

Technical Report No. 1996/04

**Higher-order Functions as Decision
Trees: Taming a Space Monster**

John Hughes, Sebastian Hunt and Colin Runciman

July 1996

Higher-order Functions as Decision Trees: Taming a Space Monster

John Hughes
Chalmers University
Gothenburg
Sweden
rjmh@cs.chalmers.se

Sebastian Hunt*
City University
London
England
seb@soi.city.ac.uk

Colin Runciman
University of York
York
England
colin@minster.york.ac.uk

Abstract

It is now over a decade since Berry and Curien published their seminal paper on concrete data structures and sequential algorithms. Although this has received attention mainly from theorists, in principle it could be used to solve a variety of practical problems. The main reason why such potential has remained untapped is the lack of any *implementation* of sequential algorithms that is not prohibitively costly. Earlier work by the first author and Alex Ferguson described an experimental implementation of sequential algorithms, for use in higher order abstract interpretation and quick enough to be highly competitive with alternative methods. But the same implementation suffers from monstrous space requirements: its space complexity is roughly linear in the *length of computations*. In this paper, we describe a substantially revised implementation in which we have largely resolved this problem of space complexity.

1 Introduction

Higher order functions are a prime ingredient of functional programming. By abstracting over functional values it is often possible to cast a solution in a way that is at once clearer, more direct and more concise than a first order alternative. But such power has a price. For the programmer, it may be difficult to trace, observe or examine the functional values passed between higher order functions. For the implementor, computations *about* higher order functions — for example, strictness analysis in a compiler — can be very expensive. In both cases, it is the choice of *representation* that poses a problem: closures in an abstract machine may be impenetrable for the programmer; compiler operations on tables defining functions extensionally are subject to combinatorial explosion.

It is now over a decade since Berry and Curien published their seminal paper[BC82] about an alternative representation of functions: their “sequential algorithms” provide a semantic model of functions as decision trees with explicit sequencing. Although this has received attention mainly from theorists, in principle it could be used to solve a variety of practical problems. For example, a sequential algorithm makes explicit the precise dependencies between inputs

*This author partly supported by a Nuffield Foundation Award to Newly Appointed Science Lecturers.

and outputs: this may be just what a programmer needs to know; in a compiler analysis, it may enable exhaustive search to be replaced by specific tests.

The main reason why such potential has remained untapped is the lack of any *implementation* of sequential algorithms that is not prohibitively costly. Berry and Curien themselves made some kind of implementation, but it was too inefficient for any practical application. A previous paper[FH93], motivated primarily by an application to higher order abstract interpretation, described an experimental implementation of sequential algorithms quick enough to be highly competitive with alternative methods. But the same implementation suffers from monstrous space requirements: its space complexity is roughly linear in the *length of computations*. In this paper, we describe a substantially revised implementation in which we have largely resolved this problem of space complexity.

Section 2 summarises Berry and Curien's sequential algorithms. Section 3 describes the standard translation from functional programs to sequential algorithms *via* categorical combinators. A straightforward implementation based on this translation makes excessive demands on memory. Computations involve the construction of many trees; though each is evaluated lazily from the root it can only grow and never shrinks; paths used as indexes into the trees grow ever longer. Section 4 shows how the problem of relentless growth can be overcome by extending sequential algorithms to incorporate explicit dereferencing. Among the key topics are how to introduce dereferencing information in the first place and how to preserve timely dereferencing through operations such as currying, uncurrying and composition. Section 5 presents three mutually supportive refinements that are important in a practical implementation: a rule to avoid copying, a reference-count garbage collector and a path-free indexing scheme. Section 6 gives a sample of experimental results we have obtained using an implementation of our ideas in Lazy ML. Section 7 briefly indicates some related work. Section 8 concludes, summarising the potential of a decision-tree representation of higher-order functions with reference to current and future applications.

2 Higher order decision trees

We can think of Berry and Curien's semantic model[BC82] as an alternative to domain theory in which *concrete data structures* play the role of domains and *sequential algorithms* play the role of functions. We shall give now a simplified account of these abstract structures.

A concrete data structure (CDS) is a domain whose elements can be represented as *forests*. Each tree has edges labelled with *selectors* and nodes labelled with *values*. The same selector labels are used to identify distinct trees in a forest: there is a labelled edge to the root of each tree. Any node in a tree is uniquely identified by its *cell*: the sequence of selectors followed to reach it. (Although this usage follows Berry and Curien, we find it a little confusing: rather we shall refer to the sequence of selectors as a *path* and use the term *cell* to refer to the node found along a given path.)

Example: A CDS for pairs of lists of integers might include the element

```

      |fst                |snd
      cons                cons
hd/   \tl                hd/   \tl
1     cons                6     nil
      hd/   \tl
      2     nil

```

in which `cons`, `nil`, `1`, `2` and `6` are values, `fst`, `snd`, `hd` and `tl` are selectors, and among the possible paths are `[fst]`, `[fst,tl,hd]` and `[snd,hd]`. \square

Sequential algorithms take a CDS forest as input and yield a CDS forest as output. The algorithms take the form of decision trees. Indeed, algorithms *are themselves* elements of a CDS, so higher order sequential algorithms can be defined. Algorithms have two possible kinds of node:

```

      case p                out v
v1/  ... \vn                s1/  ... \sn

```

A `case p` node inspects the input cell reached by path `p` and continues by following the branch labelled with the value found there. (So *values* in the input are among the *selectors* in the algorithm.) An `out v` node creates a node of the output labelled with value `v`; each succeeding subtree of the algorithm reached by a selector `s` creates a corresponding succeeding subtree of the output, also reached by selector `s`.

Example: Here is a sequential algorithm that computes the conjunction of a pair of booleans. It comprises a single decision tree with `the` as root selector.

```

      |the
      case [fst]
      true/   \false
      case [snd] out false
      true/   \false
out true    out false

```

\square

Well-formed sequential algorithms satisfy two constraints: (a) no cell may be read until its parent cell, if any, has been read; (b) no cell may be read twice along any path through the algorithm. As a consequence of (a) if we associate with any sequential algorithm a *cache* of cells already read, each successive `case` operation involves selecting a branch to follow from an existing cache entry. There is no need to retrace each path from the root. As a consequence of (b) it might be possible to discard cache entries that will not be needed again. The implementation of such caches will figure prominently in later sections.

3 Implementing functional programs as sequential algorithms

As described in an earlier paper[HF92], a small but higher order subset of Lazy ML, with the following syntax, has been implemented in an interpreter that translates programs to

sequential algorithms.

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

The interpreter is itself written in (full) Lazy ML. The type used for CDS's, including sequential algorithms, is defined along the following lines.

```
type Val = Num Int + Case Path + Out Val
type Sel = The + Fst Sel + Snd Sel + Value Val
type Path == List Sel
type CDS == Sel->Tree
type Tree = Node Val CDS
```

The decision tree for any function on integers involves infinite branching. So CDS forests and the branches from a node are represented as a *function* from selectors to trees. Consequently, as we shall see, expressions for sequential algorithms often resemble programs in continuation passing style.

The implementation applies the standard rules for compilation to categorical combinators[Cur86], but all the categorical primitives and combinators are assumed to be available as sequential algorithms. In the following program fragments we use names for them ending in **A** as a reminder of this assumption: **compA** for composition, **pairA** for functional pairing, and so on. Expressions are compiled to algorithms with an environment as input and a value as output. The rules for compiling expressions include:

```
compex env (Var v) = lookupA env v
compex env (App e1 e2) = compA apA (pairA (compex env e1) (compex env e2))
```

At compile time the environment is just a list of variable names. The result of **lookupA** is an algorithm to extract a value from a corresponding run time environment structured as nested pairs.

```
lookupA (v'._) v & (v=v') = sndA
lookupA (_.env) v = compA (lookupA env v) fstA
```

The **apA** combinator used to define function application is just the uncurried identity.

```
apA = unca idA
```

The initial run-time environment is the result of compiling a list of definitions. With each definition represented in the form `(name, (args, expr))` the relevant rules are:

```
compdefns ds = let env = map fst ds in
  fixA (compA
    (foldr (\f.\fs.pairA fs f) errA (map (compdef env . snd) ds))
    sndA)
```

```
compdef env (args,e) = comprhs env args e
```

```
comprhs env [] e = compex env e
```

```
comprhs env (a.args) e = curA (comprhs (a.env) args e)
```

It remains to discharge the assumption that sequential algorithms can be defined for the primitives including `fst` and `snd` projections, for the conditional, and for the usual family of categorical combinators including identity, pairing, currying, uncurrying and composition of functions.

3.1 primitive algorithms

The algorithm for addition, to take just one example of a primitive operator, is defined by:

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

Identity and projection functions are conveniently defined using an auxiliary function `copy`. The result of `copy p` is an algorithm for copying the input subtree rooted at path `p`.

```
idA = \s. copy [s]
fstA = \s. copy [Fst s]
sndA = \s. copy [Snd s]
```

```
copy p = Node (Case p) (\(Value v).
  Node (Out v) (\s.
    copy (p@[s])))
```

(Note: `@` is infix append in Lazy ML.) The pairing combinator can be defined like this:

```
pairA f g s = case s in Fst s': f s' || Snd s': g s' end
```

3.2 currying and uncurrying algorithms

The currying algorithm has to translate each `Case` node on a path starting `Fst s` to a similar node but with this initial `Fst` removed from the path. Similarly, each `Case` node with a path

starting `Snd s` translates to an `Out` node to construct a `Case` in the function resulting from curried application, but with the initial `Snd` removed. For the uncurrying algorithm, the same rules apply in reverse.

```
curA f s = curA' (f s)
curA' (Node (Case (Fst s.p)) f) = Node (Case (s.p)) (curA f)
curA' (Node (Case (Snd s.p)) f) = Node (Out (Case (s.p))) (curA f)
curA' (Node (Out v) f) = Node (Out (Out v)) (curA f)
```

```
unca f s = unca' (f s)
unca' (Node (Case (s.p) f) = Node (Case (Fst s.p)) (unca f)
unca' (Node (Out (Case (s.p))) f) = Node (Case (Snd s.p)) (unca f)
unca' (Node (Out (Out v)) f) = Node (Out v) (unca f)
```

3.3 composition of algorithms

The composition `compA f g` of two sequential algorithms is constructed from the *major algorithm* `f` by replacing `Case` nodes by the fragment of the *minor algorithm* `g` that computes the corresponding cell. Care is needed: although a single thread through the minor algorithm to compute any one output cell can apply `Case` to each path in its input at most once, threads to compute two or more different output cells may each separately apply `Case` to the same input paths. So a thread in the major algorithm `f` can read more than one cell in the minor algorithm `g`'s output derived from the same cell of `g`'s input. A naive replacement rule would generate composite algorithms that break the once-only rule for reading cells. To overcome this problem, the function that constructs composite algorithms *memoises* values read by the minor algorithm. We can exploit the other constraint, that no cell can be read until its parent has been read, by maintaining an *explicit cache* recording, for every cell of the minor algorithm itself that is read, the `CDS` function from that cell. Rather than search from the root for each needed fragment of this algorithm, we are guaranteed to find it in one step from the cache. Both memo table and cache can be represented simply as association lists: they map paths to values and `CDS` functions respectively.

The big problem with this implementation is the amount of memory space it requires. As it evaluates a functional program, the interpreter lazily expands the many trees involved in the program's representation. The memo table and cache for each composition are ever expanding, never shrinking, and there are many such structures since each function application compiles to a composition. We cannot simply "look ahead" to see whether cache and memo entries will be needed again: to obtain a negative answer would require executing the entire remaining algorithm, after which it would be of no interest! Also, as the depths to which trees are expanded increase, so the lengths of the lists of selectors used to access them must increase.

These problems are illustrated in Figure 1, which shows the heap profile[RW93] for a simple list-processing benchmark. The benchmark computation uses just two recursively defined functions: it computes the `length` of a list generated by `duplicating` a number five times. The top five bands in the profile represent the memory occupied by memo and cache association lists, selector paths, and closures associated with composition.

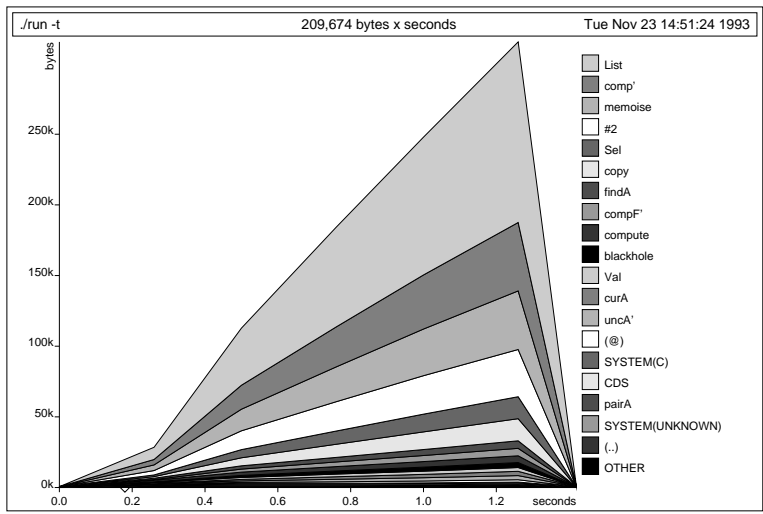


Figure 1: A heap profile of the original interpreter

4 Sequential algorithms that explicitly discard cells

What can be done to avoid the relentless growth of the input cache associated with a sequential algorithm? Intuition tells us that many entries are only needed fleetingly. Suppose the algorithm takes as its input a list represented as a `cons-nil` tree, and that it computes the length of this input in the usual way: by traversing the spine counting `cons` nodes. Each time the algorithm for `length` advances by selecting the tail of a `cons` node, we know that no further use will ever be made of the CDS function representing the branches from that node. The head will never be selected, because `length` has no need to examine elements; it is only concerned with the list spine. So an input cache entry could safely be deleted — if only the information that no further branches of the `cons` node will be selected could somehow be attached to the `length` algorithm.

4.1 dereference sets

We have experimented with several alternative schemes for incorporating such information in sequential algorithms. Most failed to preserve timely information about cache deletions through the operations of currying, uncurrying and composition. In particular, we found no satisfactory way to handle reference information by introducing new kinds of algorithm nodes separate from `case` and `out`: separating a dereference step from the last `case` in which the relevant cache entry is used leads to arbitrarily long delays in dereferencing when the algorithm is composed with another. Indeed, the *only* satisfactory scheme we have discovered is the one we now present.

To each `Case` node and to each `Out` node we attach a *set of dereference paths*. The constructions for these nodes, of type `Val`, become:

```
type Val = ... + Case (List Path) Path + Out (List Path) Val
```

The interpretation of `Case ds p` or `Out ds v` is identical to the previous interpretations of

Case `p` or Out `v`, but with an additional guarantee: in no descendent node Case `ds`' `p`' of the algorithm is `p`' an extension of one of the paths in `ds` by a single selector. That is, in any input cache all the entries associated with paths in `ds` can safely be deleted, once the Case or Out operation is performed. Importantly, the dereference set attached to a Case node can (and often does) include the very path being used for the selection.

4.2 pairs and dereferencing

Consider the representation of a composite pair such as `(1, (2,3))`.

```

| (Fst The)      | (Snd Fst The)      | (Snd Snd The)
1                2                3

```

There is no node corresponding to `(2,3)`, the second component of the outer pair. Although this is faithful to Berry and Curien, with the introduction of dereferencing it has a serious drawback: pair substructures cannot be dereferenced independently. So long as a branch to any part of a compound pair structure may yet be needed, the CDS for the whole structure must be retained. This motivates a revised representation of pairs as forests with just two branches, labelled `Fst` and `Snd`, and *boxed* components: each branch leads to a component node with a new value `Box` from which substructures of the component may be selected. The definitions of the `Val` and `Sel` types become

```

type Val = Num Int + Case Path + Out Val + Box
type Sel = The + Fst + Snd + Value Val

```

and the representation of the above example becomes

```

| Fst              | Snd
Box                Box
|                  Fst / \ Snd
1                  Box   Box
                   |    |
                   2    3

```

4.3 currying and void values

Adding appropriate dereference sets to the definitions of primitive algorithms is quite straightforward. The crux of the problem is how to deal with them in currying, uncurrying and composition.

From each dereference set in an uncurried algorithm *two* dereference sets must be generated for the curried version. A dereference path of the form `Fst.p` must translate to an element `p` of a dereference set attached to a Case or Out node of the curried algorithm *itself*; whereas the translation `p` of a dereference path of the form `Snd.p` belongs in the dereference set of a node in the curried algorithm's *result* (itself an algorithm). Assuming an auxiliary `curds` to compute this pair of sets we have:

```

curA' (Node (Out ds v) f) = Node (Out nds (Out wds v)) (curA f)
curA' (Node (Case ds (Snd.p)) f) = Node (Out nds (Case wds p)) (curA f)
where (nds,wds) = curds ds

```

However, currying a node `Case ds (Fst.p)` poses a problem: what if `ds` includes a path `Snd.q`? We need to generate some node of the result algorithm so that we can include `q` in its dereference set. *There is no suitable node.* The currying routine *cannot* simply accumulate such dereferences and attach them to the next available node: the consequent delays are disastrous for some algorithms. Our solution is to introduce a distinguished value `Void`. We interpret `Out ds Void` as a null operation so far as the input/output behaviour of an algorithm is concerned. Its only purpose is to convey the information in dereference set `ds` so that cache entries can be deleted; the node carrying it has a single branch with selector `Next`. Now we can add the equation:

```

curA' (Node (Case ds (Fst.p)) f) =
  Node (Out [] (Out wds Void)) (\Next. Node (Case nds p) (curA f))

```

4.4 uncurrying and boxed components

The same rules applied in reverse lead to a definition of uncurrying. But here another problem arises. No `Case` in a curried algorithm will translate to a `Case` on the components of the pair read by its uncurried equivalent, and no dereferencing in a curried algorithm translates to a dereferencing of the entire argument in the uncurried equivalent. Retaining a reference to the entire argument has serious consequences: in composition, for example, it may result in memo entries being retained for *every* path. Therefore the uncurried translation of *every* curried function must begin:

```

Node (Case [] [Fst]) (\(Value Box).
Node (Case [[]] [Snd]) (\(Value Box).
...

```

That is, the algorithm begins by “opening the boxes” containing the arguments (*without* reading the arguments themselves) and discarding the pair structure in which they were wrapped. Since algorithms for curried functions are not *forced* to begin in this way, currying and uncurrying are not quite inverses — but neither are they exact inverses in many functional languages with pattern-matching, where matching an argument against a pair makes a function strict. The important point is that currying followed by uncurrying *cannot delay* the time at which the root is dereferenced, though it *may advance* it.

Moreover, we can extend the same principle to the pair-structured environment. Each algorithm starts by opening the boxes it requires (*without* evaluating their contents), and discards the rest by dereferencing the environment structure from which they would be accessed. To make this compositional, we compile expressions to functions with boxed results. Each algorithm (a) unpacks what it needs from the environment, (b) generates the `Box` for its own result, and only then (c) computes the “contents” of this box.

4.5 composition

The cache used in composition is closely related to the input cache of the major algorithm: the only difference is that entries do not contain CDSs for parts of the input, but CDSs for parts of the minor algorithm that create this input. The obvious use for dereference sets is to discard unneeded entries in this cache. But how are we to determine the appropriate dereference sets for inclusion in the composite algorithm? And can dereferences also be applied to the deletion of entries in the memo table?

Each entry in the major cache refers to a thread in the minor algorithm *which itself has a cache*.

```
type MajCache = List (Path x CDS x MinCache)
type MinCache = List Path
```

Composition must also model the cache of the composite algorithm, which can be represented as a `MinCache`.

```
type ComCache == MinCache
```

Now we can formulate first approximations to the rules for dereferencing in the composite algorithm and for pruning the memo table. Whenever dereferencing occurs in the major algorithm, there are corresponding deletions in the major cache: let `rps` be the union of all `MinCache`'s found in *retained* `MajCache` entries.

1. Every path in the composite cache that has no prefix in `rps` can be dereferenced in the composite algorithm.
2. For every path in that has no *proper* prefix in `rps` the corresponding memo entry can be deleted.

However, these rules can be strengthened considerably if with each path in every minor cache (including the composite cache) we also record the branches from that path that have already been followed. A list of selectors suffices.

```
type MinCache = List (Path x List Sel)
```

Two observations can be used, both based on the constraint that cells can be read once only:

1. The *spent selector* rule: A retained minor cache entry `(p,ss)` does *not* constitute a need to retain a memo or composite cache entry for a path `p@s.q` if `s∈ss`.
2. The *selector exhaustion* rule: A composite cache entry `(p,ss)` need not be retained if `ss` exhausts all possible selectors from `p`.

The original implementation used integer values 0 and 1 for conditions, but the selector exhaustion rule motivated a more careful use of typed values, including a type of truth values. Note that this rule may permit a composite algorithm to dereference some path even though *none* of the minor algorithm threads can yet do so.

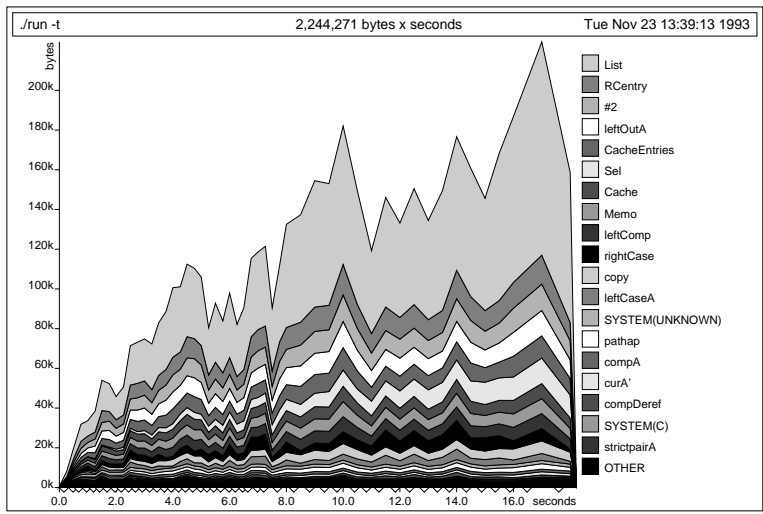


Figure 2: A heap profile of the dereferencing interpreter

5 Space-saving refinements

The introduction of dereference information was successful. We could now trace computations and see small caches of constant size in place of the large and growing caches we had observed before. Despite a lot of extra structure, both in algorithms themselves and in the caches maintained for composition, this improved space complexity in the abstract was also reflected in reduced profiles of actual memory use: compare Figure 2 with Figure 1.

To achieve effective dereferencing it had been necessary to make *all* the attendant changes described in the previous section. Partly as a result of these changes, three significant concerns remained:

1. The amount of tree-processing involved in applying combinatory operations to algorithms, even in the evaluation of small list-processing functions, seemed excessive. (Runtime for the benchmark problem has increased tenfold.)
2. Although effective, the rules used to detect unneeded memo and cache entries were costly to apply: we were uncomfortably close to invoking an intelligent mark-sweep garbage collector on every reduction! (Another reason for the much longer computation times.)
3. The problem of increasing path lengths remained. In fact, it was now worse: the introduction of boxing lengthens paths; the introduction of dereference sets increases the number of paths. (Much of the list structure represented by the top band in Figure 2 is accounted for by paths.)

We shall describe our remedies to these three problems, in order.

5.1 copy avoidance

In a drive to reduce the amount of tree-processing, the `copy` auxiliary is an obvious target. Recall that `copy` is used to define the primitive algorithms `idA`, `fstA` and `sndA`. These are

heavily used: uncurried `idA` is used in every function application; compositions of `fstA` and `sndA` are used every time a variable is looked up in the environment. The inefficiency of copy-based operations to access the environment is particularly striking: extracting the N th value from the environment involves copying it N times!

We therefore extend the language of sequential algorithms by making `Copy p s` a legitimate `Val` construction that can be used in algorithm trees. The motivation is that we can express the results of applying some combinators to `Copy` algorithms directly in terms of `Copy`: in effect, this achieves copying by sharing. Though some `Copy` nodes may have to be expanded to a tree containing only `Case` and `Out` nodes, this can be done only where, and as far as, really necessary.

Once again, the currying and uncurrying operations motivate a useful generalisation: *kth order copying*. We define the interpretation of `Copy k p s` for $k \geq 0$ by the expansion:

```
expand (Copy k p s) = let ps = p@[s] in
  Node (Case [p] ps) (\(Value v).
    Node (power (k+1) (Out []) v) (\s'.
      Node (Copy k ps s') errA ))
```

where `power n f x` computes $f^n x$. That is `Copy k` copies a part of the input tree but with `Out []` applied k times to each node value. Now we have when currying:

```
curA' (Node (v as (Copy k [] s)) _) = curA' (expand v)
curA' (Node (Copy k (Fst.p) s) _) = Node (Copy (k+1) p s) errA
curA' (Node (Copy k (Snd.p) s) _) = Node (Out [] (Copy k p s)) errA
```

and when uncurrying:

```
unca' (Node (v as (Copy 0 p s)) _) = unca' (expand v)
unca' (Node (v as (Copy k p s)) _) = Node (Copy (k-1) (Fst.p) s) errA
unca' (Node (Out ds (Copy k p s)) _) = Node (Out (map (Fst.) ds) Void) (\Next.
  Node (Copy k (Snd.p) s) errA )
```

In a composition, when `Copy 0` occurs as (part of) the major algorithm, the composition reduces to (correspondingly cached part of) the minor algorithm — subject to a simple condition on the memo table.

It is important that `Copy` is a new `Val` construction and *not* a new `Tree` construction. This enables higher-order sequential algorithms to see when their arguments are copy functions, and treat them specially.

5.2 reference count garbage collection

One way to determine legitimate dereferences in a composite algorithm and deletions from composition memo tables is by explicit traversal of all minor caches at each dereferencing point in the major algorithm. It works, but it's slow!

An alternative is to maintain all the memo and cache information associated with each composition in a single structure with reference count information. A *forest* is the appropriate structure, as in general the minor algorithm will be “at work” on different subtrees of its input, and the whole point is that we do not wish to maintain their common ancestors. Each tree in this memo-cache forest is indexed by a path.

```
type MCForest = List (Path x MCTree)
```

The type of individual trees is defined as follows.

```
type MCTree = MC Int Bool (List (Sel x (Maybe (Int x Sel x MCTree))))
```

Each construction `MC r d [... (s, Just (n,v,mc)) ...]` in a tree is associated with a path `p`. The number `r` is reference count, recording how many current minor caches include an entry for `p`. The value of `d` is `True` exactly if `p` has already been dereferenced in the composite algorithm. When the reference count `r` at the root of a memo-cache tree falls to zero the associated path is dereferenced in the composite algorithm if this has not already been done. The root node itself is discarded and each of its successors becomes a new tree in the forest. The purpose of the list of selection records held at each node of a memo-cache tree is to enable spent selectors and exhausted selector sets to be detected. Each selector `s` extends `p`: `n` is the number ($\leq r$) of minor cache entries for `p` that include `s` in their list of spent selectors; `v` is the memoised value read from input cell `p@[s]`; `mc` is the `MCTree` associated with the path `p@[s]`.

We do not have space to here for further details of deallocation based on reference counts. In fact, the gains from reference-counting rather than searching are not as great as might be expected. Incrementing and decrementing reference counts is quite expensive, because of the costs of locating and updating them in trees. An optimised implementation of single-threaded structures *cannot* help in this instance, as multi-threaded use of the memo-cache structure is essential. However, reference-counting does yield some improvement in performance, and by doing away with cache deletion rules based on comparison of paths it also paves the way for a pathless representation, to be described next.

5.3 a pathless representation

Selector paths can occupy up to half the memory needed to evaluate an algorithm. To save space can we code paths as, say, small integers? So far as the interpretation of any fixed algorithm is concerned, paths can be viewed simply as indices into the the input cache, with the exception of the final selector in a `Case` path. This suggests that paths might be replaced by positional numeric indices, supplemented by a selector in `Case` nodes. To do this, we must be more precise about cache ordering. A convenient rule is that new cache entries, created by `Case` nodes, are placed at index position 0 and displace all other entries by one index position. Dereference sets in `Case` nodes contain indices into the cache *before* the addition of the new entry. When entries with indices in a dereference set are deleted, the remaining entries “close up” to occupy consecutive index positions, always starting from 0.

Besides the benefit of avoiding memory costs due to paths, this representation actually simplifies several algorithms. For example, because copying algorithms always maintain a cache of exactly one item, not only do we avoid concatenation of lists, there is no longer any need for a separate argument representing the path at all. The expansion rule becomes:

```

expand (Copy k s) =
  Node (Case [0] 0 s) (\(Value v).
  Node (power (k+1) (Out []) v) (\s'.
  Node (Copy k s') errA ))

```

Once again, the main obstacles to the change of representation are the currying, uncurrying and composition operations. In the interpreter with explicit paths, currying examines paths to see whether they begin with `Fst` or `Snd`. Similarly, uncurrying works by prefixing these selectors onto paths. In the pathless implementation, it is necessary for both currying and uncurrying routines to maintain a *shadow cache* of the *uncurried* function. Each shadow cache entry is a simple value: one of `Empty`, `Fst` or `Snd` indicating the kind of path that would have been present in the explicit path representation. Index positions of `Fst` or `Snd` entries in this uncurried cache are translated for the curried version by subtracting the number of prior entries of the other kind. An index position `i` in a curried function is translated in the uncurried version to the position of the `i`th entry of the appropriate kind in the shadow cache.

Composition appears at first to be even more dependent on explicit paths. It is necessary to relate entries in different minor caches, and until now this has been done by comparing their path indices. Positional indices cannot be used directly, since entries for the same path may occur at different positions in different caches. However, we can restore a common form of reference by arranging that composition assigns each path a unique *path number* when it is first the subject of a `Case`. It suffices to use consecutive naturals as path numbers. In order that we can translate between path numbers and positional indices in a given minor cache, each minor cache entry includes a path number. To enable a similar translations between path numbers and positional indices in the composite algorithm's cache, a shadow cache for the composite algorithm is maintained: each entry in this shadow cache is just a path number.

One disadvantage of the pathless representation is a slight weakening of the rule reducing compositions with `Copy 0` as the major algorithm to the corresponding minor algorithm. This rule now applies only if the order of minor and composite cache entries coincides. In practice, we have found that few applications of the rule are lost; often the condition is trivially satisfied as both caches have only one entry.

6 Experimental Results

Figure 3 completes the series of heap profiles begun in Figures 1 and 2. We now have a clear order of magnitude improvement in the amount of memory required for this computation, as compared with the original interpreter. There is still some growth, but this is inherent in the desired lazy model of higher-order computation: laziness extends to the sequential algorithms representing functions — once some part of an algorithm has been constructed, it may have to be retained to avoid recomputing it later. Figure 3 also shows that the speed of the dereferencing interpreter has recovered to about half that of the original.

Figure 4 gives some indication of performance for a small sample of computations. For each version of the interpreter, before and after the introduction of dereferencing, there are columns for maximum memory use (kilobytes) and time (seconds). The example computations are: *lendup*, length of a five element list computed by duplicating a given element; *nfib*, the usual benchmark of this name with 10 as argument; *sort*, sorting a list of 3 items by insertion (worst case — reverse order); *queens*, counts the number of solutions to the N-queens problem, using

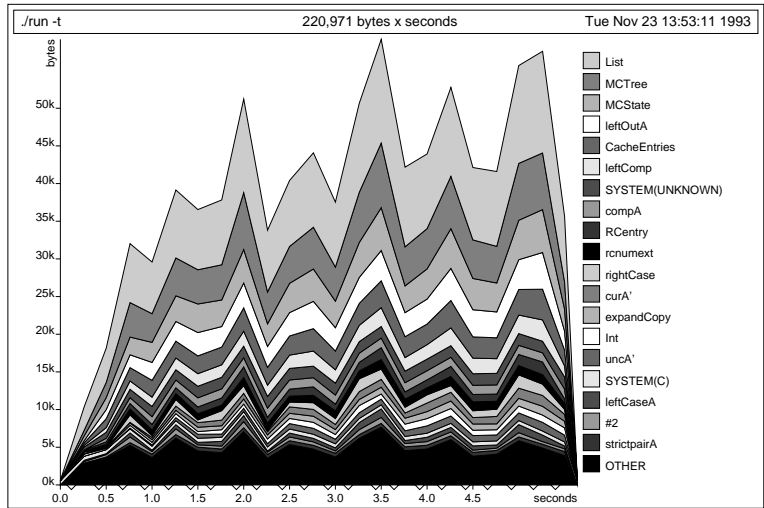


Figure 3: A heap profile of the pathless interpreter.

	old interpreter		new interpreter	
	maxmem	time	maxmem	time
lendup	800	4.3	70	9
nfib	650	1.7	70	33
sort	2200	22.0	450	34
queens			1300	160
nfibx10	850	2.3	90	46

Figure 4: Comparative performance of old and new interpreters

a program with about 20 function definitions (here $N=2$ only!); and finally *nfibx10*, applying *nfib* 10 times and summing the result.

Clearly the absolute performance of our interpreter, even with dereferencing, does not yet make it competitive as an implementation for ordinary programming! (Though it does make some practical applications possible, as we shall discuss in a moment.) Our point is rather to demonstrate progress. In every case the new interpreter requires far less memory than the old. The *nfibx10* example has been included to illustrate the “self-optimising” nature of sequential algorithms: the algorithm for *nfib* has a branch for each numeric argument; once computed, results are retained in the algorithm. So applying *nfib* ten times to the same argument costs little more than applying it once.

7 Related work

Sequential algorithms were proposed by Berry and Curien in the early 80’s as a semantic foundation for sequential programming languages[BC82]. They were primarily interested in a theoretical problem: finding a fully abstract model for PCF. However, as early as 1981 Berry proposed a practical programming language based on sequential algorithms[Ber81], and Berry and Curien did construct implementations, two of which are described in [Cur86]. Our impression, though, is that these implementations were really intended as toys for exploring the semantics, not tools for building applications.

Theoretical interest in sequential algorithms has re-awakened recently, with Cartwright and Felleison’s introduction of “observable sequential algorithms”, which solve the full abstraction problem for PCF+error-handling[CF92, Cur92].

We are not aware of any previous attempt at a practical implementation of sequential algorithms, apart from the predecessor of our current interpreter[FH93]. On the other hand, *dependency information* has been exploited in the program analysis community. Jones and Mycroft’s much-used minimal function graph analysis collects just such information for strict, first-order programs[JM86]. Our use of sequential algorithms can be thought of as an extension of minimal function graphs to lazy, higher-order programs.

8 Conclusions, applications and future work

We have described an implementation of higher-order decision trees using an extended form of Berry and Curien’s sequential algorithms. Unlike the only previous implementations known to us, it does not require space proportional to the length of computation.

We have yet to investigate the categorical properties of our modified sequential algorithms. We conjecture that they do still form a category — but not a cartesian closed category as there is only a weak product.

We believe that sequential algorithms have many practical applications. Although our experimental interpreter is neither small nor fast compared with a conventional one, its performance is good enough for work on some of these to begin. Because we have an *informative* representation of functions — dependencies between input and output are observable — self-dependencies in recursive definitions can be detected cheaply. This idea is applied in [HF92], which describes

a loop-detecting interpreter. We have implemented a loop-detecting version of our dereferencing interpreter: we had to re-introduce paths in the implementation of fixpointing, in order to identify self-dependencies, but this very limited use of paths did not lead to excessive space use. A similar idea is used in [FH93], to construct a higher-order strictness analyser which is orders of magnitude faster than comparably accurate rivals. We are working on an application to partial evaluation, where loop-detection is used to convert infinite unfolding to a recursive residual program.

Other possible applications are incremental evaluation of functional programs — the dependencies tell us what must be recomputed — and execution tracing, where dependency information can be used to print just those parts of a function’s arguments that its result actually depends on.

Often, semantically-based tools for functional languages are restricted to first-order programs, perhaps under eager evaluation. Generalising them to lazy or higher-order programs, if possible at all, may be complex and costly — a further five years’ effort seems to be typical[Myc81, GBA86]! All this could change if future tools are based on sequential algorithms: there will be no more need to treat infinite structures and functional values specially; every fresh development will apply immediately to lazy higher-order programs.

References

- [BC82] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.
- [Ber81] G. Berry. Programming with concrete data structures and sequential algorithms. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 49–57. ACM Press, October 1981.
- [CF92] R. Cartwright and M. Felleison. Observable sequentiality and full abstraction. In *Proceedings of the 9th Symposium on Principles of Programming Languages*, pages 328–342. ACM Press, January 1992.
- [Cur86] P.-L. Curien. *Categorical combinators, sequential algorithms and functional programming*. Research Notes in Theoretical Computer Science, Pitman, 1986.
- [Cur92] P.-L. Curien. Observable algorithms on concrete data structures. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 432–443. IEEE Computer Society Press, June 1992.
- [FH93] Alex Ferguson and John Hughes. Fast abstract interpretation using sequential algorithms. In *Proceedings of the 3rd International Workshop on Static Analysis*, pages 45–59. Springer Verlag, LNCS 724, September 1993.
- [GBA86] C.L. Hankin G.L. Burn and S. Abramsky. Strictness analysis of higher order functions. *Science of Computer Programming*, 7:249–278, November 1986.
- [HF92] John Hughes and Alex Ferguson. A loop detecting interpreter for lazy, higher order programs. In *Functional Programming*. Springer Workshops in Computing, 1992.
- [JM86] N.D. Jones and A. Mycroft. Dataflow of applicative programs using minimal function graphs. In *Proceedings of the 13th Symposium on Principles of Programming Languages*, pages 296–306. ACM Press, January 1986.

- [Myc81] A. Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD Thesis CST-15-81, University of Edinburgh, Department of Computer Science, December 1981.
- [RW93] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.