

# A Loop-detecting Interpreter for Lazy, Higher-order Programs

John Hughes

Informationsbehandling, Chalmers Tekniska Högskola  
S-41296 GÖTEBORG, Sweden

Alex Ferguson

Department of Computing Science, University of Glasgow  
Glasgow G12 8QQ

## Abstract

Interpreters that detect some forms of non-termination have a variety of applications, from abstract interpretation to partial evaluation. A simple and often used strategy is to test for a repeated state, but this cannot handle infinite values (such as first-class functions) or unevaluated states such as arise in lazy programs. In this paper we propose using Berry and Curien's theory of sequential algorithms as a semantic foundation for loop detection: we obtain straightforwardly a loop detector for lazy higher-order functional programs, which is more effective than the simple strategy even for strict first order cases.

## 1 Introduction

We normally expect an interpreter, given a non-terminating program to interpret, to fail to terminate itself. But there are applications in which it is useful for an interpreter to stop in such cases, with a report that the interpreted program is in an infinite loop.

One such application is *abstract interpretation*[1], a method of compile-time analysis in which the program to be analysed is interpreted over abstract data, representing sets of possible concrete data. The result of an abstract interpretation may frequently be  $\perp$ , but this conveys definite information and we certainly do not expect the compiler to loop in such cases. Another application is *partial evaluation*[2], in which a program is interpreted given some of its data, yielding a 'residual program' consisting of the operations on the unknown data. In this application a loop generates an infinite residual program, and loop detection is used to express the result as a recursive program instead.

The very simplest loop-detecting strategy for functional programs is to compare the actual arguments of each function call with the arguments of any enclosing call of the same function. If they are the same, the interpreted program is looping. This strategy is used in Young's pending analyser[3] (an abstract interpreter), and in Jones' partial evaluator MIX[2]. But it suffers from serious disadvantages: since function arguments must be compared, they cannot be functions, or infinite or partial data structures. Thus the strategy is applicable only to strict, first-order functional languages, not the lazy, higher-order languages that are most interesting. Moreover, even in the strict case this strategy

fails to detect some quite obvious loops — for example, consider

```
let rec f m n = if m=0 then 1 else f m (n+1)
```

Clearly if  $m$  is non-zero, then  $f$  loops, but the loop is not detectable because no two calls of  $f$  have the same second argument.

Holst and Hughes presented a loop-detecting interpreter which keeps track of the parameters that the control flow depends on, and thereby detects the loop in the example above[4]. That interpreter also delays the comparison of arguments until they are needed in the normal course of evaluation, and thereby supports lazy evaluation. But it is limited to first-order programs operating on atomic data. More recently Holst has generalised that work to programs operating on lazy data-structures, but higher-order functions still present a problem.

This paper takes a new approach based on Berry and Curien’s theory of sequential algorithms[5]. A sequential algorithm encapsulates more information than a mapping from inputs to outputs: it tells us also how each *part* of the output depends on the parts of the input. This extra information makes it possible to define an efficient fix-point finder, which detects self-dependent parts of the fixpoint and substitutes an explicit representation of  $\perp$  — that is, detects loops. By representing functions as sequential algorithms we have constructed a loop-detecting interpreter for lazy higher-order programs.

Naturally no loop-detecting interpreter can detect all loops — we cannot solve the halting problem. This need not matter in the applications we have in mind. Let us call a program whose interpretation terminates (perhaps by detecting a loop) *quasi-terminating*. An abstract interpreter may exploit the fact that all programs operating over finite domains are quasi-terminating to guarantee termination. A partial evaluator may simply require the programmer to supply a quasi-terminating program for partial evaluation — this is the approach taken by MIX. Alternatively, we may use a *quasi-termination analysis* to check that a program is quasi-terminating before interpreting or partially evaluating it. No such analysis can recognise all quasi-terminating programs, but one may recognise a large and useful class. Of course, each loop-detection strategy needs its own quasi-termination analysis. Such an analysis for the simple strict strategy was developed by Holst[6]; we have not yet begun to investigate an analysis corresponding to the loop-detection strategy presented here.

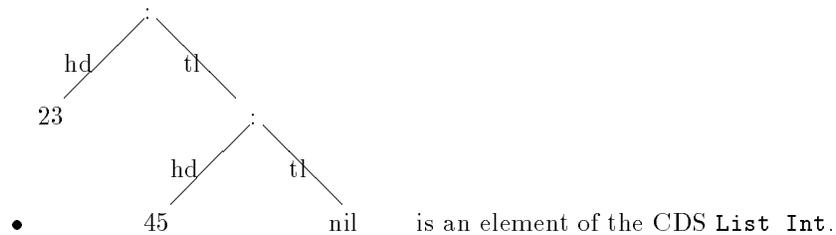
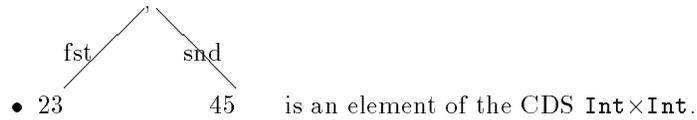
## 2 Concrete Data Structures and Sequential Algorithms

Berry and Curien’s theory is an alternative framework for denotational semantics, which makes the *sequentiality* of terms in the  $\lambda$ -calculus explicit. In place of Scott domains, we use *concrete data structures*. In place of continuous functions, we use *sequential algorithms*. In this section we give a simplified presentation of the theory.

## 2.1 Concrete Data Structures

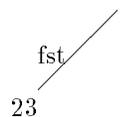
Concrete data structures (CDSs) correspond to types or domains, and their elements represent values. An element of a CDS is a *tree*, with labelled nodes and edges. Consider a few examples:

- 23 is an element of the CDS **Int** (it is a tree with a single labelled node).



We can think of the node labels as constructor names, and the edge labels as selector names. A little nomenclature: we will call the node labels *values* and the edge labels *selectors*. Any node can be identified by the sequence of selectors on the path to it from the root of the tree, and we will call such a sequence a *cell*. For example, the value 45 in the element of the list CDS above is in the cell **[tl,hd]**.

The elements of a CDS can be ordered to form a domain. The least element is the empty tree with no nodes or edges, and an element approximates another if the larger can be obtained from the smaller by adding nodes and edges. The pictures above illustrate total elements: partial elements look like total ones with some subtrees missing. For example,



represents a pair with an undefined second component.

## 2.2 Sequential Algorithms

The corresponding notion to a function from one CDS to another is a sequential algorithm. This is just a program in a very restricted language. Sequential algorithms are constructed from a **case** construct to inspect the input, and an **out** construct to produce the output.

```

    case cell of
      val: ...      out val then
      val: ...      sel: ...
      ...           sel: ...
      ...           ...

```

The cell referred to in a **case** must be a cell of the input, and the value in that cell must be one of the values listed. A **case** construct is executed by inspecting the input and continuing to execute the branch corresponding to the value found. If the input cell does not exist (because the input is a partial element) then execution stops and no output is produced. An **out** construct writes the value supplied to the root node of the output CDS, and then attaches subtrees with the given selectors to the root, using the algorithm labelled by each selector to construct the corresponding subtree. As an example, the algorithm below computes the conjunction of a pair of booleans:

```

case [] of
  ,: case [fst] of
    true: case [snd] of
      true: out true
      false: out false
    false: out false

```

Recall that cells are sequences of selectors — so `[]` refers to the root of the input (which since it is a pair must contain the pair constructor `,`), `[fst]` refers to its first component, and `snd` refers to its second. As another example, here is an algorithm to compute a pair of the boolean input and its negation:

```

out , then
  fst: case [] of
    true: out true
    false: out false
  snd: case [] of
    true: out false
    false: out true

```

Well-formed sequential algorithms must obey two rules:

- A **case** construct may not inspect the same cell as any enclosing **case**.
- A **case** construct may only inspect a cell whose parent was inspected by an enclosing **case**<sup>1</sup>.

One pleasant consequence is that there are only finitely many sequential algorithms between finite CDSs.

### 2.3 The CDS of Sequential Algorithms

A sequential algorithm may be regarded as a decision tree — and therefore may itself be represented as an element of a CDS! The nodes represent **case** and **out** constructs, and so are labelled either **case** *c* or **out** *v*, where *c* is a cell of the input type and *v* is a value of the output type. The edges emanating

---

<sup>1</sup>It is this rule that forces the algorithm for conjunction to inspect the root of its argument, even though it can only contain the pair constructor.

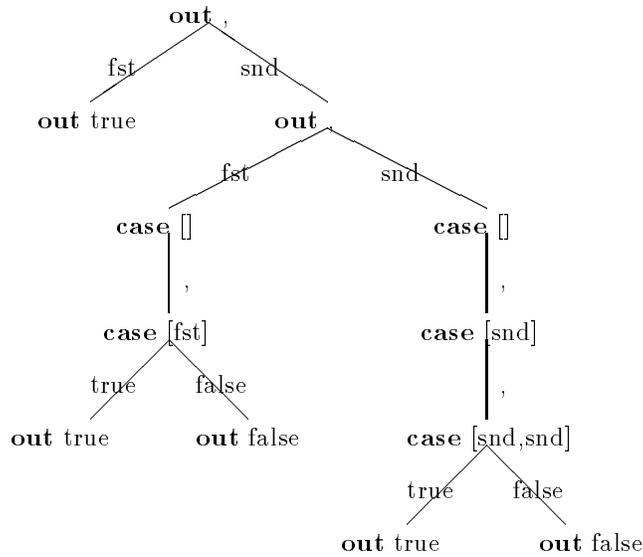


### 3 A Lazy Fixpointing Algorithm for Sequential Algorithms

The fixpoint of a sequential algorithm  $f$  can be constructed in much the same way as the fixpoint of a continuous function. Starting from the empty tree, we can repeatedly run  $f$  to construct a sequence of better and better defined partial trees. The fixpoint of  $f$  is then the union of all of these. For example, consider the function

```
f :: (Bool X (Bool X Bool)) -> (Bool X (Bool X Bool))
f p = (true, (fst p, snd(snd p)))
```

which corresponds to the sequential algorithm



Interpreted as a continuous function, the approximations to its fixpoint are  $\perp$ ,  $(true, (\perp, \perp))$ , and  $(true, (true, \perp))$ , the last of these being the fixpoint. Running the sequential algorithm repeatedly, starting with the empty tree, produces the corresponding partial trees



In each case the first approximation has two undefined components, while the fixpoint has only one. Consider the following question: how many approximations must we compute to be certain that a component of the fixpoint is really undefined? When finding the fixpoint of a continuous function, we cannot be certain of this until two successive approximations are equal *everywhere*, because the putative undefined component might depend on any other component. Thus in this example both components of the nested pair appear to be undefined after one iteration, but in fact the first component depends on a part of the fixpoint which is now available, and so becomes defined after the second iteration. Just in case the remaining undefined component depends on the first, we must compute a third iteration and verify that the result is exactly the same as after the second. This comparison of complete approximations is always expensive, may be impossible if the fixpoint is a function, and in any case may be pointless since in general we do not reach the fixpoint after finitely many iterations.

In contrast, when finding the fixpoint of a sequential algorithm we can decorate the nodes of the approximations with the cells in the previous approximations that they depend on. When a node cannot be computed, we can tell which missing cell of the previous approximation it depended on. In the example, the cell `[snd, snd]` cannot be computed in the second iteration because it depends on itself! It's now clear that this cell will remain undefined in every future iteration, so we can immediately conclude that it is undefined in the fixpoint also. It is this ability to conclude that parts of the fixpoint are undefined after finitely many iterations, even if the fixpoint itself is never reached after any finite number, that makes loop detection possible.

In the implemented algorithm, there is no need to construct a series of approximations explicitly. Instead we use *circular programming*: we give a recursive definition of an annotated tree representing the fixpoint, in which nodes are decorated with the transitive closure of the cells they depend on, and in which nodes that depend on themselves are replaced by an explicit representation of an empty tree.

## 4 An Implementation of Sequential Algorithms

In our implementation, elements of CDSs are represented using the following type:

```
rec type CDS = Event Val (Sel -> CDS) + Empty + Depend (List Sel) CDS
and type Val = Num Int + Comma + Case (List Sel) + Out Val
and type Sel = Fst + Snd + Value Val
```

Empty trees are represented by `Empty`, non-empty trees are represented as `Events`, and nodes that depend on other cells are represented during fixpointing using `Depend`.

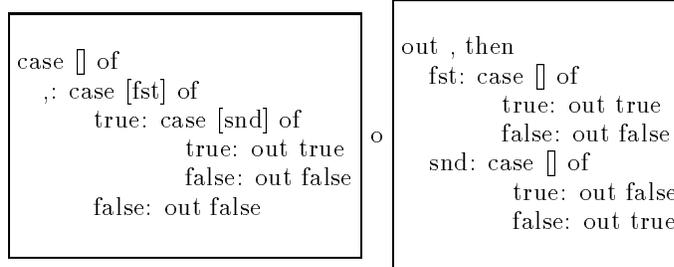
Since the decision tree for a function on integers will contain infinite branching, we have chosen to represent the collection of sub-trees of a node as a *function* from a selector name to a sub-tree. As a result, if the same cell of a CDS is inspected more than once, our implementation must normally recompute it. We regard this as reasonable since inspecting a cell of a function's decision tree corresponds to calling it.

The implementations of sequential algorithms resemble programs in continuation passing style. For example, the addition algorithm `+ :: Int X Int -> Int` is represented by

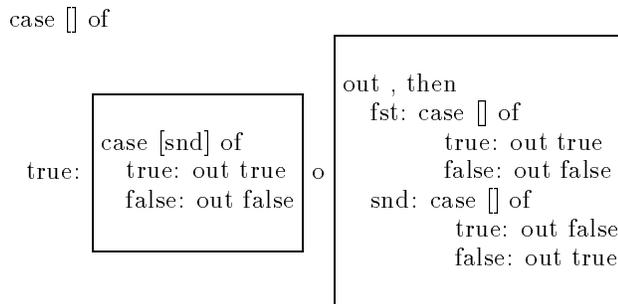
```
plusA = Event (Case []) (\(Value Comma).
  Event (Case [Fst]) (\(Value (Num m)).
    Event (Case [Snd]) (\(Value (Num n)).
      Event (Out (Num (m+n))) (\_. errA)))
```

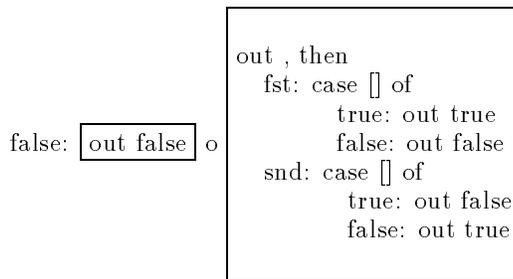
The composition `f o g` of two sequential algorithms can be constructed from the algorithm `f` by replacing `case` nodes by the code in `g` that computes the corresponding cell. Some care is needed since, although the code in `g` for computing any particular cell of its output will read each cell of `g`'s input at most once, it is possible that the algorithms for computing two different cells of `g`'s output will both read the same cell of `g`'s input. If `f` uses both cells together then the value read from `g`'s input must be saved at the first use, so that it need not be re-read for the second.

As an example, we compute the composition of the two algorithms given as examples in section 2.2: the algorithm for conjunction, and the algorithm taking a boolean input and computing a pair of the boolean and its negation. We refer below to these algorithms as `f` and `g` respectively. We are computing

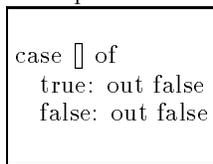


The outermost `case` in `f` reads the comma produced by the first `out` of `g`, so no computation in the composed algorithm is produced from this `case`. To compute the `[fst]` cell of `g`'s result, we copy the `case []` from `g` and then compose the `true` and `false` branches of `f` with `g`:





The **false** branch simply outputs **false**; in the **true** branch we must compute the `[snd]` cell of **g**'s result. The algorithm for doing so begins with a `case []`, but this cell of the input has been read by an enclosing `case!`. The value read was **true**, so we compute the `[snd]` cell of **g**'s output as **false**. The final composition is



As another optimisation, rather than find the code for computing each cell of **g**'s result by starting from the root each time, our implementation records the subtree of **g** at every cell inspected. Since no cell can be inspected until its parent has been, we are guaranteed to find every cell inspected in one step from this cache. Because components of trees are normally recomputed each time they are used in our implementation, this optimisation brings a very large improvement in performance.

We have also implemented currying and uncurrying of algorithms, the projections `fst` and `snd`, pairing and conditional choice of algorithms satisfying

```
(pairA f g) x = (f x, g x)
(ifA p f g) x = if p x then f x else g x
```

and the identity algorithm — in short, we have implemented a family of categorical combinators operating on sequential algorithms.

## 5 A Loop Detecting Interpreter

We have used the implementation of sequential algorithms described above to construct a loop-detecting interpreter for a small subset of LML. The syntax of the interpreted language is shown in Figure 1. The language is untyped and higher-order, with three kinds of data: numbers<sup>2</sup>, pairs, and functions. In addition to the built-in operators, `fst` and `snd` are provided as primitive functions. If recursive types such as lists are required they must be modelled using pairs.

Toy LML programs are translated into categorical combinators in the standard way: each expression denotes a function from the environment to its value. An environment binding  $n$  values is represented as nested pairs of the form

<sup>2</sup>Booleans are modelled by 0 and 1.

```

program ::= let rec def {and def}* in expr
def ::= identifier identifier* = expr
expr ::= if expr then expr else expr
        | expr op expr
        | expr expr
        | number
        | identifier
        | (expr)
        | (expr, expr)
op ::= = | < | + | - | * | /

```

Figure 1: The Syntax of Toy LML

$((\dots(\perp, x_1), \dots), x_n)$ , with the result that currying a denotation corresponds to  $\lambda$ -abstraction in the source. Each definition denotes a function from the (recursively defined) top-level environment to the value defined. By combining the denotations of the definitions we construct a function from the top-level environment to itself, and take its fixpoint. This supplies the environment used to evaluate terms.

The implemented interpreter reads a Toy LML program from a file, and then interactively evaluates expressions using the functions defined.

## 6 Practical Results

### 6.1 First-Order Examples on Atomic Data

We begin with some examples from [4].

- `let rec x = x+1`

This loop is detected by most functional language implementations.

```

x
** loop **

```

- `let rec f x y = if x=0 then f x (y+1) else y`

This function loops if `x` is zero, but not otherwise. The loop cannot be detected by the simple strict strategy since no two recursive calls have the same second argument.

```

f 0 0
** loop **

```

- `let rec h x y = if x=0 then h y (y+1) else y`

This function does not loop — but its first two calls have the same parameters as the first two calls of `f` above, making it appear to be looping at first.

```
h 0 0
2
```

- `let rec j x y = if x=0 then j (j x y) y else 0`  
This function loops if `x` is zero.

```
j 0 0
** loop **
```

## 6.2 First-Order Examples on Lazy Data-structures

The following examples illustrate the detection of looping components of partial structures.

- `let rec k x = (x, fst(k x))`  
This function doesn't loop, despite the recursive call with an identical argument.

```
k 1
(1,1)
```

- `let rec l x = (x, snd(l x))`  
This very similar function does loop, in its second component.

```
l 1
(1,** loop **)
```

- `let rec ones = (1, ones)`  
This definition does not loop — it defines an infinite structure. Our interpreter implements this lazily.

```
fst ones
1
fst (snd ones)
1
```

- `let rec len xs = if xs=0 then 0 else 1+len (snd xs)`  
This is the standard length function, operating on pairs with nil coded as zero. It works on finite lists

```
len (1, (2,0))
2
```

but the following loop is *not* detectable.

```
len ones
^C
```

The reason that this loop cannot be detected is that the CDS representation of `ones` is as an infinite tree, not as a cyclic graph. We therefore cannot verify that `len` is applied to the same argument in each recursion.

### 6.3 Higher-order Examples

- `let rec g x = g x`  
This function loops whatever its argument

```
g 0
** loop **
```

but our interpreter can in actually discover more than this:

```
g
** loop **
```

`g` is the undefined function.

- `let rec fix f = f (fix f)`  
`and id x = x`  
`and g p = (1, snd p)`

It's possible to define a fixed-point operator in Toy LML. Loop detection works just as well if loops are constructed using this operator rather than by explicit recursive definitions.

```
fix id
** loop **
fix g
(1,** loop **)
```

- `let rec upd f x y z = if z=x then y else f z`  
`and f g = if g 0=0 then 1 else f (upd g 1 (g 1+1))`  
`and g x = x+1`

Here the higher-order function `f` loops if it is passed a function mapping zero to anything but zero. But no two recursive calls of `f` have the same function parameter, so no method which relies on spotting identical parameters can detect the loop. Yet all that matters is that `f`'s parameter maps zero to the same value in each call, which it does.

```
f g
** loop **
```

### 6.4 Performance

The loop-detecting interpreter is slow. Running the `nfib` benchmark on a SPARCstation 1 it performs 60 calls per second. This is about 25 times slower than a comparable non-loop-detecting interpreter for the same language. It also uses a lot of memory: `nfib 14` cannot be computed in an 8MB heap. The present interpreter is very much a prototype, however, and we may hope to improve on these figures.

We did compare the fixpoint algorithm described with another, based on Young's pending analysis. Rather than compute a tree annotated with dependencies, the pending algorithm passes a list of 'pending cells' which are in the

process of being computed, and returns an empty tree if an attempt is ever made to inspect one of them. The pending algorithm is not amenable to memoisation since there is no obvious relationship between a cell of the fixpoint evaluated with one pending list, and the same cell evaluated with another. Nevertheless, in view of the success of pending analysis (which works in a very similar way) one might expect this algorithm to perform well. We found that, for first-order programs, it is slightly more efficient (the `nfib` number rises to 68). But for higher-order programs it is disastrously slow. For example, consider

```
let rec fix f = f (fix f)
and   fact f n = if n=0 then 1 else if 0<n then n*f(n-1) else f n
and fac = fix fact
```

which defines the factorial function as the fixpoint of a higher-order functional. This function runs thirty times faster on our present interpreter than on the ‘pending’ one.

## 7 Conclusions and Future Work

The theory of sequential algorithms provides a simple framework within which a loop-detecting fixpoint operator can be defined. Sequential algorithms can model the  $\lambda$ -calculus, so we can thereby construct a loop-detecting interpreter for higher-order, lazy programs operating on structured data. Such an interpreter is slow, but not hopelessly so.

There may be some mileage in extending the language of sequential algorithms. In particular, the identity function, and functions such as `fst` and `snd` are quite inefficient at present since they must essentially *copy* the appropriate part of the input to the output. The addition of a `copy` construct to the language, allowing the output of a sequential algorithm to share an entire subtree with the input, might potentially yield a significant improvement in time and space behaviour. The disadvantage is that every new construct must be interpreted by the sequential algorithms for higher-order functions.

Given a `copy` construct to create them, it might make sense to use *graphs* rather than trees as the fundamental data structure. This would permit cyclic structures such as the ‘infinite’ list of ones to be represented explicitly, and might enable the detection of loops such as `len ones`.

So far we have no quasi-termination analysis for any lazy loop-detecting interpreter. It will be interesting to see whether the relatively simple framework of sequential algorithms is helpful in developing such an analysis.

Finally, the loop-detecting interpreter is of little use in isolation. We plan to integrate it into a CDS-based partial evaluator. In comparison with current systems, such a partial evaluator should be able to evaluate programs lazily, unfold higher-order functions more aggressively without risking non-termination, and create fewer unnecessary versions of residual functions. We are developing a CDS-based abstract interpreter[7], which can analyse some reasonable higher-order programs over a thousand times faster than competing frontier-based systems[8].

## Acknowledgements

It has been a pleasure to work on another approach to loop detection with Carsten Kehler Holst.

## References

- [1] Patrick Cousot and Radhia Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, ACM Principles of Programming Languages, 1977.
- [2] Neil D. Jones, Peter Sestoft and Harald Søndergård, *Mix: a self-applicable partial evaluator for experiments in compiler generation*, Lisp and Symbolic Computation, 2(1):9-50, 1989.
- [3] Jonathon Young and Paul Hudak, *Finding fixpoints on function spaces*, Research Report YALEU/DCS/RR-505, Dept. of Computer Science, Yale University (December 1986).
- [4] Carsten Kehler Holst and John Hughes, *A Loop Detecting Interpreter for Lazy Programs*, Proc. Glasgow 1991 Workshop on Functional Programming, Springer-Verlag Workshops in Computing, 1992.
- [5] G. Berry and P.-L. Curien, *Theory and practice of sequential algorithms: the kernel of the applicative language CDS*, in Algebraic Methods in semantics, 35-87, Cambridge University Press (1985).
- [6] Carsten Kehler Holst, *Finiteness analysis*, in ACM Conference on Functional Programming Languages and Computer Architectures, Cambridge MA, 1991.
- [7] Alex Ferguson and John Hughes, *Abstract Interpretation of Higher Order Functions using Concrete Data Structures*, in Functional Programming, Glasgow 1992, eds. J. Launchbury and P. M. Sansom, Springer Verlag Workshops in Computing, 1992.
- [8] Sebastian Hunt, personal communications, 1992.

## A The CDS Implementation

LML implementation of CDS operations for an UNTYPED language types to represent CDSs

```
rec type CDS = Event Val (Sel -> CDS)
              + Empty
              + Depend (List Sel) CDS
and type Val = Num Int + Comma + Case (List Sel) + Out Val
and type Sel = Fst + Snd + Value Val
```

operations on sequential algorithms  
functions which copy a part of their input.

```

and   idA = copy []
and   fstA = Event (Case []) (\(Value Comma). copy [Fst])
and   sndA = Event (Case []) (\(Value Comma). copy [Snd])

```

copy c constructs an algorithm for copying the tree rooted at cell c.

```

and   copy c = Event (Case c) (\(Value v).
                        Event (Out v) (\s.
                                    copy (c@[s])))

```

primitives, constant functions, and pairing.

```

and   errA = fail "type error"
and   primA op = Event (Case []) (\(Value Comma).
                        Event (Case [Fst]) (\(Value (Num m)).
                        Event (Case [Snd]) (\(Value (Num n)).
                        Event (Out (Num (op m n))) (\_. errA))))
and   numA n = Event (Out (Num n)) (\_.errA)
and   pairA f g = Event (Out Comma)
                    (\s. case s in Fst: f || Snd: g end)

```

currying and uncurrying. Application is easily defined in terms of uncurry.

```

and   curA (Event (Case []) f) = curA (f (Value Comma))
||   curA (Event (Case (Fst.c)) f) =
      Event (Case c) (\v. curA (f v))
||   curA (Event (Case (Snd.c)) f) =
      Event (Out (Case c)) (\v. curA (f v))
||   curA (Event (Out v) f) =
      Event (Out (Out v)) (\v. curA (f v))
||   curA Empty = Empty

and   uncA f = Event (Case []) (\(Value Comma). uncA' f)
and   uncA' (Event (Case c) f) =
      Event (Case (Fst.c)) (\v.uncA' (f v))
||   uncA' (Event (Out (Case c)) f) =
      Event (Case (Snd.c)) (\v.uncA' (f v))
||   uncA' (Event (Out (Out v)) f) =
      Event (Out v) (\s.uncA' (f s))
||   uncA' Empty = Empty

and   apA = uncA idA

```

composition. comp' composes f and g, copying outputs from f and replacing cases in f by the appropriate computation from g, using the function findA to locate it. The first parameter of comp' is an association list mapping the previously inspected cells of g's result to their subtrees. The function memoise removes any multiple reads of the same cell of the input.

```

and   compA f g = memoise [] (comp' [] f g)
and   comp' m (Event (Out v) f) g =
      Event (Out v) (\s. comp' m (f s) g)
||   comp' m (Event (Case c) f) g =

```

```

        findA m c g (\m'.\v. comp' m' (f (Value v)) g)
||      comp' m Empty g = Empty

```

find m c g k finds the cell c of g's result and applies k to a new memory and the value found there. The memory contains all the cells previously inspected, which is guaranteed not to include c, but to include c's parent. The root cell is therefore a special case.

```

and    findA [] [] g k = compute g (\v.\f. k [([],f)] v)
||    findA m c g k = compute (assoc m (init c) (last c)) (\v.\f.
                                k ((c,f).m) v)

```

compute root of argument's result and apply continuation to val and successors found there.

```

and    compute (Event (Out v) g) k = k v g
||    compute (Event (Case c) g) k =
        Event (Case c) (\s. compute (g s) k)
||    compute Empty k = Empty
||    compute (Depend c g) k = Depend c (compute g k)

```

```

and    memoise m (Event (Case c) f) =
        if member (map fst m) c then memoise m (f (assoc m c))
        else Event (Case c) (\v. memoise ((c,v).m) (f v))
||    memoise m (Event (Out v) f) =
        Event (Out v) (\s. memoise m (f s))
||    memoise m Empty = Empty
||    memoise m (Depend c f) = Depend c (memoise m f)

```

conditional

```

and    ifA p f g = memoise [] (compute p (\v.\_.
                                        if v = Num 0 then g else f))

```

The fixpoint algorithm computes a cds in which each node is labelled with the cells it depends on, and self-dependent nodes are Empty. The dependencies are stripped to give the final result. This is not the standard fix: its type is  $(Env \times \tau \rightarrow \tau) \rightarrow (Env \rightarrow \tau)$ .

```

and    fixA f =
        stripD (let rec x = blackhole [] (compF f (pairA idA x))
                in x)

```

```

and    stripD (Depend c f) = stripD f
||    stripD (Event v f) = Event v (stripD o f)
||    stripD Empty = Empty

```

blackhole c cds maps to Empty any states of the CDS that depend on themselves. c is the cell of the root of the cds given.

```

and    blackhole c (Depend c' f) =
        if c=c' then Empty else Depend c' (blackhole c f)
||    blackhole c (Event (Out v) f) =

```

```

      Event (Out v) (\s. blackhole (c@[s]) (f s))
||   blackhole c (Event (Case c') f) =
      Event (Case c') (\s. blackhole c (f s))
      -- in this case we shouldn't add s to the cell being
      -- computed.
||   blackhole c Empty = Empty

```

compF composes f and g, where g produces a “dependant” CDS, and introduces dependencies of f on g to the result.

```

and   compF f g = memoise [] (compF' [] f g)
and   compF' m (Event (Out v) f) g =
      Event (Out v) (\s. compF' m (f s) g)
||   compF' m (Event (Case (Snd.c)) f) g = -- reading the fixpoint
      Depend c (findA m (Snd.c) g
                  (\m'.\v. compF' m' (f (Value v)) g))
||   compF' m (Event (Case c) f) g = -- reading the env or root
      findA m c g (\m'.\v. compF' m' (f (Value v)) g)
||   compF' m Empty g = Empty

```