Term) (uni a)
```

Example 4 (*Rigid Arithmetic*). In order to define efficient deterministic operations on integers, we provide the following primitive deconstructor for integers that can only be defined by inspecting the internal structure of terms.

```
unint      :: Logical b ⇒ Term Int
                         → (Int → LP b) → LP b
unint a f  =  rigid′ (λ(Int a) → f a) (uni a)
```

With the help of the following shorthand,

```
liftInt₂ f a b  =  unint a (λa →
                     unint b (λb →
                       return (int (f a b))))
```

we can, for example, define rigid functions for addition and subtraction.

```
match′        :: Logical a ⇒ Uni → Uni → LP a → LP a
match′ a b k  =  do a_r ← root a ; b_r ← root b ; ma a_r b_r
  where
    ma (Var v)      (Var w)      | v == w  =  k
    ma (Var v)      b                      =  bindVar v b
                                              >> k
    ma _            (Var w)                =  rigidMatch
    ma (Int a)      (Int b)      | a == b  =  k
    ma (Ctr n as)   (Ctr m bs)   | n == m  =  zipm as bs
    ma _            _                      =  mzero
    zipm []         []        =  k
    zipm (x : xs)   (y : ys)  =  match′ x y (zipm xs ys)
    rigidMatch  =  rigid′ (λb → match′ a b k) b
```

Figure 7. Rigid matching algorithm

$$a \mathbin{\bar{+}} b \;=\; \texttt{liftInt}_2\ (+)\ a\ b$$
$$a \mathbin{\bar{-}} b \;=\; \texttt{liftInt}_2\ (-)\ a\ b$$

The results of these functions will only become known once their arguments, a and b, have been instantiated. □

Example 5 (*Relational Arithmetic*). Using the following generally-useful shorthand,

$$a \mathrel{\dot{\Leftarrow}} m \;=\; \texttt{do}\ b \leftarrow m\ ;\ a \doteq b$$

a deterministic addition relation can be defined as follows.

```
a +̂ b  =  do c ← a +̄ b
             b ⇐̇ c −̄ a
             a ⇐̇ c −̄ b
             return c
```

Once any two of a, b, and c are known, the other can be inferred. □

3.8 Rigid Pattern Matching

Recall that the caseOf function allows term deconstruction via pattern matching. A slight variant of this is rigidCaseOf, whereby matching *suspends* until the scrutinee is instantiated at least as much as the pattern it is being matched against. To implement rigidCaseOf we use the match method of the Logical class. An action of the form match p a k executes k if the pattern p matches the scrutinee a. The rigidCaseOf function differs from caseOf in that the match function is used in place of unification.

```
rigidCaseOf a as  =  do (pat, rhs) ← free >>= as
                        match pat a rhs
```

Recall that for values of type Term a, match is defined simply as match′. The definition of match' is given in Figure 7. The main difference from unification is that the continuation k is called when matching succeeds. When an unbound variable in the scrutinee is matched against a bound value in a pattern, then matching is suspended, via a call to rigid′, until that variable becomes bound.

Example 6 (*Rigid List Concatenation*). A rigid list concatenation relation can be defined as follows.

```
(+̄+)       :: Term [a] → Term [a] → LP (Term [a])
as +̄+ bs   =  rigidCaseOf as alts
  where
    alts (a, as)  =
        nil       ↠ return bs
      ⊕ a ▷ as    ↠ liftM (a ▷) (as +̄+ bs)
```

The result is returned once the spine of as is fully instantiated. Although deterministic, the relation permits the values in the lists

involved to flow in any direction, for example, from as and bs to the result, and vice-versa. □

4. Application to Wired

The performance of modern circuits largely depends on the effect of *wire-length* on circuit properties such as signal delay and power consumption. To help design such circuits, a language called *Wired*, embedded in Haskell, has been developed in which wires are expressed explicitly just as components are. In this section we reiterate the benefits of making Wired a relational language, originally explained in [1], but do so more precisely by fully implementing a simple version of it called *MiniWired*. We also discuss some alternative design decisions that are made possible by the availability of non-determinism, and demonstrate MiniWired by describing and analysing a parallel prefix circuit.

4.1 Descriptions in Wired

In MiniWired, circuits are modelled as rectangular *tiles* that can be composed with each other to give larger tiles. Data may enter or leave a tile on each of its four *edges*, which are named west, north, south and east:

```
data Tile a  =  Tile { west   :: Edge a
                     , north  :: Edge a
                     , south  :: Edge a
                     , east   :: Edge a }
```

The *edge* of a tile is a list of *segments*, and each segment may or may not carry a signal. The length of the edge is stored explicitly for convenience, although this information is redundant.

```
data Edge a  =  Edge { len   :: Term Int
                     , segs  :: Term [Maybe a] }
```

The following function for constructing an edge from a list of segments is useful.

```
edge as  =  Edge (int (length as)) (foldr (▷) nil as)
```

The `Edge` and `Tile` types are isomorphic to pairs and 4-tuples respectively, and can be instantiated under the `Logical` class in the same way that tuples are, as described in section 3.5. However, tiles are treated slightly different to 4-tuples in that when they are created, they are constrained to be *rectangular*. The full instance definitions of `Edge` and `Tile` are shown in Appendix 3.

A circuit description is defined to be a relation over tiles. The type variable a represents the type of data that flows through the circuit.

```
type Desc a  =  LP (Tile a)
```

A description has both a physical size and a behaviour. The size is defined by the number of segments in the north and west edges of the tile (or equivalently, the south and east edges). The behaviour is defined by a *relation* between the signals on the edges of the tile. The use of a relation rather than a function greatly reduces the number of distinct primitive tiles that are required. Diagrams of a selection of primitive tiles are shown in Figure 8. Two of these primitives, `dot` and `thinY` are defined in Figure 9, and the rest are defined in Appendix 4.

4.2 Combinators

MiniWired allows two circuit descriptions to be combined by placing them one *beside* the other, or one *below* the other. To define these two combinators it is necessary to be able to join the edges of tiles together.

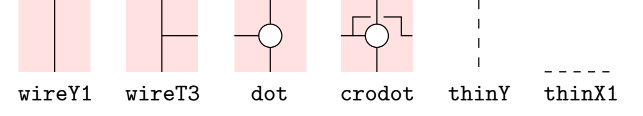

Figure 8. Some primitive tiles

```
dot f  =
  do (a, b)  ←  free
     c       ←  f a b
     let w   =  edge [just a]
     let n   =  edge [just b]
     let s   =  edge [just c]
     let e   =  edge [nothing]
     return (Tile w n s e)

thinY  =  do we      ←  free
             let ns  =  edge []
             return (Tile we ns ns we)
```

Figure 9. Definitions of the primitive tiles `dot` and `thinY`

```
(↔)      ::  Desc a → Desc a → Desc a
d0 ↔ d1  =   do t0 ← d0 ; t1 ← d1
                east t0 ≐ west t1
                n ← join (north t0) (north t1)
                s ← join (south t0) (south t1)
                return (Tile (west t0) n s (east t1))
```

Figure 10. The “beside” combinator

```
join e0 e1  =  liftM2 Edge
                 (len  e0 +̂ len  e1)
                 (segs e0 +̂+ segs e1)
```

Notice that where two edges are joined, their combined length is computed using the addition relation defined in Example 5. The reason for using a relation is to allow the length of an edge, which may be unknown, to be inferred from its surrounding context. For example, the height of `thinY`, which is unconstrained, can be inferred if it is placed beside a circuit of known height, such as `wireY1`. This ability to infer lengths from contexts can simplify circuit description by reducing the amount of information that needs to be given explicitly by the programmer.

The definition of the “beside” combinator is given in Figure 10. A description is placed beside another by joining the northern and southern edges and unifying the eastern western edges. The “below” combinator is defined similarly in Appendix 5.

A row of tiles can be obtained by iteratively applying the “beside” combinator the desired number of times.

```
rowN 0 d  =  thinY
rowN n d  =  d ↔ rowN (n−1) d
```

Sometimes the desired length of a row is obvious from the context in which it is placed, and shouldn’t need to be stated explicitly. Such a row – any number of tiles long – can be defined as follows.

```
row d  =  thinY ⊕ (d ↔ row d)
```

However, there is a slight problem with this definition: it has infinitely many solutions, and so circuit generation will get stuck

```
sklansky f 0  =  thinX1
sklansky f n  =  (left ↕ skl) ↔ (right ↕ skl)
  where
    skl       =  sklansky f (n−1)
    left      =  row wireY1 ↔ wireT3
    right     =  row (crodot f) ↔ dot f
```

Figure 11. Definition of Sklansky's prefix network in MiniWired

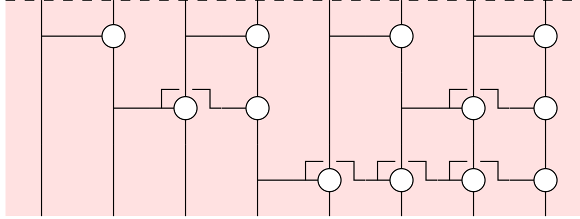

Figure 12. The 8-bit instance of Sklansky

down an infinitely long search path. A much more sensible approach is to wait until the length of the surrounding context is known before searching; this way circuit generation should be terminating.

```
row d  =  do n ← free
             s ← unint n (λn →
                    thinY ⊕ (d ↔ row d))
             n ≐ len (north s)
             return s
```

So far, tile edges have not been constrained to have non-negative length, and so by creating negatively sized tiles, `row` may still be non-terminating. This can be resolved by modifying the rigid subtraction operator to fail if its result is negative, as shown in Appendix 6. Alternatively, `row` can be defined to be fully deterministic, as shown in Appendix 7. The deterministic and non-deterministic variations of `row` behave differently when rows are placed beside rows: the former favours expanding the first row, whereas the latter fairly expands each in a non-deterministic fashion. Future experiments with Wired should help us decide which is preferable, but the availability of non-determinism certainly gives the library a more flexible feel.

The use of relations also helps to reduce the number of distinct combinators that need to be defined because relations abstract over the direction of data flow.

Example 7 (*Sklansky*). A parallel prefix network takes a list $x_0, \cdots, x_n$ and applies a given associative operator $\circ$ to every prefix of the list to yield the list $x_0, x_0 \circ x_1, x_0 \circ x_1 \circ x_2, \cdots, x_0 \circ \cdots \circ x_n$. These networks are very common in modern microprocessors – for example, in the carry generation in fast adders (see [7] for further explanation). Sklansky's minimum depth network is a recursive composition of two half-size networks. It can be defined in MiniWired as shown in Figure 11. The diagram of an 8-bit instance of this network is shown in Figure 12. □

4.3 Analysis

An important capability of Wired is to estimate and control various performance properties of a circuit, such as *propagation delay*. One way to estimate circuit delays is to use non-standard interpretation techniques: gates in the circuit are replaced by non-standard versions which compute properties of their signals rather than computing actual signal values. A non-standard gate might, for example, compute delay estimates of its output signals given delay estimates of its inputs.

For a more accurate delay analysis that takes account of some low-level electrical phenomena, *bi-directional* evaluation is required. The delay from the output of one gate to the input of another depends on the signal's *load*. The load is a measurement of the number of gates connected to the signal (and their sizes), and is characterised by a capacitance value. So capacitances need to be propagated backwards and summed at every fork point before calculation of the delay can proceed in the forwards direction.

We demonstrate a simple bi-directional analysis where the output delay of each gate, d_{out}, is calculated as follows.

$$d_{out} \quad = \quad maximum(d_0, d_1, \cdots, d_n) + d_{int} + (c \times N)$$

Here $d_0 \cdots d_n$ are input delays, d_{int} is the internal delay of the gate, N is the number of gates driven by the output signal and c is a constant which decides the significance of a high fanout. The fanout, N, is calculated by backwards propagation and d_{out} by forwards propagation. This analysis can implemented using our library as shown in Appendix 8.

Non-standard interpretation may seem a little lightweight considering that a circuit could simply be passed to an existing, highly accurate, external analysis tool. But it provides a simple way to write *custom* analyses where circuit generation can *interact* with circuit analysis within the language. This is important for developing circuit generators which use search for optimisation, and for so-called adaptive circuits which are parameterised by delay profiles of their input/output signals [14]. We are planning to experiment with adaptive circuits in Wired in the future.

5. Discussion

In this section, we discuss related work and further applications and features of our library.

5.1 Embedded Logic Programming

The first embedding of logic programming in Haskell, by Seres and Spivey [15], was extended by Claessen [2] to use *monads* and *typed* logical variables. In Claessen's approach, a logical data type for lists is introduced with the declaration:

```
data List a  =  VarL (Var (List a)) | Nil | a ::: List a
```

Then the new type must be instantiated under the `Free` and `Unify` classes (containing a total of 3 methods). Our represtation of logical terms has several advantages:

- *Simplicity*. We can introduce a new logical data type very easily, without having to instantiate various type classes.
- *Clarity*. We have a clear correspondence between normal Haskell data types, of type a, and logical data types of type `Term` a. Claessen must invent new, unfamiliar names for logical data types. Our approach facilitates conversion between logical terms and normal Haskell values.
- *Abstraction*. Claessen exposes implementation details of his library, such as the `Var` data type, to the programmer. If the programmer `case`-deconstructs a logical term, how should they handle the `Var` case? Our abstraction prevents this abuse of logical terms.

Claessen does not discuss how to handle primitive Haskell types, such as integers and arithmetic, which we support conveniently using *residuation*. Furthermore, we have shown how *pattern matching* can be achieved, allowing simpler definitions of predicates. Overall, we believe these improvements make the prospect of embedded logic programming much more attractive.

In a separate project, we have developed a monad for non deterministic computations that allows both *depth* and *breadth* first search *together*. Separate disjunction operators are provided for each form of search and they interact in a sensible way. In future we may experiment with this monad as the basis for our logic programming library. We may also experiment with the *fair* conjunction and disjunction operators (for fair backtracking) presented by Kiselyov et al. [9].

5.2 Dedicated Logic Programming

Predicates written using our approach look similar to corresponding Prolog predicates. One difference is that our logical variables are *explicitly quantified* and *typed*. But the main difference is that our approach is *embedded* in Haskell. In Wired, this lets us use logic programming features where they are needed, and to use normal Haskell everywhere else. In particular, our embedding allows logical variables, unification, and residuation together with (higher order) functions, algebraic types, type classes, and existing Haskell libraries such as Lava [4]. Similar features, apart from type classes and Lava, are available in Curry [5]. Indeed, our first version of MiniWired was developed in Curry. Moving MiniWired over to our library was straightforward and its run-time performance under each compiler (MCC and GHC) was similar.

5.3 Application to test-data generation

Lindblad [11] and Naylor [13] have recently shown how functional-logic programming techniques can aid property-based testing. In particular, test-data can be automatically generated that satisfies restrictive antecedents that program properties often have. Both Lindblad and Naylor develop external analysis tools that accept Haskell programs as input. An interesting question is whether a library-based approach, such as our logic programming library, can be used to obtain the similar benefits.

To illustrate this possibility, suppose that we wish to test the following law about the standard Haskell matrix transposition function:

```
prop_transpose m =
  isMatrix m ⇒ m == transpose (transpose m)
```

Here, `isMatrix` m holds only for a list of lists m where each inner list has the same length. The problem with random or exhaustive testing is that many inputs will be generated which do not satisfy `isMatrix`. This is a bit wasteful since we know that an implication with a false antecedent will hold! A possible solution is to write the antecedent, in this case `isMatrix`, using our library:

```
sameLen        :: Term [a] → Term [a] → LP ()
sameLen as bs   =  caseOf (as, bs) alts
  where
    alts (a, as, b, bs)  =
         (nil   , nil   )  ↠  return ()
       ⊕ (a ▷ as, b ▷ bs)  ↠  sameLen as bs

isMatrix     :: Term [[a]] → LP ()
isMatrix m    =  caseOf m alts
  where
    alts (a, b, m)  =
         nil           ↠  return ()
       ⊕ (a ▷ nil)     ↠  return ()
       ⊕ (a ▷ b ▷ m)   ↠  (sameLen a b ≫ isMatrix (b ▷ m))
```

Now `isMatrix` can be used to generate test data of a suitable form for the consequent of the property. The actual content of the matrix will be left uninstantiated by `isMatrix`, so standard random or exhaustive testing could be used to finish off the test data generation.

It seems likely that residuation could also play a useful part in this application. For example, Lindblad proposes a parallel conjunction operator `>&<` such that an expression of the form `p x >&< q x` evaluates `p x` and `q x` in parallel. So, if at any stage a partially instantiated `x` falsifies `p` *or* `q` then the conjunction immediately evaluates to false. It seems that a similar strategy to parallel conjunction could be obtained by duplicating the conjunct `q x` and executing it using residuation. In this way, evaluation of the new residuating conjunct will proceed progressively as the `p` instantiates `x`.

In future, we would like to explore this whole idea more fully.

5.4 Lazy Narrowing

Logic programming predicates such as *isMatrix* can be conveniently used as data generators. And by using *lazy narrowing* [12], an evaluation strategy from functional-logic programming, so too can any boolean expression written in a functional language. So an interesting question is whether our library can support lazy narrowing.

For example, we might define `append` not as a predicate but as a monadic function:

```
append as bs  =  caseOf as alts
  where
    alts (a, as)  =
         nil        ↠  return bs
       ⊕ (a ▷ as)   ↠  liftM (a ▷) (append as bs)
```

However, this definition does not have the same general behaviour as our `app` predicate. The reason is that monadic sequencing causes the recursive call to `append` to happen *before* the head of the result list is returned. The end result is that this function doesn't terminate when its first input is uninstantiated, even if the result of the function is known to be `nil`. In lazy narrowing, evaluation of a function proceeds only when its result is *demanded*.

The above problem can be overcome by introducing *thunks* in the definition of `caseOf`. However, this approach feels rather complex. Instead, lazy narrowing seems much simpler to support if the library is *not* monadic. We are currently developing such a library in which `append` would be written as:

```
append          :: Term [a] → Term [a] → Term [a]
append as bs    =  caseOf as alts
  where
    alts (a, as)  =
        nil       ↠ bs
      ⊕ (a ▷ as)  ↠ (a ▷ append as bs)
```

Now append simply constructs an abstract syntax tree that can be easily evaluated by lazy narrowing. The problem with this approach is that evaluation will not preserve *sharing*. For an application like test data generation, we could probably live with this problem since the benefit over random and exhaustive data generation will still be very large. Alternatively, the problem could be solved using an impure feature like *observable sharing* [3]. In fact, since sharing cannot affect the result of evaluation, it could probably be argued that the library would still be purely functional, despite using impure features internally. Furthermore, compiler optimisations that change sharing behavior would not be a problem since the same result will be returned whether or not sharing is modified.

6. Conclusions and Future Work

We have presented a Haskell library that allows functional and logic programming features to be used together. A number of improvements to the original library by Claessen were made, resulting in a simpler, more powerfull, and generally more attractive library to use. In particular, we contributed easier data type creation, a cleaner representation of logical terms, pattern matching, residuation, and a clean way of dealing with primitive Haskell data types and functions. These improvements were driven by our motivating application, MiniWired, which was used both to demonstrate use of functional and logical features in a real application, and how our library could be used to capture those features.

Claessen and Ljunglöf conclude [2] that "a polytypic programming tool" would be very helpful to avoid having to instantiate type classes every time a new logical data type is added. And they also suggest that "syntactic sugar" for pattern matching could help the definitions feel less "clumsy". Our library solves both of these problems without using any features beyond Haskell'98. Arguably, our library is still a little clumsy in that pattern variables must be explicitly quantified, but our approach certainly feels much less clumsy than no pattern matching at all. Although not presented here, we use a type class to convert between values of type Term a and those of type a. It is useful, though not necessary, to instantiate this class for each new type. Thankfully, with a few combinators, this task is made very easy and can be done without any internal knowledge of how the library works.

For future work, Claessen and Ljunglöf suggested that they would like to explore how their library "can be made more practical, by using it in more realistic programs". We have applied our library to two realistic applications here: one to circuit design and the other to test-data generation.

In future work, we hope to develop a non-monadic variant of the library, based on lazy narrowing, and explore its use in property-based test-data generation. We would also like to examine the various properties of our library constructs. One of the fundamental properties that we expect to hold is that, in a deterministic program, conjunction ($\gg$) is commutative, i.e. the order in which rigid and ($\doteq$) conjuncts are expressed doesn't matter.

The next step for Wired is to apply it to the design of Sheeran's novel adaptive partial product reduction array [14] (part of a multiplier circuit). While Sheeran's original Lava design took only very crude account of wire-lengths, with a simple forwards analysis, ours should be able to achieve more accuracy, and possibly result in more efficient structures.

Acknowledgments

This research has received Intel-custom funding from the Semiconductor Research Corporation. It is also funded by the Swedish research agency Vetenskapsrådet.

The first author is supported by an award from the Engineering and Physical Sciences Research Council of the United Kingdom, and would like to thank Koen Claessen and Mary Sheeran for making possible his visit to Chalmers, from which this work resulted.

We would like to thank Mary Sheeran and the anonymous referees for substantial constructive suggestions and comments.

References

[1] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

[2] Koen Claessen and Peter Ljunglöf. Typed logical variables in Haskell. In *Proc. of Haskell Workshop*. ACM SIGPLAN, 2000.

[3] Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In *Advances in Computing Science ASIAN'99; 5th Asian Computing Science Conference*, volume 1742 of *LNCS*, pages 62–73. Springer-Verlag, 1999.

[4] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *Lecture Notes in Computer Science*, pages 355–369. Springer Verlag, 2001.

[5] Michael Hanus, Herbert Kuchen, and Jose Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

[6] Ralf Hinze. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 186–197, New York, NY, USA, 2000. ACM Press.

[7] Ralf Hinze. An algebra of scans. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004.

[8] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design.*, Elsevier, 1990.

[9] Oleg Kiselyov, Chung-Chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In *Proc. of International Conference on Functional Programming*, volume 40, pages 192–203. ACM SIGPLAN, 2005.

[10] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, New York, NY, USA, 1999. ACM Press.

[11] Fredrik Lindblad. Property directed generation of first-order test data. The Eigth Symposium on Trends in Functional Programming, New York, 2007, to appear.

[12] Rita Loogen, Francisco Javier Lopez-Fraguas, and Mario Rodriguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *The 5th International Symposium on Programming Language*

Implementation and Logic Programming, LNCS 714, pages 184–200, 1993.

[13] Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *SCAM '07: Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, Washington, DC, USA, September 2007. IEEE Computer Society. To appear.

[14] Mary Sheeran. Generating Fast Multipliers Using Clever Circuits. In *Formal Methods in Computer-Aided Design, 5th International Conference*, LNCS 3312. Springer-Verlag, 2004.

[15] Michael Spivey and Silvija Seres. Embedding Prolog in Haskell. In *Haskell Workshop*, Paris, September 1999.

[16] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.

Appendices

This section contains a number appendices containing function definitions that have been referenced in the main body of the paper.

```
newtype BS s a  =
  BS { run  ::  ∀b. s → ((s, a) → b → b) → b → b }
instance Monad (BS s) where
  return a  =  BS (λs k_s k_f → k_s (s, a) k_f)
  m >>= f   =  BS (λs k_s k_f →
                    run m s (λ(s', a) k_f →
                      run (f a) s' k_s k_f) k_f)
instance MonadPlus (BS s) where
  mzero     =  BS (λs k_s k_f → k_f)
  m_0 ⊕ m_1 =  BS (λs k_s k_f →
                    run m_0 s k_s (run m_1 s k_s k_f))
get         ::  BS s s
get         =   BS (λs k_s k_f → k_s (s, s) k_f)
put         ::  s → BS s ()
put s       =   BS (λ_ k_s k_f → k_s (s, ()) k_f)
```

Appendix 1: An efficient continuation-based BS monad

```
type IntMap a          =  [(Int, a)]
empty                  ::  IntMap a
empty                  =  []
insert                 ::  Int → a
                       →   IntMap a → IntMap a
insert i a []          =  [(i, a)]
insert i a ((j, b) : ps)
   | i == j            =  (j, a) : ps
   | otherwise         =  (j, b) : insert i a ps
(!)                    ::  IntMap a → Int → a
((i, a) : ps) ! j      =  if i == j then a else ps ! j
```

Appendix 2: Specification of the IntMap data structure

```
instance Logical (Edge a) where
  free        = do  n ← free ; as ← free
                    return (Edge n as)
  a ≐ b       = do  len a  ≐ len b
                    segs a ≐ segs b
  match a b k = match (len a) (len b)
                  (match (segs a) (segs b) k)
instance Logical (Tile a) where
  free        = do  (n, s) ← free
                    (e, w) ← free
                    (x, y) ← free
                    return (Tile (Edge y w) (Edge x n)
                                 (Edge x s)  (Edge y e))
  a ≐ b       = north a ≐ north b ≫ east a ≐ east b
              ≫ south a ≐ south b ≫ west a ≐ west b
  match a b k = match (north a) (north b)
                  (match (east a) (east b)
                    (match (south a) (south b)
                      (match (west a) (west b) k)))
```

Appendix 3: The `Edge` and `Tile` instances of the `Logical` class

```
wireT3   =
  do a          ← free
     (a0, a1)   ← fork a
     let w      = edge [nothing]
     let n      = edge [just a]
     let s      = edge [just a0]
     let e      = edge [just a1]
     return (Tile w n s e)
crodot f  =
  do a          ← free
     b          ← free
     (a0, a1)   ← fork a
     c          ← f a0 b
     let w      = edge [just a]
     let n      = edge [just b]
     let s      = edge [just c]
     let e      = edge [just a1]
     return (Tile w n s e)
fork a   = return (a, a)
wireY1   =
  do a          ← free
     let ns     = edge [just a]
     let we     = edge [nothing]
     return (Tile we ns ns we)
thinX1   =
  do a          ← free
     let ns     = edge [just a]
     let we     = edge []
     return (Tile we ns ns we)
thinY    =
  do we         ← free
     let ns     = edge []
     return (Tile we ns ns we)
```

Appendix 4: Remaining definitions of the primitive tile set

```
(↨)      :: Desc a → Desc a → Desc a
d0 ↨ d1  =  do t0 ← d0 ; t1 ← d1
               north t0 ≐ south t1
               e ← join (east t0) (east t1)
               w ← join (west t0) (west t1)
               return (Tile w (north t1) (south t0) e)
```

Appendix 5: The "below" combinator

```
a ∸ b  =  unint a (λa′ → unint b (λb′ → sub a′ b′))
  where
    sub a b  =  if a>b then mzero
                else return (int (a−b))
```

Appendix 6: Non-negative subtraction

```
row    :: Desc a → Desc a
row d  =  do n ← free
             s ← unint n (λn′ → case n′ of
                   0 → thinY
                   _ → d ↔ row d)
             n ≐ len (north s)
             return s
```

Appendix 7: Fully-deterministic definition of `row`

```
unpair p f   =  do (a, b) ← free
                   p ≐ pair a b
                   f a b
type Delay   =  Term (Int, Int)
fork         :: Delay → LP (Delay, Delay)
fork a       =  do (b_fo, b_del) ← free
                   (c_fo, c_del) ← free
                   unpair a (λa_fo a_del →
                     do a_fo ⇐̇ b_fo ∔ c_fo
                        b_del ≐ a_del
                        c_del ≐ a_del
                        return ( pair b_fo b_del
                               , pair c_fo c_del ))
gate         :: Delay → Delay → LP Delay
gate a b     =  do unpair a (λa_fo a_del →
                     unpair b (λb_fo b_del →
                       do a_fo ≐ int 1
                          b_fo ≐ int 1
                          c_fo ← free
                          c_del ← calc a_del b_del c_fo
                          return (pair c_fo c_del)))
calc d0 d1 n  =
  unint d0 (λd0 →
    unint d1 (λd1 →
      unint n (λn →
        return (int (max d0 d1+100+3*n)))))
```

Appendix 8: A simple bi-directional delay analysis by non-standard interpretation