Programming Research Group

COMBINING MEMOISATION AND CHANGE PROPAGATION

Oege de Moor and Ganesh Sittampalam

PRG-RR-03-22

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Acar, Blelloch and Harper proposed a new algorithm for change propagation, which maintains a dynamic dependence graph augmented by time stamps. The crux of their algorithm is that the time stamps can be maintained using the ordered list structure of Dietz and Sleator, yielding only constant time overheads on a full run, and $\mathcal{O}(\log\ n)$ overheads on incremental update operations, where n is the number of steps in the full run.

Like any pure form of change propagation, it performs well for "deep" changes in an input structure, for instance near the terminals of a syntax tree. This contrasts with the use of memoisation, which performs well for "shallow" changes near the root.

In this paper we present a combination of change propagation and memoisation that works equally well for deep and shallow changes. It takes $\mathcal{O}(\log\ n)$ overheads for a full run, and $\mathcal{O}(\log^2\ n)$ per update operation. The extra space requirements of the combination with memoisation are are a constant factor compared to pure change propagation.

1 Introduction

This paper presents a simple technique for the incremental evaluation of functional programs, without a need for program analysis and without restrictions on the type of input change allowed. Previous work in this area has focused on two separate techniques, namely memoisation and change propagation. The aim of this paper is to demonstrate how these can be profitably combined. To illustrate this aim, let us consider a number of simple algorithms, and see how our technique leads to an incremental version.

Precedence parsing. As a first example, consider a program that implements Floyd's precedence parsing algorithm [10]. It takes a list of numbers, and it turns that list into a heap, that is a tree where the label of each subtree is at most any of its descendants. Furthermore, the in-order traversal of the heap is the original list.

To do this in linear time, one traverses the input list from right to left, maintaining the heap corresponding to the input read so far. To add a new number, we walk up the left-hand spine of the heap, starting at the left-most node, until a node is found whose label is at least the new element. At this point, a new node is created, with the traversed part of the spine as its right-hand descendant. It is easily seen that this is a simplified version of the standard algorithms for bottom-up parsing.

Now consider the incremental evaluation of this program when a few elements near the end of the list are changed. If we employ a standard change-propagation technique, only a few re-evaluations are necessary. By contrast, when using memoisation as the strategy for incremental evaluation, almost the whole list needs to be re-parsed. The situation is reversed when a change is made near the beginning of the list: here memoisation wins.

Given this observation, it is not surprising that on average, either technique will require update times that grow at least linearly with the size of the input. This is illustrated by the upper line in Figure 1, which depicts the running time of change propagation, using Acar *et al.*'s implementation of their own algorithm [2]. This implementation does not use the constant-time data structure, but instead the simpler

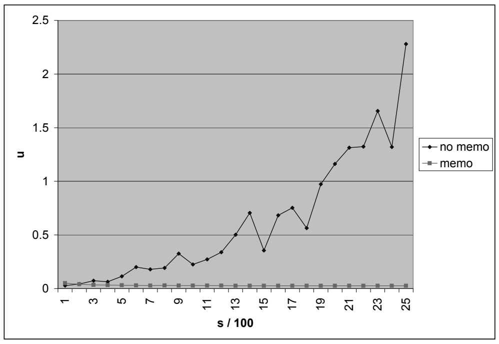

Figure 1: Floyd's precedence parsing.

algorithm from [9] that requires $\mathcal{O}(\log\ n)$ overheads.[1] To produce these numbers, we conducted a number of experiments using Moscow ML, on a 1.7GHz Pentium 4 with 512K RAM, running Windows 2000. We generated random input lists, and then executed 80 random edits (insertions or deletions of consecutive subsequences of a random length between 1 and 32), with a change propagation after each 16 edits. Each such experiment was repeated 10 times, and we took the average of the results.

The horizontal axis shows the size of the inputs s, divided by a 100. On the vertical axis, we plot the quantity $u = e/(d \cdot (\log^2\ s)$, where e is the elapsed time in milliseconds. Furthermore s is the input size, and d is the minimum edit distance between one version of the input and the next. We divide by the minimum edit distance rather than the actual number of edit operations to factor out the effects of overlapping edits; if there are no overlaps $d = 256$. We divide by $\log^2\ s$ because that is the cost of a recomputation step in our method. The quantity u is thus intended to be a measure of the number of recalculations that results from changing (inserting or deleting) one input element.

In the same figure, the lower line displays the same quantity u for the combination of change propagation and memoisation proposed in this paper. This tends towards a constant (0.025), so we may conclude that the cost of an edit action is $\mathcal{O}(\log^2\ s)$. As we shall see later, the memoised version only takes a constant factor of extra space compared to the program without memoisation.

Naturally there is a price to pay for this good behaviour, and this is that the original computation (before any incremental updates) may be slowed down by $\mathcal{O}(\log\ s)$ in the worst case. That compares badly with the change propagation algorithm of Acar *et al.* which has constant-time overheads [2]. As stated before, however, for this draft we used the implementation of Acar *et al.* that also has $\mathcal{O}(\log\ s)$ overheads. This is illustrated in Figure 2, which plots the time taken for the original run. Both implementations are merely a proof of concept, and they have not been tuned in any way, so one should not place too much weight on the difference in constant factors.

There are many good solutions to incremental parsing (in particular [20]), and we do not wish to claim an improvement over these specialised algorithms. Rather, our

[1] The final version of this paper will include numbers for the best known data structure, namely [6].

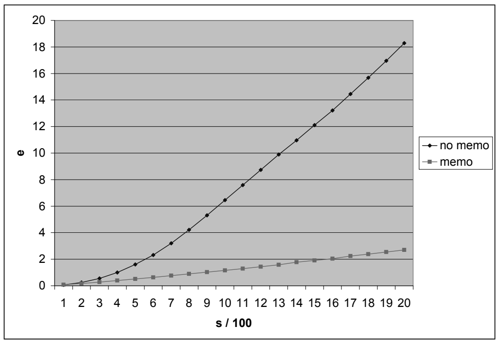

Figure 2: Time for initial run.

aim is to obtain efficient incremental algorithms from simple functional programs, without any need for clever modifications. In particular, the techniques of this paper can be applied by a compiler.

0/1 Knapsack. Let us now turn to a typical dynamic programming problem, namely the 0/1 knapsack. Here we used the algorithm of Ahrens and Finke [4] that maintains a list of potential solutions in strictly decreasing order of value and weight. At each step, a new element is added to all elements of this list, and the infeasible solutions (whose total weight exceeds the total capacity) are filtered out. The new list is then merged with the old list. During the merging, when solution x has higher value but lower weight than y, we discard y. When all elements have been processed, the head of the list has the highest value, so that is returned as the result. Because the list of potential solutions is strictly decreasing in weight, the total running time is never more than $\mathcal{O}(c \cdot n)$, where c is the capacity and n the number of elements.

Figure 3 shows that again we obtain an average update time of $\mathcal{O}(\log^2 s)$. The numbers for this figure were obtained in the same way as those for Figure 1. In all experiments, the capacity was set to $c = 20,000$; only the list of input items was edited, with values and weights varying randomly between 0 and 999.

It is important to stress that in this example, there are two kinds of tabulation going on. The first is the list in the dynamic programming algorithm of Ahrens and Finke. The second is the use of memoisation to improve incremental behaviour. These two are quite different, and the first kind of tabulation is present in both series of Figure 3. In [8] it is shown how many other dynamic programming algorithms can be expressed in a similar style.

Quicksort. In their paper [2] on change propagation, Acar *et al.* illustrate the efficiency of their technique with a program for Hoare's Quicksort, and they prove that the expected time for adding an element at the end of the input is $\mathcal{O}(\log s)$. Change propagation always works better towards the end of the list, so this is the best case, in the sense that edits at random places are likely to be more expensive.

Figure 4 confirms this intuition. Again we performed the same set of experiments, using Acar's code for quicksort. Without memoisation, the cost of an update is superlinear on average, even when divided by $\log^2 s$. By contrast, with memoisation we see sublinear behaviour, giving rise to the hope that the elapsed time per update

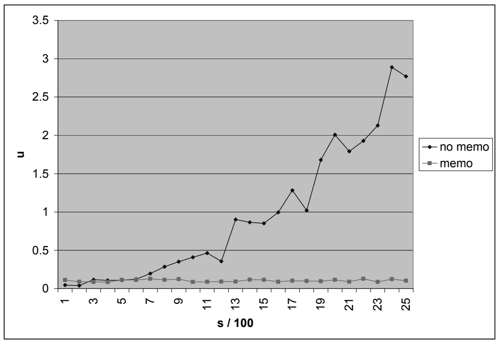

Figure 3: Ahrens and Finke's 0/1 Knapsack.

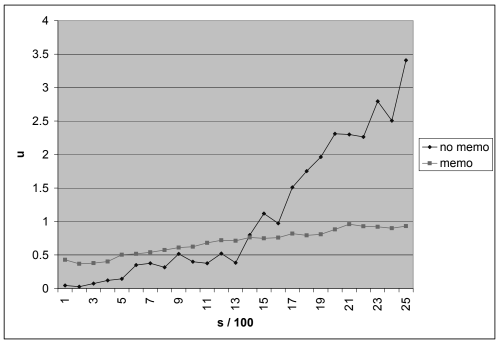

Figure 4: Hoare's Quicksort.

gets close to the $\mathcal{O}(\log^3 s)$. This is the theoretically best result in our setting — $\mathcal{O}(\log s)$ recomputations that each cost $\mathcal{O}(\log^2(s \cdot \log s))) = \mathcal{O}(\log^2 s)$. It would be worthwhile to do a formal analysis along the lines of [2] to verify this behaviour.

Weighted tree depth. We conclude our list of examples by one that computes a property of trees. Consider binary trees with integer values at the leaves. The *weighted depth* of such a tree multiplies the values with the depth at which they occur, and then computes the maximum.

In this case we considered two kinds of edit action: replacement of a subtree by a new tree, and swapping two existing sub trees. The size of new subtrees was selected randomly between 1 and the size of the original tree. As before we performed 80 such edits, with a change propagation after each block of 16. To compute the measure d we used the size of the actual edits, because an appropriate notion of minimum edit distance would be hard to define. The results are shown in Figure 5; tests with the non-memoised version were stopped at $s = 900$ because of excessive time consumption.

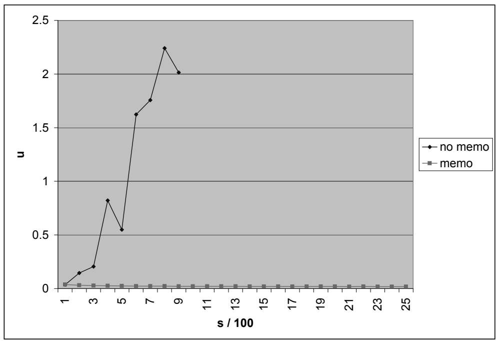

Figure 5: Weighted tree depth.

Overview The remainder of this paper is structured as follows. First we briefly review the change propagation algorithm of Acar *et al.* [2], and in particular its use of time stamps in Section 2. Next, it is shown how this can be extended with a memoisation strategy. This involves a change in the representation of time stamps, which is discussed further in Section 4. We conclude with a discussion of related work, and issues for further investigation.

2 Change propagation

In change propagation, we dynamically maintain a dependence graph, recording which values are used in a computation. The nodes of the graph are values, and the edges represent "read" operations. When A is used to compute B, there is an edge from A to B; we assume the edge is labelled with the relevant operation, so it easy to re-execute.

Clearly, when the value of A is changed, we may need to reconsider the value of B, by re-executing the relevant read operation. It would be sub-optimal, however, to do this in any order, as this can easily lead to the same value being re-evaluated multiple times. Even worse, we may re-evaluate values that are no longer required.

To cope with this ordering problem, it turns out to be necessary to maintain a structure for determining whether one read operation occurred within another. This can be achieved by labelling each edge in the dependence graph with the time the relevant operation was started, and each node with the time the computation of the associated value completed. The time span of a read operation is now defined as (s, t), where s is the start time and t is the finish time of its target. A read operation e' is included in e if the start time of e' falls within the time span of e. Each operation gets its own time stamp, and thus no two reads can have overlapping time spans, unless one is contained in the other.

Of course we do not want to re-label the whole graph with time stamps when doing a small incremental update: it is thus necessary to generate new time stamps in between existing ones. There exists an ingenious algorithm of Dietz and Sleator that allows us to maintain the set of time stamps with constant time comparison and insertion operations [9].

We now summarise the change propagation algorithm. Let E be the set of edges of the dependence graph, and let PQ be a priority queue of edges, ordered by their starting time. When a set of node values has changed, we can propagate the new values as follows:

$PQ := \{ e \in E \mid e \text{ read } v, v \text{ has changed} \};$
while $PQ \neq \{\}$ **do**
  $e := delmin(PQ);$
  $out := \{ e' \in E \mid start(e) < start(e') < end(e) \};$
  $PQ := PQ - out;$
  $E := E - out;$
  $v := val(tgt(e));$
  $run(e)$ in time interval $(start(e), end(e));$
  if $v \neq val(tgt(e))$
  then $PQ := PQ \cup \{ e' \in E \mid e' \text{ read } tgt(e) \}$
end

The priority queue is initialised to all reads of nodes that have changed. As long as the priority queue is not empty, we delete the read e that occurred first from the queue. At this point, all reads that happened in the open time interval $(start(e), end(e))$ become obsolete and they are discarded, both from the priority queue, and from the dependence graph. We now save the value of the target node of e, and the operation corresponding to e is re-executed. If the value changed, all the readers of $tgt(e)$ are added to the queue. Note that each of the priority queue operations takes no more than logarithmic time, so the overheads of change propagation are bounded by $\mathcal{O}(\log X)$ where X is the maximum number of edges in the queue.

2.1 Implementation in ML

To implement these ideas in ML, Acar *et al.* have put forward a library of primitives for writing incremental programs [2]. The main type is that of modifiable references — these are the nodes of the dependence graph, and we write *'a modr* for modifiables that hold values of type *'a*. The type *changeable* denotes a computation that may change the values of modifiables. To create such a modifiable reference, we have the library function

$$\begin{aligned} modr :\ & (('a \times 'a) \to bool) \to \\ & ('a\ dest \to changeable) \to \\ & 'a\ modr \end{aligned}$$

It takes an equality test as the first argument. The second argument is a function that represents the computation of its value. It takes a destination of type *'a dest*, and one can think of this as the address of a modifiable reference. The only way to write to a destination is using the operation

$$write : ('a\ dest \times 'a) \to changeable$$

It remains to show how we can create an edge in the dependence graph. This is achieved with

$$readIn : ('a\ modr \times ('a \to changeable)) \to changeable$$

It takes the reference to read, and the operation to execute upon its value as parameters.

We now illustrate the above definitions by demonstrating how a small program is made incremental. Consider the problem of computing the minimum of a tree of type

```
datatype tree = leaf of int | bin of (tree × tree)
```

The straightforward program for doing this traverses the tree in the obvious way:

```
fun mintree t
=
case t of
    leaf n ⇒ n
  | bin(u, v) ⇒ let val mu = mintree u
                    val mv = mintree v
                in
                  min mu mv
                end
```

To make this an incremental computation, we first define a type of modifiable trees:

```
datatype mtree' = mleaf of int | mbin of (mtree × mtree)
withtype mtree = mtree' modref
```

The corresponding minimum function is then constructed by transliterating the original program, making all the dependencies explicit by writing intermediate results to appropriate modifiables, and accessing these new modifiables with *readIn* operations:

```
fun mintree (t : mtree modref) : int modref
=
let
   fun mintree' (t : mtree modref)
                (d : int dest)
                :
                changeable
   =
   readIn(t, fn t' ⇒
   case t' of
       mleaf n ⇒ write(d, n)
     | mbin(u, v) ⇒ let val mu = modr (op eq) (mintree' u)
                        val mv = modr (op eq) (mintree' v)
                    in readIn(mu, fn a ⇒
                       readIn(mv, fn b ⇒
                       write(d, min a b)))
                    end )
in
  modr (op =) (tmintree' t)
end
```

In addition to the above operations for programming incremental computations, Acar *et al.*'s library also has a number of meta primitives, which are used to control incremental evaluation, but which are not used in the computations themselves. The first of these is

$$change \ : \ ('a\ modref \times 'a) \rightarrow unit$$

This is used to change an input variable. When all such required changes have been made, one calls

$$propagate \ : \ unit \rightarrow unit$$

which triggers the execution of the change propagation algorithm. The resulting value of a modifiable reference can be inspected using

$$deref \ : \ 'a\ modref \rightarrow 'a$$

2.2 Internals of the library

We now present a detailed implementation of the change propagation library in ML, in preparation for the novel features that we shall introduce in the next section. This implementation is due to Umut Acar, who kindly made it available for our experiments.

The type of reads is defined

$$\textbf{type}\ read \ = \ ((unit \rightarrow unit) \times Times.t \times Times.t)$$

In words, a *read* is a triple of an operation (that needs to be re-executed whenever the read is re-evaluated), a start time and a finishing time. Reads are compared by looking at their starting times, and using this comparison we can construct the queue PQ for change propagation.

The type *changeable* is simply an opaque synonym for *unit*, designed to hel enforce correct usage of the library.

A modifiable reference is defined as a triple of a reading operation, an operation for writing the modifiable, and a list of reads that inspected the value of this modifiable:

$$\begin{array}{l}
\textbf{type}\ 'a\ modref \\
= \\
\{\ \begin{array}{lll} rd & : & (unit \rightarrow 'a)\ ref, \\ wrt & : & ('a \rightarrow unit)\ ref, \\ rdrs & : & read\ list\ ref \quad \} \end{array}
\end{array}$$

Because all of these components are in fact reference cells, modifiables can be compared for equality. To write to a modifiable, or to change its value, we simply call its writer field:

$$\textbf{fun}\ write\ (\{wrt, \ldots\} \ : \ 'a\ dest,\ v) = (!wrt)(v)$$

$$\textbf{fun}\ change(l \ : \ 'a\ dest,\ v) = write(l, v)$$

To create a new modifiable, one uses the function *modr*, and the outline of its definition is

$$
\begin{array}{l}
\textbf{fun}\ \mathit{modr}\ \mathit{cmp}\ f \\
= \\
\textbf{let} \quad \begin{array}{l}
\textbf{val}\ \mathit{rd} \quad = \quad \mathit{ref}(\textbf{fn}\,() \Rightarrow \mathit{raise}\ \mathit{unsetMod}) \\
\textbf{val}\ \mathit{wrt} \quad = \quad \mathit{ref}(\textbf{fn}\ v \Rightarrow \mathit{raise}\ \mathit{unsetMod}) \\
\textbf{val}\ \mathit{rdrs} \quad = \quad \mathit{ref}(\mathit{nil}) \\
\textbf{val}\ m \quad = \quad \{\mathit{rd} = \mathit{rd},\ \mathit{wrt} = \mathit{wrt},\ \mathit{rdrs} = \mathit{rdrs}\} \\
\\
\textbf{fun}\ \mathit{update}\ t\ v \ = \ \ldots \\
\\
\textbf{fun}\ \mathit{write}(v) = \ldots
\end{array} \\
\textbf{in} \\
\quad \mathit{wrt} := \mathit{write}; \\
\quad f(m); \\
\quad m \\
\textbf{end}
\end{array}
$$

Note how *rd* and *wrt* are initialised to raise an exception; their value will be adjusted in due course. The interesting parts of the above definition are *update* and *write*. The latter operation is used to define the value of the modifiable when it is first created, while the second is used for subsequent updates during change propagation.

Let us first examine the definition of *write*:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{write}\ v\ = \\
(\ \mathit{rd}\ :=\ (\textbf{fn}\,() \Rightarrow v); \\
\quad \mathit{Times.insert}\ \mathit{currentTime}; \\
\quad \mathit{wrt}\ :=\ \mathit{update}(!\mathit{currentTime})\ \ )
\end{array}
$$

That is, the read operation is initialised to return the new value. Furthermore, a new time stamp is created, just between the current time (a global variable) and its immediate successor; the current time is furthermore set to this new stamp. The *Times* structure manages the list of time stamps. Finally, the *wrt* field of the modifiable is set to $\mathit{update}(!\mathit{currentTime})$, ready for use in the change propagation algorithm.

To complete the definition of $\mathit{modr}\ \mathit{cmp}\ f$, it remains to spell out the details of *update*:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{update}\ t\ v \\
= \\
(\ \textbf{if}\ \ \mathit{cmp}(v, (!\mathit{rd})())\ \ \textbf{then}\ \ () \\
\quad \textbf{else}\ (\ \ \mathit{rd}\ :=\ (\textbf{fn}\,() \Rightarrow v); \\
\qquad\quad \mathit{List.app}\ (\textbf{fn}\ x \Rightarrow PQ\ :=\ Q.\mathit{insert}(x, !PQ))\ \ (!\mathit{rdrs}); \\
\qquad\quad \mathit{rdrs} := \mathit{nil}\ ); \\
\quad \mathit{currentTime} := t)
\end{array}
$$

Let us examine this definition in detail. First we compare the current value of the modifiable (as returned by $(!\mathit{rd})()$) with the old value v. If they are the same, nothing needs to be done. If they differ, however, all the readers have to be enqueued for change propagation.

That list of readers is built up by applications of the *readIn* operation:

```
fun readIn({rd, rdrs, ...} : 'a modref, f)
=
let val start = Times.insert(currentTime)
    fun run() = ( f((!rd)());
                  rdrs := (run, start, (!currentTime)) :: (!rdrs) )
in
  run()
end
```

First a new time stamp is created, just after the current time, and the current time is advanced to that new stamp: this sequence of operations is effected by a call to the primitive operation *Times.insert*(*currentTime*). Then we create the operation that must be re-run whenever this read is invalidated during change propagation: it first runs f (the continuation that is passed as an argument), and then it updates the list of readers.

Finally we arrive at the implementation of change propagation itself, that is the function *propagate*:

```
fun propagate()
=
let val ctime = !currentTime
in
  propagate'();
  currentTime := ctime
end
```

We save the current time, and then do the propagation proper; after propagation is completed, the current time is restored, in case we want to set up new computations. The proper propagation algorithm deletes elements from the priority queue until none remain:

```
fun propagate' ()
=
if Q.isEmpty(!PQ)
then ()
else let val ((f, start, stop), pq) = Q.deleteMin(!PQ)
         val _ = PQ := pq
     in
        if (Times.isSplicedOutstart)
        then propagate'()
        else ( Times.spliceOut(start, stop);
               currentTime := start;
               f();
               propagate'() )
     end
```

If the queue is not empty, we delete its minimum element, that is the read that occurred first. Now if the starting time of that read was previously invalidated, the read is simply thrown away. Note that this is a lazy strategy for discarding reads from the queue; in our abstract explanation of change propagation this was done eagerly. If the starting time of the read is still valid, we invalidate the whole time interval between its start and stop times: everything in between needs to be reconsidered. This is achieved using the primitive *spliceOut*(*start*, *stop*) of the *Times* structure. We re-set the current time to the old start time, and re-run the read operation. This process iterates until the queue is empty.

Let us now summarise the operations in the *Times* structure, which maintains the linear order of time stamps:

$$insert \ : \ t \ ref \rightarrow t$$

generates a new time stamp immediately after the stamp passed in its argument, and the argument is updated to that new stamp. A call to $spliceOut(s, t)$ invalidates all time stamps in the open interval (s, t). Finally, $compare(s, t)$ checks whether s occurs before t in the time stamp order. As mentioned before, these operations can all be implemented in amortised constant time using the order maintenance algorithm of Dietz and Sleator [9]. For the purposes of this paper, the details of that algorithm are not important, because we shall use a different representation of time stamps.

3 Combination with memoisation

To augment the library for change propagation with memoisation, we add a single new primitive, which is a variation on the operation *modr* for creating new modifiables. It is named *tabulate*, and its type is

$$\begin{array}{ll} & ((('a \times 'a) \rightarrow order) \quad \times \quad (('b \times 'b) \rightarrow bool)) \\ \rightarrow & ('a \times ('b \ dest \rightarrow changeable)) \\ \rightarrow & 'b \ modref \end{array}$$

The first argument is a comparison operator for values of type $'a$, and an equality test for values of type $'b$. Given these, *tabulate* produces a memoised version of *modr*. When applied to a pair $(x \ : \ 'a, \ \ f \ : \ 'b \ dest \rightarrow changeable)$, it first checks whether there already is an entry corresponding to x in the table. If there is, we simply return the modifiable that was computed earlier. Otherwise, a new modifiable is computed, stored in the table, and returned.

Of course the difficulty here is that the table manipulation has to interact correctly with the way time stamps are used in change propagation. We shall turn to this question shortly, but let us first consider an example.

Recall the example of computing the minimum of a tree. To take advantage of tabulation, we need to memoise the computation on portions of the list. There exists an operation for comparing modifiable references called *cmpModr* (the actual ordering is not important, as long as there is one). We now define

$$\mathbf{val} \ table \ = \ tabulate(cmpModr, \ \mathbf{op} =)$$

and replace the last use of *modr* (**op** =) in *mintree′* by a call to *table*. This tiny change produces the sort of performance improvements we saw in the introduction.

At first sight, it may appear that this is an erroneous use of memoisation, because mere equality of a modifiable reference does not mean that the tree reachable through this reference is unchanged. We are, however, only memoising part of the dependence graph, and not memoising the resulting value of a computation. Change propagation itself will ensure that any changes within the memoised piece of graph are correctly propagated.

From the above description, it will be clear that any use of the tabulation primitive must obey the rule that the relevant *modr* operation happens only once during a given run of the program. If this restriction was not satisfied, the same piece of dependence graph would conceptually be created at two different times — contradicting the assumption that time is linearly ordered. It also becomes apparent that the table will only hold items that were valid parts of the dependence graph in the previous run of the program: items that are no longer valid are purged from the table after each change propagation. Such purging also guarantees that the space requirements of memoisation are no more than a constant factor.

3.1 Implementation

In order to implement memoisation, we make the following alterations to the earlier program for change propagation.

First, we add a field to the implementation of modifiables, simply to give them a numerical identity, so that we can impose an order relation on them. Second, we add two pieces of global state, namely a mapping RPQ from invalidated time intervals to priority queues, and a list *untablers* that remove invalidated elements from the memo tables upon completion of propagation.

The mapping RPQ is necessary because we may splice out an interval, only to decide later (through a table lookup) that it needs to be spliced in again. When that happens, any elements in the priority queue that were previously marked invalid have to be re-instated.

In fact, it will turn out to be expedient to return to the originally described strategy of eagerly purging the priority queue of invalidated elements. Accordingly, we change the local definition of *update* inside *modr*:

```
fun update t v
=
( if cmp(v, (!rd)())
  then ()
  else ( rd := (fn () => v);
         List.app (fn x =>  if Times.isSplicedOut (startread x)
                            then ()
                            else PQ := Q.insert(x, !PQ))
                  (!rdrs);
         rdrs := nil);
  currentTime := t )
```

The difference with Acar *et al.*'s original program is that we only insert reads that have not been spliced out.

The definition of *propagate* is changed accordingly, to maintain the invariant that there are only valid reads in the priority queue:

```
fun propagate' PQ
=
 if  Q.isEmpty(!PQ)
 then  ()
 else let  val  (fss, pq)  =  Q.deleteMin(!PQ)
           val  (f, start, stop)  =  fss
           val  _  =  PQ := pq
           val  s  =  Times.next start
           val  t  =  Times.prev stop
           val  (PQ', st)  =  Q.split(!PQ, liftTime s, liftTime t)
      in
         PQ  :=  PQ';
         Times.spliceOut(start, stop);
         RPQ  :=  Splaymap.insert(!RPQ, Times.ival s, st);
         currentTime  :=  start;
         f();
         propagate'(PQ)
      end
```

As before, we delete the minimum element from the queue. If the relevant time interval is empty, there is nothing to do. Otherwise, we find the closed interval $[s, t]$ corresponding to the open interval (*start*, *stop*), using the primitives *Times.next* and *Times.prev* that return the immediate successor and predecessor of a time stamp. It is necessary to identify this closed interval because this will be used later to make the correct entry into the domain of *RPQ*. The queue is then split into all the elements that fall in the closed time interval $[s, t]$, and those that are outside it PQ'. The operation *liftTime* lifts a time stamp to a read: it is a type coercion. The old queue is updated to PQ', and thus the invariant about validity of its contents is restored. The invalidated part *st* of the original queue is added to the map *RPQ*. If (part of) the time interval $[s, t]$ is re-used later in a table lookup, we shall reinstate the relevant part of *st*. The operation *Times.ival* returns an identifier for the interval in which stamp s occurs.

The *propagate* function itself needs to perform two further tasks, namely cleaning up the set of time intervals, and purging tables in the process. Furthermore the map *RPQ* needs to be emptied:

```
fun prop PQ
=
let val ctime  = !currentTime
in
  propagate' PQ;
  currentTime  := ctime
end

fun propagate()
=
( prop(PQ);
  let fun untable t  =  List.app  (fn f ⇒ f t)
                                   (!untablers)
  in  Times.commit untable  end;
  RPQ  := Splaymap.mkDictTimes.ivalCompare  )
```

The operation *Times.commit untable* cleans up the data structures that hold time stamps, removing all invalidated time stamps permanently. It is not possible to do such permanent removal in a *spliceOut* operation, because table lookups can cause an invalidated interval to become active again. Any table entries that were recorded during the invalidated intervals are removed at this point.

We now turn to the implementation of *tabulate* itself, which is where most of the hard work is done. Recall its signature

$$\begin{array}{l} \textbf{fun}\ \textit{tabulate}\ (\ \textit{cmpA}\ :\ ('a \times 'a) \to \textit{order}, \\ \qquad\qquad\qquad \textit{cmpB}\ :\ ('b \times 'b) \to \textit{bool}\ ) \\ \qquad\qquad\quad : \\ \qquad\qquad\quad ('a \times ('b\ \textit{dest} \to \textit{changeable})) \ \to\ 'b\ \textit{modref} \end{array}$$

The first argument *cmpA* is used to build the memo table (using a splaymap), and the second is the equality test for change propagation. We create in fact two tables. The first table is just for lookups:

$$\begin{array}{l} \textbf{val}\ \textit{table}\ :\ ('a,\ (\textit{Times.t} \times \textit{Times.t} \times 'b\ \textit{modref})) \\ \qquad\qquad\quad \textit{Splaymap.dict} \\ \qquad\qquad\quad \textit{ref} \\ \qquad = \\ \qquad \textit{ref}\ (\textit{Splaymap.mkDict}\ \textit{cmpA}) \end{array}$$

it maps values of type $'a$ to triples of a start time s, a finish time t and a modifiable that was created during the closed interval $[s, t]$. The second table is an inverse map on the starting times:

$$\begin{array}{l} \textbf{val}\ \textit{table}'\ :\ (\textit{int}, 'a)\ \textit{Splaymap.dict}\ \textit{ref} \\ \qquad = \\ \qquad \textit{ref}\ (\textit{Splaymap.mkDict}\ \textit{Int.compare}) \end{array}$$

Note that in fact the domain of this table is *int* rather than *Times.t*: this is because we want to compare on the *identity* of time stamps, not on their relative order that is returned by *Times.compare*. Given this second inverse index, it is easy to program the *untabler* operation. It takes an invalidated time stamp t as its argument, and seeks to remove that from the table:

$$\begin{array}{l} \textbf{fun}\ \textit{untabler}\ t \\ = \\ \textbf{case}\ \ \textit{Splaymap.peek}(!\textit{table}', \textit{Times.idno}\ t)\ \ \textbf{of} \\ \qquad \textit{SOME}\ m \\ \qquad \Rightarrow\ \textbf{let}\ \ \textbf{val}\ (\textit{ntable}, \_)\ =\ \textit{Splaymap.remove}(!\textit{table}, m) \\ \qquad\qquad\qquad \textbf{val}\ (\textit{ntable}', \_)\ =\ \textit{Splaymap.remove}(!\textit{table}', \textit{Times.idno}\ t) \\ \qquad\quad\ \ \textbf{in} \\ \qquad\qquad \textit{table}\ :=\ \textit{ntable}; \\ \qquad\qquad \textit{table}'\ :=\ \textit{ntable}' \\ \qquad\quad\ \ \textbf{end} \\ \ \ \mid\ \ \textit{NONE} \\ \qquad \Rightarrow\ () \end{array}$$

Here *Times.idno* returns the numerical identity of a time stamp; it is used for comparison purposes only.

Inserting an element into the tables structure is equally easy, but care has to be taken that the interval recorded is in fact closed rather than open. As before, this is achieved using the primitive *Times.next*, which returns the immediate successor of a time stamp:

```
fun compute (x, f)
=
let val start  = !currentTime
    val m      = modr cmpB f
    val start' = Times.nextstart
in
  table := Splaymap.insert( !table,
                            x,
                            (start', !currentTime, m));
  table' := Splaymap.insert( !table',
                             Times.idno start',
                             x)
  m
end
```

We now have all the ingredients to define the crucial part of *tabulate*, namely the *lookup* function that is returned as its result. As in any form of memoisation, *lookup* inspects its table to see whether the relevant entry is there; if it is not, it needs to be computed, and stored in the table:

```
fun lookup (x,f)
=
case Splaymap.peek(!table, x) of
    NONE ⇒ compute(x,f)
  | SOME (start, stop, y) ⇒ ...
```

The interesting part is indicated by the elipses above: here we need to re-use the previous computation of y. There are two possibilities to consider here. First, the time interval may already have been invalidated: in that case, the defunct interval needs to be spliced back in. Alternatively, it has not yet been invalidated — but then it would be invalidated later, otherwise the linearity restriction on the use of *tabulate* would be violated by the client program.

Let us first examine the case when the $[start, stop]$ has been invalidated. We need to update the map of invalidated queues RPQ:

```
let val ival       = Times.ival start
    val (RPQ', pq) = ((Splaymap.remove(!RPQ, ival))
                     handle NotFound ⇒ (!RPQ, Q.empty))

    val (pq', qd)  = Q.split( pq,
                              liftTime start,
                              liftTime stop)

    val remainder  = Times.spliceIn( currentTime,
                                     start,
                                     stop)
in
  RPQ := Splaymap.insert(RPQ', remainder, pq');
  prop(ref qd);
  y
end
```

That is, we first find an identifier for the interval in which *start* occurs, and then determine the priority queue that contains any reads that need to be re-considered. That priority queue is split accordingly, into pq' (those reads that fall outside the closed interval $[start, stop]$) and qd (reads that occurred during $[start, stop]$). Next, the interval $[start, stop]$ is inserted into the active time line, immediately after $currentTime$. Whatever is left over after $[start, stop]$ is removed from its containing interval is called $remainder$. The pair $(remainder, pq')$ is put back into the RPQ map. Finally, we call the propagation algorithm with the queue qd. This is to ensure that upon return of the modifiable y, it actually holds a valid value. If we instead merged qd into the existing priority queue PQ, the result might be suboptimal, as subsequent read operations on y could get the wrong value, causing unnecessary re-executions to happen.

We now turn to the case where an entry $(start, stop, y)$ has been found in the table, but the stamp $start$ has not yet been invalidated. In this case, we split the queue as before, and move the closed time interval $[start, stop]$ from its present location to just after the current time; and the current time is set to $stop$. Finally we propagate with the queue of reads that need re-execution during $[start, stop]$:

```
let val (PQ', qd) = Q.split( !PQ,
                             liftTime start,
                             liftTime stop)
in
   Times.move(currentTime, start, stop);
   PQ := PQ';
   prop(ref qd);
   y
end
```

4 Time stamps

Now that we have outlined the main data structures and algorithm for the combination of memoisation and change propagation, it is necessary to examine the structure $Times$ in some more detail. It defines two abstract types, namely the type t of time stamps and the type i of time intervals. Intervals represent sequences of time stamps. One interval is said to be the *active* interval; this contains the time stamps that are still valid in the dependence graph.

Each time stamp belongs to a unique interval, which is returned by the function $ival$. The only way to create a new time stamp is to insert it immediately after an existing stamp, using

$$insert \;:\; t\ ref \to t$$

The argument of $insert$ is updated to contain the new stamp, and this is also the value returned by $insert$. Note that we have

$$ival(insert\ t) \;=\; ival\ t$$

In our programs, we only call $insert$ on the active interval.

The comparison operator $compare(s, t)$ can only be applied to time stamps that are in the same interval, and it tests whether s precedes t in that interval.

The operator $spliceOut(s, t)$ takes two stamps that occur in the active interval, and it cuts out the open interval between s and t. The intervals that are spliced out in this way are placed in a list, so that when $Times.commit$ is called, we can re-examine each of the intervals to determine whether they should be completely removed.

The operator *spliceIn*(x, s, t) has the opposite effect, namely to take a reference x to a time stamp in the active interval, and to paste the sequence from s to t inclusive immediately after $!x$. The reference itself is updated to t. Here $[s, t]$ is cut out of its surrounding interval, and what remains is still marked for permanent removal upon execution of *commit*. It is a requirement that s and t occur in the same interval, and that this is not the active interval.

The final time manipulation that deserves highlighting is *move*(x, s, t). This is very much like *spliceIn*, but $[s, t]$ is in fact part of the active interval; all that *move* does is to change the position of $[s, t]$, to immediately follow $!x$.

As we have seen, upon completion of change propagation the function *commit finalise* is called. This goes through the list of spliced out intervals, and it removes them from the set of intervals that may be manipulated. Furthermore, on each of the constituent time stamps t, the function *finalise* t is called. This is to perform clean up operations, namely to purge the memo tables of invalidated elements.

There is a simple implementation of the above operations, namely as height-balanced binary trees, with time stamps at the leaves and no data at internal nodes. All the relevant operations can be carried out in logarithmic time; in particular, to compare two time stamps, we merely need to find the nearest common ancestor. Because there is no data at internal nodes, two trees can be concatenated in time proportional to their height difference.

5 A worked example

Again let us return to the example of computing the minimum of a tree. In Figure 6, a sample tree is shown, along with the dependence graph that was produced during its first run. The ovals depict modifiables that are internal nodes of the original tree, whereas the rectangles are leaves. The diamonds are integer modifiables that are constructed during the computation of the minimum. Each modifiable is labelled with the time stamp when it was first written. Each read is labelled with a letter indicating what read operation it represents in the program: A is the first occurrence of *readIn*, B the second and C the third. Reads are also labelled with their starting time. Below we shall refer to modifiables that are tree nodes as t_i, where i is a time stamp, and similarly to minima as m_i.

Our aim is to trace through change propagation, after the following updates have been made:

let

  val t_9' $=$ *deref* t_9
  val t_4' $=$ *deref* t_4
  val t_7' $=$ *deref* t_7

in

  change(t_5, t_9');
  change$(t_9,$ *mleaf* $45)$;
  change(t_7, t_4');
  change(t_4, t_7');
  change$(t_3,$ *mleaf* $98)$

end

It is easy to read off from the dependence graph that there are five reads that will be queued as a result of these changes, namely

$$(A, 13, 20), (A, 16, 17), (A, 21, 22), (A, 26, 33), (A, 27, 28)$$

When the first element is taken off the queue, we splice out the open interval $(13, 20)$. This causes $(A, 16, 17)$ to be stored in RPQ (as a singleton heap). The re-execution of $(A, 13, 20)$ is a simple affair, as we replaced its source by *mleaf* 886.

In the next iteration of propagation, we reexecute the read $(A, 21, 22)$. Now because tree node 5 is overwritten with *deref* t_9, this triggers a table lookup for the creation of the m_{28}. The relevant interval $[27, 28]$ has not been spliced out yet, so we move it directly behind the current time, which is 21 (because we are in the process of re-executing $(A, 21, 22)$). As a result, the active time line now consists of

$$1, \ldots, 13, 20, 21, 27, 28, 22, 23, 24, 25, 26, 29, \ldots, 36$$

> Now because $(A, 27, 28)$ is also on the priority queue, we have to do a recursive propagation, with the singleton queue $[(A, 27, 28)]$. This triggers another table lookup, this time for the creation of m_{15}. The time interval is $[14, 15]$; there are no relevant items for this interval in RPQ, so there is no recursive propagation in this case. The re-execution of $(A, 27, 28)$ does however also trigger the table lookup for m_{17}, and this time there is an element of RPQ that must be dealt with, namely $(A, 16, 17)$. At this point the active time line is
>
> $$\begin{aligned}1, \ldots, 13, 20, 21, 27, 14, 15, 16, 17, 28,\\ 22, 23, 24, 25, 26, 29, \ldots, 36\end{aligned}$$

After these recursive excursions, we are back at the re-execution of $(A, 21, 22)$, and a second table lookup is made, namely for m_{30}. Since $(29, 30)$ has not been spliced out, it is moved immediately behind the current time, which is 27. The priority queue now consists of

$$(B, 23, 25), (C, 24, 25), (A, 26, 33), (B, 31, 33)$$

From this point onwards, we apply regular change propagation, successively re-executing $(B, 23, 25)$, $(A, 26, 33)$, and $(B, 34, 36)$.

6 Complexity Analysis

The use of time stamps imposes an extra $\mathcal{O}(\log\ n)$ factor on a full run of a computation of length n, because $Times.insert()$ requires $\mathcal{O}(\log\ n)$ work. As the memo tables are implemented via splay maps, these incur the same overheads. We conclude that the cost of a full run of length n becomes $\mathcal{O}(\log\ n \cdot n)$.

For an incremental update, there are the overheads of managing the time stamps, the priority queue PQ of reads, the table RPQ of discarded sections of PQ, and of course the memo table lookups. The time stamps, RPQ and the memo tables are all easily seen to account for a factor of $\mathcal{O}(\log\ n)$; the interesting part of the analysis concerns the priority queue.

The priority queue is implemented with a balanced binary tree, whose size is obviously bounded by n. Because comparison of time stamps takes $\mathcal{O}(\log\ n)$, the

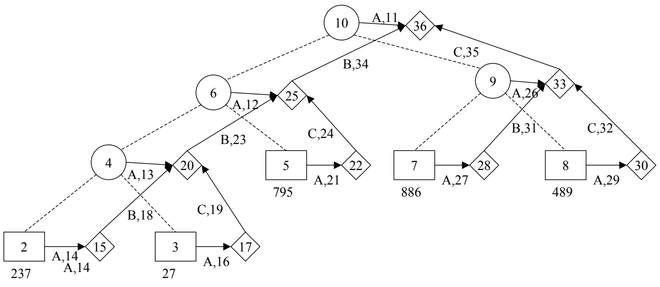

Figure 6: Dependence graph

cost of insertion and deletion is $\mathcal{O}(\log^2 n)$. The operation $(pq, pq') = \mathit{split}(!PQ, s, t)$ is used to separate the priority queue into those elements pq that are outside the closed interval $[s, t]$, and those pq' that are in it. We implement this operation by first computing a list xs of all the elements in the queue that fall inside $[s, t]$, deleting the elements of xs from the queue to obtain pq, and then building a new queue pq' from xs. The cost of computing xs is $\mathcal{O}(\log^2 n + m)$, where m is the length of xs. The computation of pq then takes $\mathcal{O}(m \cdot \log^2 n)$, and building the new queue pq' takes $\mathcal{O}(m)$.

During a run of *propagate*, a read can be taken out of the queue at most once via *split*, and then it can be put back in at most once via *split* during a table lookup. The cost of these splits can thus be amortised over all the reads that are moved out (and possibly back in) of the queue during the run, accounting for $\mathcal{O}(\log^2 n)$ per read.

We conclude that the overheads per re-executed read are $\mathcal{O}(\log^2 n)$.

7 Related work and discussion

The result presented here is just a small step from the pioneering papers by Acar, Blelloch and Harper on change propagation [2] and memoisation [3]. Indeed, in these works it is already suggested that the two techniques might be profitably combined — but no indication is given how such a combination might be achieved. One could argue that we have not quite managed a full integration, because our use of memoisation imposes the linearity constraint, that each table entry is accessed only once in the course of a computation. This precludes the use of tabling for the usual purpose of dynamic programming — that kind of tabling has to be encoded first in the purely functional program before it can be made incremental. The knapsack example from the introduction illustrates this point.

There is a vast body of literature on incremental evaluation that precedes these developments. Ramalingam and Reps present a categorised bibliography in [16]. Below we shall structure our discussion following their categories.

Assessment. At present we only have a complexity analysis for the overheads incurred per re-executed read, namely $\mathcal{O}(\log^2 n)$. It would obviously be desirable to have a systematic way of analysing the number of re-executed reads itself, but at present we lack the tools to do so. In [1] such a framework is presented for change propagation, where a formal notion of computation trace is introduced, and it is shown how that the number of re-executed reads is equivalent to an appropriate notion of edit distance between traces. To obtain a similar result, we would need a modify their definition of edit distance to cover our use of tabulation, with a richer set of edit actions that includes movement of existing subtraces.

An incremental algorithm is said to be *bounded* if the time it takes to update the output solution only depends on the size of the change, and not on the total computation time. Like the original change propagation of [2], our method is not bounded in general. However, in many applications (for example Floyd's precence parsing algorithm) the size of the queue is bounded by a constant. If the operations on time stamps take constant time, that leads to a bounded algorithm using [2]. Unfortunately, at present we do not know how to implement the necessary operations on time stamps to obtain bounded algorithms for our method. It may be possible to do so using a dynamic solution for nearest common ancestor queries [5], but we have not yet investigated this in detail.

While the slopes of the graphs at the beginning of this paper are encouraging, we must add a word of caution about the constant factors involved. They are quite large; for instance in the knapsack example, the time for the original run is a factor of $180 \cdot \log n$ more than for the non-incremental version. As we mentioned before, however, our implementation is merely a proof of concept, and we expect vast improvements are possible. For the moment, however, our techniques are most likely to be useful when the operations saved are quite costly (as for example in [11]). The dominating cost is that of managing time stamps. Apart from improving the data structure for time stamps, we can identify two other venues for improvement:

One possibility would be to process the input at a coarser level of granularity, partitioning it into blocks that are processed using the ordinary static algorithm. It is unlikely, however, that this can be done automatically for a large class of programs.

Another option may be to transform our use of dynamic dependencies into static dependencies where possible, using partial evaluation [13]. Previous experience [19] suggests that Carlsson's formulation of [2] in terms of *arrows* would be a suitable starting point for that line of inquiry [7]. The techniques put forward by Liu are also likely to be useful in making such a move from dynamic to static dependencies in particular applications [14]. If all dynamic dependencies can be made statical, we end up with a version of Reps' change propagation, discussed below.

Data structure update problems. Undoubtedly the best developed application of incremental updates is that of attributed derivation trees. In a seminal paper, Reps, Teitelbaum and Demers set out to show how change propagation can be applied to this problem [17]. The notion of change propagation is however subtly different than that of [2]: first of all, it uses static dependencies between values rather than dynamic ones. Consequently, there is no notion of *containment* between read operations, and thus no notion of time stamps. Instead, to ensure that reads are re-executed in the right order, the invalidated edges are processed in topological order. Evidently there are less administrative costs in this version of change propagation; but due to the use of static dependencies and the absence of containment, it is possible that more reads are re-executed. In order to efficiently maintain the dependence graph and its topological sorting, there is an additional cost in moving a *cursor* round the tree; changes can only be made at the position of this cursor. By contrast, [2] is completely unconstrained in the changes that can be made to the input, and the same applies to our method.

Memoisation has also been used in this context, starting with the work of Pugh and Teitelbaum [15]. A key contribution of this work was to improve the input data structures so that memoisation would be more effective. In particular, it is argued that *long stringy trees* (often representing lists) should be avoided in favour of balanced tree structures, because they lead to particularly bad behaviour of memoisation.

Incremental Formal Systems. It would be of interest to compare the asymptotic performance of our techniques on various kinds of parsers. The results for precedence parsing reported earlier in this paper give good reason to believe that we may be able to do almost as well as [20] when incrementalising general LR parsers expressed in a purely functional style. It remains to be seen whether top-down parsers can be similarly improved.

Implementation frameworks. In collaboration with Joao Saraiva, we are in the process of constructing a simple backend for the attribute grammar system *lrc* that makes use of the techniques described here, to gain a better understanding of the differences with change propagation as proposed by Reps. Such an implementation would also allow us to make a comparison with other methods of incremental reevaluation such as [18] that are solely based on memoisation.

It would be of great help in further experimentation to have a compiler that takes pure ML programs and produces the corresponding incremental code in continuation-passing style. It is our belief that it is easy to write such a compiler, although the best use of tabulation may at times require further analysis, of the kind in [14]. The result would be a system in a long tradition, including Alphonse [12]. In his design for Alphonse, Hoover did in fact propose a combination of Pugh's memoisation and change propagation with a dynamic dependence graph. That combination is simpler than ours because of the absence of time stamps. Unfortunately there are no performance numbers in [12] for comparison against ours.

Acknowledgements

We would like to thank Umut Acar and Magnus Carlsson for interesting discussions about change propagation. Umut made the code of his library available for our experiments. Ken Friis Larsen suggested the use of height-balanced trees to represent time stamps. The Programming Tools group at Oxford provided helpful feedback at all stages of this work.

References

[1] Umut Acar, Guy E. Blelloch, Robert Harper, Jorge Vittes, and Shan-Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA04)*, pages ?–?, New Orleans, LA, January 2004. ACM.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Principles of programming languages*, pages 247–259. ACM Press, 2002.

[3] Umut A. A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Principles of programming languages*, pages 14–25. ACM Press, 2003.

[4] J. H. Ahrens and G. Finke. Merging and sorting applied to the 0-1 knapsack problem. *Operations Research*, 23(6):1099–1109, 1975.

[5] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264. ACM Press, 2002.

[6] Michael A. Bender, Richard Cole, Erik D. Demain, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *10th Annual European Symposium on Algorithms*, volume 152–164 of *Lecture Notes in Computer Science*, 2002.

[7] Magnus Carlsson. Monads for incremental computing. In *International conference on Functional programming*, pages 26–35. ACM Press, 2002.

[8] O. De Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[9] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372. ACM Press, 1987.

[10] Robert W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM (JACM)*, 10(3):316–333, 1963.

[11] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Programming language design and implementation*, pages 311–320. ACM Press, 2000.

[12] Roger Hoover. Alphonse: incremental computation as a programming abstraction. In *Programming language design and implementation*, pages 261–272. ACM Press, 1992.

[13] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[14] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):546–585, 1998.

[15] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Principles of programming languages*, pages 315–328. ACM Press, 1989.

[16] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Principles of programming languages*, pages 502–510. ACM Press, 1993.

[17] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):449–477, 1983.

[18] J. Saraiva, S. D. Swierstra, and M. F. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction*, pages 279–294, 2000.

[19] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Second International Summer School on Advanced Functional Programming*, volume 1126 of *Lecture Notes in Computer Science*, pages 184–207. Springer Verlag, 1996.

[20] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):980–1013, 1998.