*Technical Report*

UCAM-CL-TR-197
ISSN 1476-2986

Number 197

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# The semantics and implementation of aggregates

# or

# how to express concurrency without destroying determinism

Thomas Clarke

July 1990

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*https://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*https://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# The Semantics and Implementation of Aggregates
# or
# How to express concurrency without destroying determinism

Thomas Clarke*
University of Cambridge Computer Laboratory
E-mail: tjwc@uk.ac.cam.cl

July 24, 1990

**Abstract**

**Keywords: aggregate, non-determinism, parallel functional programming, single assignment language.**

**This paper investigates the relationship between declarative semantics and concurrent computation. A fundamental programming construction, the aggregate, is identified. Aggregates have a simple declarative semantics yet cannot be written in pure functional languages. The addition of aggregates to a functional language increases expressiveness without destroying determinism or referential transparency. Specific aggregates can be used to implement concurrent graph marking, time deterministic merge of lazy lists, and write-once locations.**

## 1 Introduction

One motivation for the study of functional languages is that they allow great freedom in evaluation order, and so can express concurrency naturally. It is thus surprising to find, on further study, that pure functional languages cannot express some types of concurrent programs. Why do we have to resort to impure functions with operational semantics in order to implement efficient concurrent programs? One answer might be: because the program is indeterminate. However, many of the concurrent programs we wish to write do have determinate solutions and still cannot be represented in functional languages. For example the concurrent graph marking algorithm described on page 3 does not have indeterminate semantics and yet cannot be written within a pure functional language. The work described here arises from an analysis of this problem.

The main contribution of this paper is the identification of a class of computations, called *aggregates*, which are globally determinate and yet have a concurrency which cannot be adequately represented in functional languages. We will see how aggregates can be implemented with optimal concurrency, and that a functional language augmented with aggregates can express declaratively the concurrent graph marking algorithm.

*The author thanks Queens' College, Cambridge, for a Research Fellowship which supported this work.

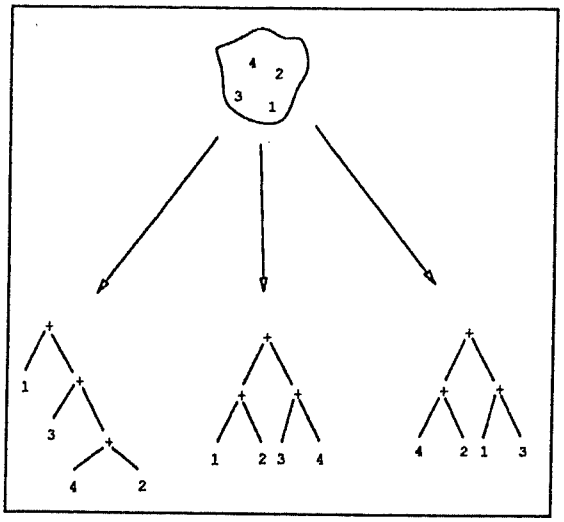


Figure 1: Different aggregate evaluation trees

## 2 Concurrent evaluation of associative and commutative operations

In imperative languages it is usual for concurrent processes to communicate through the update of shared data structures. The communication can be represented by a set of calls to constructor procedures, each of which performs some update to the shared structure. Consistency is preserved by a set of interlocks between these constructors.

Is there any way to represent a similar style of global communication in a functional language, without sacrificing the simple semantics and freedom from extraneous evaluation order constraints which make functional programs so attractive?

Consider an associative commutative operator, for example plus. A concurrently specified collection of integers can be aggregated with such an operator to give a result which is independent of the order in which elements are aggregated. Figure 1 illustrates this. The commutativity and associativity of the operator ensures that any possible evaluation tree with nodes representing operations, and one leaf for each element, will give the same result.

This *aggregate* performs a type of global communication, by creating a single data structure from concurrently evaluated expressions in such a way that no extra constraints on evaluation order are introduced. In a functional language aggregates must be represented by a fixed tree of aggregate operations; the result of the root operation is the value of the aggregate. This representation contains extra (operational) information beyond that required to determine the aggregate value, since it specifies one particular evaluation tree.

Within an aggregate calculation the implementation is free to choose dynamically the tree which allows the most concurrency. The intermediate values of the computation are then indeterminate. Nevertheless any strict aggregate operation has the same result whichever evaluation tree is chosen, so that aggregates have a semantics identical to that of the functional (fixed evaluation tree) operation.

The possibility of dynamically choosing evaluation trees makes aggregate computation more efficient and more concurrent than its functional equivalent. The concurrency derives

from the freedom of evaluation order, because elements may be aggregated in whatever order they are determined. In a multiprocessor, elements may be aggregated separately on each processor, to reduce communication costs.

The histogram problem demonstrates the greater efficiency of aggregates. Suppose you are given a tree of integers with $n$ leaves and asked for a histogram recording the number of instances of each integer in its leaves. The integers lie in the range [1–c]. A naive functional solution to this problem, merging arrays implemented by binary trees, will perform $n$ merges of trees of size $c$ and so use $O(cn)$ operations. A better functional solution, using sparse arrays, could reduce this to $O(n \log n)$ operations at the cost of a higher management overhead.

The computation can instead be done with an array of sum aggregates, as will be described in Section 6. The number of operations needed is exactly $n$. This is possible because trivial additions, with 0 as one argument, need not be performed. The position of the non-trivial operations depends on data, so a functional program with an evaluation tree large enough to include all possible operations cannot avoid redundant computation.

We may incorporate an aggregate in a functional language as an abstract datatype with an aggregating operation $\diamond$, and pack and unpack functions. Every value of this datatype can be thought of as a concurrently constructed multiset of elements, combined with $\diamond$. A functional program specifying a tree of $\diamond$ operations to be evaluated program defines such a multiset, and intermediate results of the combination can be protected from inspection by the datatype, allowing the implementation to use the most efficient (non-deterministic) evaluation strategy to find the result.

This abstract datatype protects a non-deterministic implementation from the view of the outside world, but it is still semantically trivial. An intelligent compiler could make use of the implementation without the programmer needing to know that this was being done.

The aggregate encapsulates *incremental* data construction, so an interesting question arises. Is it ever reasonable to obtain information from the value of the aggregate before construction is complete? The order in which elements are added to the aggregate is not defined, so it would appear that any such examination must result in an indeterminate value. In fact this is not always the case, as we will see in the following example.

Consider a concurrent mark operation which constructs the set of nodes in a directed graph $\langle V, E \rangle$ reachable from some root set of nodes $R$. The set of edges $E$ defines a function $next: V \to \wp(V)$ which returns, for any node $v$, the nodes immediately reachable from $v$. $next$ can be extended to a function on subsets of $V$ in the normal way:

$$next(\{x\}) = next(x)$$

$$next(A \cup B) = next(A) \cup next(B)$$

The set of reachable nodes

$$M = R \cup next(R) \cup next^2(R) \ldots = next^*(R)$$

can then be defined recursively as a smallest set satisfying:

$$M = R \cup close(M)$$

where $close(M) = M \cup next(M)$.

This equation looks similar to that for a recursively defined lazy list, however $M$ is a set and the order in which elements of M are computed need not be fixed. We have here

```
Procedure mark_all(R)=
Begin
  Foreach x In V Do marked[x] := false;
  Foreach x In R Do mark(x);
End

Procedure mark(v)=
Local c;
Begin
  Begin_atomic_section
    c:=marked[v];
    marked[v]:=true;
  End_atomic_section
  If Not c Then
    Foreach x In next(v) Do mark(x);
End
```

Figure 2: Concurrent procedural mark algorithm

an aggregate in which the partially constructed aggregate value is used to direct further construction. It could be solved iteratively using *close*:

$$M := R$$
$$\textbf{while}\ M \neq close(M)$$
$$\textbf{do}\ M := close(M)$$

A more efficient implementation of this follows from the observation that, as in Dijkstra's least path algorithm,, nodes which already been marked do not need to be remarked:

$$M := R$$
$$N := next(R)$$
$$\textbf{while}\ N \neq \emptyset\ \textbf{do}$$
$$\textbf{begin}$$
$$\quad M := M \cup N;$$
$$\quad N := next(N) - M$$
$$\textbf{end}$$

This is equivalent to the procedural concurrent mark algorithm shown in Figure 2, in which **Foreach** is a parallel set mapping operation. When written procedurally care must be taken to interlock the atomic section of code with a separate semaphore for each array element, so allowing maximum concurrency.

Our aim in using aggregates is for a recursive aggregate definition to have an implementation which is identical to the maximally concurrent procedural algorithm, whilst preserving a simple denotational semantics.

We may understand the semantics of set aggregate recursion by noticing that as the algorithm progresses the set $M$ is an increasing lower bound to the answer. The value of $M$ can be interpreted as a semantic approximation $[\![M]\!]$ to the real answer, with:

$$M' \subseteq M \Leftrightarrow [\![M']\!] \sqsubseteq [\![M]\!]$$

The semantic order $\sqsubseteq$, representing increasing knowledge about the value of $M$, is identical to set inclusion. Adding elements to an aggregate corresponds to knowing more about its denotation, and the least solution of a recursive aggregate equation can be found by iterating until the result is stable[1].

Aggregate recursion differs from normal functional recursion. Consider a pure function $f$ defined by:

$$f = h\, f$$

The denotation of $f$ is:

$$[\![f]\!] = \bigsqcup_{i \geq 0} [\![h]\!]^i(\bot)$$

so $h$ is analogous to the function *next*. However we never work out $[\![f]\!]$ directly, since the computation would in general not terminate, instead we compute $f(n)$ for some fixed $n$, working out only those values $f(x)$ necessary to construct the answer.

Suppose now that $f : A \to B$ and $A$ is finite. We wish to determine the entire function $f$. This could be done by computing a data object $\boldsymbol{f}$ representing $[\![f]\!]$ from $[\![h]\!]$. Let $\boldsymbol{f}$ be of type $Den = (A * B)\ Set$, so that the $[\![f]\!]$ is specified by a set of of $(argument, value)$ pairs. The denotation $h$ can be written as a function $\boldsymbol{h} : Den \to Den$. Then $\boldsymbol{f}$ can be found from $\boldsymbol{h}$ by the iteration:

**Input:** $\boldsymbol{h} \equiv [\![h]\!] : Den \to Den$

$\boldsymbol{f} := \boldsymbol{a} := \boldsymbol{h}\, \emptyset$;
**While** $\boldsymbol{a} \neq \emptyset$ **Do**
**begin**
  $\boldsymbol{a} := (\boldsymbol{h}\, \boldsymbol{a}) - \boldsymbol{f}$;
  $\boldsymbol{f} := \boldsymbol{f} \cup \boldsymbol{a}$;
**end**

**Output:** $\boldsymbol{f} \equiv [\![f]\!] : Den$

This is an operation similar to aggregate recursion, and it gives a programmer the chance to inspect objects equivalent to denotations. This provides more information about $f$ than is available using functional recursion, in particular partial functions can be constructed and inspected.

Aggregates are more flexible than this because tailor-made domains may be chosen over which to do recursion. However this algorithm shares an implementation problem with aggregate recursions—it exhibits concurrency which cannot be expressed efficiently in the $\lambda$-calculus.

This motivates the study of aggregates not only for efficient implementations but also to express new semantics. In the next sections we will see how both cases can be dealt with uniformly by defining abstract datatypes which hide the effect of indeterminate evaluation orders.

# 3 Aggregates

## Simple Aggregates

**Defn. 1** *A* simple aggregate *over a type $T$ is an encapsulated datatype $T'$ with operations:*

[1] That is, if it exists. Section 4 will give conditions for least solutions to exist.

$$\diamond_{T'} : T' \times T' \to T'$$
$$\iota_{T'} : T'$$
$$in_{T'} : T \to T'$$
$$out_{T'} : T' \to T$$

*which satisfy the conditions:*

$$\forall x, y \in T'.\, x \diamond_{T'} y = y \diamond_{T'} x \qquad \textit{(commutativity)}$$
$$\forall x, y, z \in T'.\, x \diamond (y \diamond_{T'} z) = (x \diamond_{T'} y) \diamond z \qquad \textit{(associativity)}$$
$$\forall x \in T'.\, \iota_{T'} \diamond_{T'} x = x \qquad \textit{(identity)}$$
$$out_{T'} = in_{T'}^{-1}$$

The last requirement ensures that the map from $T$ to $T'$ is bijective and $T'$ can be identified naturally with $T$. We therefore say that a simple aggregate $T'$ is defined by a triple:

$$\langle T, \diamond : T \times T \to T, \iota : T \rangle$$

since from these $T'$, $out_{T'}$, $in_{T'}$ can be inferred.

A value of an aggregate over $T$ is a function of the non-empty multiset of its constituent values in $T$, each introduced through $in_{T'}$ and combined with $\diamond_{T'}$. The function can be extended by mapping $\emptyset$ onto $\iota$, so that any multiset of elements can be aggregated. This uniformity will prove helpful when we consider aggregate implementation strategies in Section 5. We will call the individual values which are aggregated to form a particular aggregate value its *elements*: the elements of an aggregate value are the leaves of the tree of $\diamond_{T'}$ operations whose root returns the value.

Throughout this paper we will be concerned with aggregates $T'$ which are defined from types $T$, with various permitted operations defined on $T'$ from corresponding operations on $T$. We use the same symbol for an operation on $T$ and $T'$, giving the type as suffix whenever confusion might arise. Similarly the suffixes on $in_{T'}$, $out_{T'}$ and $\iota_{T'}$ will be omitted when the type is clear from context.

In the case of simple aggregates the semantics of $T'$ is identical to that of $T$, although its implementation is different. In order to describe aggregate recursion we will define a new type of aggregate: an encapsulated datatype $T'$ with semantics which is different from its defining type $T$.

## Early Readable Aggregates

An *Early Readable Aggregate* (ERA) is an aggregate from values of which information can be read before construction is completed. In order for this information not to contain evaluation-order dependent non-determinism it is necessary to restrict this reading. Let $T', \diamond$ be the type and operation of a simple aggregate. Suppose there exists a partial order $\leq$ on $T'$ such that $\forall x, y \in T'\, x \leq x \diamond y$. With respect to this the aggregate value is monotonic increasing as extra aggregate elements are included, so lower bounds may be established for an aggregate value which is only approximately known. We then hope to define a domain for $T'$ which includes values defined only by lower bounds.

Given two aggregates equipped with partial orders, $S'$ and $T'$ we will wish, in order to write recursions, to be able to define arbitrary terminating functions $f : S' \to T'$ which preserve $\leq$, so that

$$\forall x, y \in S'.\, x \leq_{S'} y \Rightarrow f(x) \leq_{T'} f(y).$$

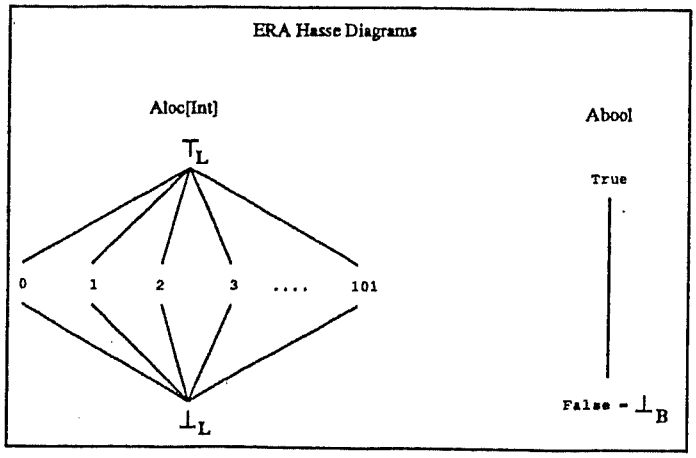


Figure 3: ERA partial orders

We will represent this possibility as a constructor function $\theta : (S \rightarrow_M T) \rightarrow (S' \rightarrow T')$: Where the type $S \rightarrow_M T$ contains all total functions $S \rightarrow T$ which preserve $\leq$ as above.

This constructor could not be written as a high order function in a conventional functional language since the set of all monotonic functions $S \rightarrow T$ cannot be represented within a decidable type system. We will use it to introduce extra functions on ERAs such as the array operations described in Section 6. In a proof system such as HOL ([Gor87]) the set of all monotonic functions $S \rightarrow T$ could be defined, and this constructor used to introduce user-defined functions.

This is summarised by the following definition:

**Defn. 2** *Suppose the type $T$ has a flat domain, and:*

$\langle T, \diamond, \iota \rangle$ defines a simple aggregate.
$\leq : T \times T \rightarrow Bool$ is a partial order on $T$.
$\forall x, y \in T.(x \leq x \diamond y)$.

*The ERA defined by $\langle T, \diamond, \iota, \leq \rangle$ is an abstract datatype $T'$ which has the operations $\diamond_{T'}$, $\iota_{T'}$, $in_{T'}$, $out_{T'}$ satisfying the conditions for a simple aggregate together with new operations:*

$$\leq_{T'}: T' \times T' \rightarrow Bool$$
$$\theta : (S \rightarrow_M T) \rightarrow (S' \rightarrow T')$$

*satisfying:*

$$a \leq b \Leftrightarrow (in\, a) \leq_{T'} (in\, b)$$
$$a \leq b \Rightarrow (in\, a) \leq_{T'} (in\, b) \diamond_{T'} (in \bot)$$
$$out_{T'} \circ (\theta\, f) \circ in_{S'} = f \qquad (\circ \text{ denotes composition})$$

$\leq_{T'}$ is thus defined to be a (deterministic) $\bot$-avoiding version of $\leq$. The full semantics of ERAs will be studied in the next section where a complete set of axioms for an ERA will be presented.

ERAs are not restricted to total orders because many useful instances of incremental construction can only be represented by a partial order. One example of an ERA partial order is the *location order*—which represents a write-once location. This contains elements

$\perp_L$), for an empty location, and $\top_L$, for an error. All other elements, representing different values contained in the location, are greater than $\perp_L$ and less than $\top_L$. The Hasse diagram for this is shown in Figure 3. The simplest ERA order is the two element *boolean order*, {**true**, **false**}, in which **false** < **true**. Although this is total sets can be represented by an ERA over tuples of booleans with the product order, which is a partial order equivalent to set inclusion.

### Po-ERAs

Suppose $\leq$ is a partial order with a binary least upper bound operation:

$$lub(a,b) = x \Rightarrow (a \leq x) \wedge (b \leq x) \wedge (a \leq y \wedge b \leq y \Rightarrow x \leq y)$$

Suppose also that $\leq$ has a minimum element $\iota$. Then *lub* is associative, commutative and respects $\leq$ and we can define an ERA from $\langle T, lub, \iota, \leq \rangle$. These two conditions on the order are together equivalent to the existence of a least upper bound to all finite sets.

**Defn. 3** *Let $\leq$ be a partial order on type $T$ with least upper bound, lub, on finite sets. The ERA defined by $\langle T, \lambda(a,b).lub(\{a,b\}), lub(\emptyset), \leq \rangle$ is called a po-ERA.*

Po-ERAs have a mathematical structure similar to but not identical with semantic domains. Recursion will be defined (for all ERAs) by using limits of ascending sequences rather than least upper bounds. Even in po-ERAs the existence of a finite least upper bound does not imply the existence of arbitrary least upper bounds.

## 4 Semantics

We have looked at an algorithm—graph marking—for which the solution is the least fixed point of a recursive equation. Can we express such recursions uniformly using abstract datatypes in such a way that the meaning of programs is well defined? We hope to establish a framework within which the exact meaning of the graph marking algorithm, as the solution to an equation, is clear. To do this we need to work out a semantics for early readable aggregates. The unusual semantics of ERAs stems from the partial order, $\leq$, which makes precise the way in which construction of a multiset of elements is incremental. This enables two new programming techniques.

Firstly a lower bound to an ERA's value can be found before the value has been constructed. The function which does this, $\leq_{T'}$, is deterministic because it will only terminate when it is sure of the answer: that is to say, as soon as the aggregate is greater than the lower bound, or when the aggregate has finished evaluation. Secondly recursive aggregate equations become meaningful. We will be able to define a new operator which finds the least fixed point of a monotonic function $f : T' \to T'$ with respect to the order $\leq$.

Let us now define the semantics of ERA functions. For the remainder of this section we will be working with a typed functional language $\mathcal{L}$. Within $\mathcal{L}$ we will consider an ERA $T'$ defined by the tuple $\langle T, \diamond, \iota, \leq \rangle$. Remember that $T$ is a type in $\mathcal{L}$, and:

$$\diamond : T \times T \to T$$
$$\iota : T$$
$$\leq : T \times T \to Bool$$

are functions within $\mathcal{L}$ which we are given. The ERA $T'$ is an encapsulated datatype in $\mathcal{L}$ equipped with corresponding functions $\{\diamond_{T'}, \iota_{T'}, \leq_{T'}, \theta, in_{T'}, out_{T'}\} \in \mathcal{L}$. We are going to define the set of values in $T'$, and then give definitions of the ERA functions in terms of these values. This is more convenient than a list of axioms defining the external behaviour of the ERA functions. The world we are studying is illustrated in the diagram:

$$\begin{array}{rcc} & \diamond & \\ T: & \iota & \\ & \leq & \end{array} \quad \begin{array}{c} in :\rightarrow \\ out :\leftarrow \end{array} \quad \begin{array}{rcl} & \diamond_{T'} & \\ T': & \iota_{T'} & \theta:(S\rightarrow_M T)\rightarrow(S'\rightarrow T') \\ & \leq_{T'} & \end{array}$$

In this diagram $\theta$ converts $\leq$-monotonic (terminating) functions of type $S \rightarrow T$ to equivalent functions $S' \rightarrow T'$ between any ERA type $S'$ and $T'$, where $S'$ is an ERA over $S$. The informal semantics given in the last section means that a value in $T'$ may be defined only by a lower bound. For example $x = a \diamond \bot$ defines a value $x \in T'$ about which we only know that $a \leq x$. The values of the ERA datatype will reflect this by being either *approximate* or *precise*. Approximate values are known to within a lower bound and precise values are defined exactly. Intuitively, approximate values correspond to non-terminating computations, values with an infinite number of elements, or with one or more elements of the form $in\ \bot$. In contrast to non-terminating functions, ERA values which do not terminate are still useful because they can be tested against lower bounds. The adjective *non-terminating* will be used as a synonym for approximate and *terminating* for precise, when describing an ERA value.

When speaking of ERA values we will use the adjectives approximate, non-terminating and precise, terminating as synonyms.

The set of precise ERA values is isomorphic to $T$, and if $a \in T$ we write the corresponding precise value as $a_.$. The approximate ERA values also include a set isomorphic to $T$, but may also include extra limit points. We write $x_\uparrow$ for an approximate ERA value where either $x \in T$ or $x \in T_{lim}$ is a *limit point*. Limit points will be defined below from the order $\leq$ in such a way that $\leq$ extends consistently to the set $T \cup T_{lim}$, which can be thought of as the closure of $T$ under limits of ascending sequences.

Thus we have defined $T' = \langle T, \bullet \rangle \cup \langle T \cup T_{lim}, \uparrow \rangle$. The notations $x_\uparrow$ and $x_.$ can be read as 'at least $a$' and 'precisely $a$'.

The set of axioms which define ERA functions are listed in Figure 4. The top group of axioms restate the conditions on $\diamond$, $\iota$, $\leq$ comprising the definition of an ERA. Then $\diamond_{T'}$ is defined to give an approximate value if either of its arguments is approximate. The next two groups define the interface between ERA values and the rest of $\mathcal{L}$. An important property of approximate values is that they can't be read using *out*, and the definition of $\leq_{T'}$ says that if two ERA values are compared the comparison will terminate if one of the arguments is precise and the other is either an incompatible lower bound or precise.

Note that the mapping of limit points under these axioms depends only on their place in the order $\leq$ since they cannot be read with *out*.

## Soundness

These axioms define the behaviour of the ERA functions. However we do not yet know whether we have introduced extra complexity into the semantics of $\mathcal{L}$ by allowing the construction of functions which are either not monotonic or not continuous.

$\diamond_{T'}$ is associative, commutative.
$\iota_\bullet$ is an identity for $\diamond$.
$a_\bullet \leq_{T'} a_\bullet \diamond x_\bullet$.
} From ERA definition.

$$
\begin{array}{lcll}
a_\bullet \diamond_{T'} b_\bullet & = & (a \diamond b)_\bullet & \\
a_\bullet \diamond_{T'} b_\uparrow & = & (a \diamond b)_\uparrow & \\
a_\uparrow \diamond_{T'} b_\uparrow & = & (a \diamond b)_\uparrow & \\
 & & & \\
in\ a & = & a_\bullet & (a \neq \bot) \\
in\ \bot & = & \iota_\uparrow & \\
out\ a_\bullet & = & a & \\
out\ a_\uparrow & = & \bot & \\
 & & & \\
a_\bullet \leq_{T'} b_\bullet & = & a \leq b & \\
a_\bullet \leq_{T'} b_\uparrow & = & \textbf{true} & (a \leq b) \\
a_\uparrow \leq_{T'} b_\uparrow & = & \textbf{false} & (b < a) \\
x \leq_{T'} y & = & \bot & (x, y \text{ not as above}) \\
 & & & \\
f'\ a_\uparrow & = & (f\ a)_\uparrow & (f' = \theta(f)) \\
f'\ a_\bullet & = & (f\ a)_\bullet &
\end{array}
$$

Figure 4: Equations for ERAs

We can prove that this is not possible by defining a partial order $\sqsubseteq$ on $T'$ with respect to which the functions $in, out, \diamond_{T'}, \leq_{T'}, (\theta f)$ are monotonic. The order will correspond to information about values in $T'$; $a \sqsubseteq b$ if and only if $b$ is consistent with and at least as informative as $a$:

$$
\begin{array}{l}
a \leq b \Leftrightarrow a_\uparrow \sqsubseteq b_\uparrow \\
a \leq b \Leftrightarrow a_\uparrow \sqsubseteq b_\bullet \\
a = b \Leftrightarrow a_\bullet \sqsubseteq b_\bullet
\end{array}
$$

It is now straightforward to verify that each of the ERA functions is monotonic with respect to this order on $T'$ and the usual (flat) denotational order on $T$. The functions $f : S' \to T'$ admitted by $\theta$ are also monotonic with respect to the orders so defined on $S'$ and $T'$, because of the condition that the domain of $\theta$ contains only $\leq$-monotonic total functions.

We will not specify a complete denotational semantics for $\mathcal{L}$, although it can be seen from this definition that $\sqsubseteq$ is similar to a semantic order on $T'$. However the solution to ERA recursive equations will be defined below using the limit of an $\leq$-increasing sequence, without the need for a least upper bound on $T'$, which need not exist. With this proviso it is helpful to note that the element $\iota_\uparrow \in T'$ is a completely uninformative lower bound. It is the minimum element in $\sqsubseteq$ and so might correspond to a $\bot$ element of $T'$ in some denotational semantics.

### Limit Points

Approximate ERA values may be defined by an infinite increasing sequence of values in $T$. Operationally this represents a value $x_\uparrow$ which satisfies a set of bounds in the order $\leq$.

We define the set of all infinite ascending sequences in $T$:

$$T^\infty = \{(a_i); \forall i.\, a_i \in T,\, a_i < a_{i+1}\}$$

Now for any $(x_i) \in T^\infty, a \in T$ we use the natural definition of upper bounds of $(x_i)$:

$$a \geq (x_i) \Leftrightarrow \forall j.\, a \geq x_j$$

Define two sequences in $T^\infty$ to be equivalent if they have the same set of upper bounds in $T$. If $(x_i) \in T^\infty$ we denote its equivalence class by $[x_i]$ and call the quotient set $T_{lim}$. This quotient is the set of limit points constructed from $\leq$. For example the limit points of the rationals are the irrational real numbers, and the integers have just one limit point, $[i]$, which represents unbounded sequences.

The order $\leq$ on $T$ extends naturally to $T_{lim}$. Let $U(x)$ be the set of upper bounds in $T$ of $x \in T_{lim}$. $\forall a \in T, x, y \in T_{lim}$:

$$a \leq x \Leftrightarrow \forall u \in U(x).\, a \leq u$$
$$x \leq y \Leftrightarrow U(x) \supseteq U(y)$$

## Aggregates with an infinite number of elements

Limit points in $T'$ are necessary to represent aggregates with an infinite number of elements. Let

$$a = a_0 :: a_1 :: \ldots$$

be an infinite lazy list of type $T'$. The values $(c_i)$:

$$c_0 = \bot$$
$$c_1 = a_0 :: \bot$$
$$c_2 = a_0 :: a_1 :: \bot$$
$$\ldots$$

are an ascending sequence in the semantic domain of $\mathcal{L}$, so their images under the function `collect`, see below, which aggregates a list of type $T$, must converge to the image of their limit: `collect`$(a)$.

```
fun collect   []  =ι.
  | collect (h::t)= (in h) ◇T' collect t;
```

Therefore this is always an approximate value, even if $a$ is bounded by one of its elements. If $a_i = in(i)$, in the aggregate defined by $\leq$ on the integers, `collect`$(a) = \omega_\uparrow$, where $\omega = [i]$ is the infinite (and only) limit point in this aggregate. However if $a_i = in\ 0$, `collect`$(a) = 0_\uparrow$, not $0_.$. This non-terminating value reflects the fact that evaluation of an infinite lazy list will never be complete.

## Modelling ERAs with lazy lists

Values in $T'$ may be be identified with equivalence classes of $\leq$-increasing lazy lists of values in $T$. ERA functions can then be defined with the help of a non-deterministic choice operator as functions on these lazy lists.

This is an operational definition of ERAs which is similar to the way in which Burton defines his *improving values* in [Bur89]. Burton's improving values are defined by an abstract datatype with two associative, commutative operations, which he calls *spec_max* and

*minimum*. These have a semantics which allows speculative minimax searching. Burton's *improving value* semantics is similar to ERA semantics on po-ERAs with total orders. Then Burton's *spec_max* operation is identical to $\diamond$, and *minimum* is a $\perp$-avoiding minimum operation which terminates more often than the equivalent function using $\leq$:

$$Min\ x\ y = \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y$$

Burton represents values in $T'$ by equivalence classes of possibly infinite, strictly increasing, lists of values in $T$, with the interpretation that each element represents a lower bound of the $T'$ value, and the proviso that no element of a list may be $\perp$, although the tail of the list may be $\perp$. There are a number of different cases to consider:

- A finite list:

  $$a_0 :: a_1 :: \ldots :: a_n \quad (a_i < a_{i+1})$$

  This is equivalent to the precise value $a_{n\bullet}$.

- A list which does not terminate:

  $$a_0 :: a_1 :: \ldots :: a_n :: \perp \quad (a_i < a_{i+1})$$

  This is equivalent to the approximate value $a_{n\uparrow}$.

- An infinite list:

  $$l = a_0 :: a_1 :: a_2 :: \ldots \quad (a_i < a_{i+1})$$

  This is equivalent to an approximate limit value $[a_i]_\bullet$.

In [Bur89] Burton characterises limit values according to least upper bounds. The (unique in a total order) case in which no upper bound exists he calls $\infty$, otherwise he assumes that there is a least upper bound $a$ and calls the resulting value $a - \epsilon$.

This simple semantics is not used here, because it does not represent the general case in which least upper bounds may not exist in $T$. Even with a restriction to total orders for $\leq$ least upper bounds may not exist. A counterexample can be found in the rational numbers with the usual order. Then the increasing sequence $(x_i)$ where:

$$x_i = \sum_{j=0}^{i} \frac{1}{j!}$$

has an irrational limit, which in the real number system is $e$. It has no least upper bound in the rationals.

It is more satisfactory to leave limit values defined by equivalence classes of lists which have the same set of upper bounds, as we have done above in the definition of $T_{lim}$. A limit value whose set of upper bounds has a least element $a$ is then equivalent to Burton's $a - \epsilon$.

In our semantics the natural extension of $\leq$ from $T$ to $T_{lim}$, together with the equations in Figure 4, defines the semantics of limit elements $x_\uparrow$. We can see from this that the problem of limit ERA values is only a technical one, because the finite semantics extends uniquely to limits.

The difference between ERAs and *improving values* reflects a difference in application. Burton is concerned with control of speculative evaluation, and we are concerned with specification of concurrent evaluation. There is considerable overlap, and it is possible to

add an operator like *minimum* to ERAs, in order to control nested speculative evaluation. Equally it would be possible to extend Burton's datatype to partial orders, replacing maximum and minimum by least upper bound and greatest lower bound. Then it is possible that concurrent ERA recursion, described in the next section, could be used with *improving values.*

## ERA Recursion

The graph marking problem requires the use of an ERA value which is the least fixed point of an equation:

$$a : T' = f'\, a$$

where $f' : T' \to T'$.

We can now give this equation a precise meaning by using the representation of $T'$ values as increasing lists of $T$ values. A function $f' : T' \to T'$ is of the form:

$$f' = \lambda x.\, c \diamond_{T'} (\theta f x)$$

and may be represented by an equivalent function of type $List[T] \to List[T]$, which merges together, using an elementwise $\diamond$ operation, two lazy lists, one representing $c$ and one representing $f$ mapped over the argument list. The solution to the recursive equation can be represented by the recursively defined list:

$$l = truncate\ ((\iota :: f\ l) \diamond c)$$

where *truncate* ends the list when a repeated element indicates that the fixed point has been found:

$$truncate\ (a :: b :: x) = \textbf{if } a = b \textbf{ then } a :: [] \textbf{ else } a :: truncate\ (b :: x)$$

For example if $c = \iota$ the elements of the list are the $\leq$-ascending sequence:

$$\iota,\ f\iota,\ f^2\iota,\ \ldots$$

If $f^n\iota = f^{n+1}\iota$, the list is finite, and $in\ f^n\iota$ is the least solution to the equation $a = f'\, a$. Otherwise the list represents a limit point $(in\ f^i\iota)$.

This solution to an ERA value recursion is thus a value $v_x$ where $v$ is the least (with respect to $\leq$) fixed point of $f'$ over the set $T \cup T_{lim}$. The solution will be an approximate value whenever the equivalent lazy list does not terminate. This may be because $c$ is approximate, either through being an infinite or a non-terminating list, or because the sequence of $T$ values found by the iteration is strictly increasing.

We thus have three distinct ways of constructing a non-terminating ERA value:

- One of its elements is $(in\ \bot)$.
- It has an infinite number of elements.
- It is a non-terminating ERA recursion.

These are all accommodated within a semantic model which has only two types of values: approximate and precise. These values have meanings which are transparent and easy to reason about when writing programs, and limit values do not present extra complication because they are defined by the extension of $\leq$ to limits and the equations in Figure 4.

## ERAs on non-flat domains

When an ERA is over a type $T$ which is not flat the definition of semantics is more complicated. In a po-ERA it is no longer certain that the usual $\diamond$ operation:

$$\diamond = \lambda(a, b).\ \textbf{if}\ a \leq b\ \textbf{then}\ b\ \textbf{else}\ a$$

will form an operator which is associative. Consider a set of lazy lists of integers, with the lexicographic ordering $\leq_l$.

$$((2 :: 0) \diamond (1 :: \bot)) \diamond (1, \bot) = (2 :: 0)$$

whereas

$$(2 :: 0) \diamond ((1 :: \bot) \diamond (1 :: \bot)) = \bot$$

because one of the $\leq_l$ comparisons will not terminate. A lazier implementation of $\diamond$ would restore associativity:

```
fun (h::t)◇(h',::t') =
       if h = h' then (h::(t ◇ t')
  else if h > h' then (h::t)
  else (h'::t');
```

A more serious problem is that in ERAs over non-flat type the ERA functions $\leq_{T'}$, *out* may not be monotonic. We will therefore not define ERA operations over non-flat types except for the particularly simple case of *product* types. ERAs can be constructed over product types, each component of which defines an ERA, in two different ways. Consider ERAs $S', T'$ over types $S, T$. If the product is coalesced, so that its components cannot independently be non-terminating, the product (partial) order on $S \times T$ will define an ERA $(S \times T)'$ in which non-termination is a global property. Alternatively the tuple of ERA types, $S' \times T'$, is a product in which each component separately is either precise or approximate. In both cases the algorithm of the next section will enable highly concurrent ERA operations, and in particular recursion, over products.

## Conclusions

Recursion on ERAs could be implemented functionally by constructing a truncated, recursively defined, lazy list, but this does not represent the concurrency which derives from the commutativity and associativity of the ERA $\diamond$ operator. We can now reexamine the naive graph reachability defining equation:

$$M = R \cup close(M)$$

The set of graph nodes $V$ can be made into a po-ERA by set inclusion, with $M \diamond M' = M \cup M'$. The recursion can then be expressed as a po-ERA recursive equation. In order to get the optimal semantics for this recursion it is necessary to treat each element of the set independently. We can now understand this more clearly—it corresponds to a mutual recursion over a product of boolean po-ERAs, each of which can be updated separately. In Section 6 below a functional program using arrays of boolean aggregates will be given for this equation which, when implemented with the algorithm for ERA recursion described in the next section, provides an optimal solution to the graph reachability problem.

# 5 Implementation

An Aggregate $T'$ over type $T$ may be implemented by storing an intermediate result, initialised to $\iota$, of type $T$, and performing an in-place update to add each element as it becomes known. In a multiprocessor a separate intermediate result can be kept for each processor with the final result constructed by aggregating intermediate results across processors: this ensures that sequential updates do not limit global concurrency.

This implementation requires that all intermediate $\diamond$ operations in a computation be replaced by an equivalent set of in-place updates. We will assume that a compiler can make this transformation, and also implement 'update permit' reference counts (one for each separately updatable intermediate result) so that the aggregate can be read as soon as all threads of computation which combine to form the aggregate value have terminated.

The compiler is free to define intermediate result locations at compile time or dynamically (e.g. one per processor involved in the aggregate computation), and must ensure that appropriate counts are kept. Typically each processor will have its own private result which is locally updated. The global result is got by combining each local result when the computation has finished. Intermediate results can be combined in an arbitrary tree of $\diamond$ operations, so in a mesh connected multiprocessor this may be arranged to make best use of communication topology.

The complexity of simple aggregate implementation depends on the degree of dynamic process scheduling in the computation, and the extent to which optimally concurrent evaluation is required. Identifying aggregates in a program puts the management of this where it can best be decided—with the compiler and run-time system.

### ERAs

The implementation of a simple aggregate does not raise issues of lazy evaluation since the result is always strict on (all) aggregate elements. The implementation of ERAs is more complicated. Two feasible strategies for ERA value computation are *parallel order* and *suspensive*. The two strategies have widely differing consequences for implementation.

The semantics we have given for ERAs is $\perp$-avoiding. This requires all elements of the ERA be evaluated fairly in parallel, otherwise evaluation of an element which turned out to be $\perp$ could block discovery of the ERA's lower bound.

In parallel order evaluation these concurrent evaluations are started when the ERA is specified and never stopped. Parallel order is the parallel equivalent of applicative order and specifies that functions and arguments are evaluated in parallel. It is the natural evaluation strategy for highly concurrent systems since it maximises potential algorithmic concurrency. Unfortunately it means that an ERA element of *in* $\perp$, while not necessarily stopping a result from emerging, will have a permanent impact on future CPU resources.

In suspensive evaluation the fair parallel computations which have been started may be suspended as soon as they are known not to be necessary. This is 'lazy' in the sense that some unnecessary work is not done. However in general the parallel evaluation of ERA elements (even with suspensive evaluation) means that work is done speculatively which may turn out not to have been necessary. Suspensive strategy means that the minimum speculative evaluation necessary to preserve $\perp$-avoiding semantics is done.

We will first define suspensive evaluation of single ERAs. At any time the evaluation of an ERA may be active (with each unfinished element scheduled fairly in parallel) or suspended, with each unfinished element suspended. During evaluation the value $v$ of an

ERA $T'$ is represented by a *current lower bound* (*clb*) of type $T$. As elements are evaluated the *clb* is updated appropriately. For each element $a$:

$$clb := clb \diamond a.$$

If $v$ is an argument of *out* evaluation will continue until $v$ is known precisely. This will require all elements of the ERA to be fully evaluated. If the ERA order has a maximum element $m$, and $clb = m$, then $v \in \{m_\uparrow, m_\bullet\}$. A different *out* function can be defined using $\leq_{T'}$:

$$newout\ x = \textbf{if}\ (in\ m) \leq_{T'} x\ \textbf{then}\ m\ \textbf{else}\ out\ x$$

If this function is used suspensive evaluation of $\leq_{T'}$, defined below, will result in evaluation similar to a parallel **or**. As soon as one ERA element equal to $m$ is found all other evaluation is suspended.

In general when the ERA is the argument of $\leq_{T'}$ evaluation may be suspended before a maximum *clb* is reached. Suppose $x, y$ are values in $T'$. From the ERA defining equations in Figure 4 the necessary and sufficient conditions for $y \leq_{T'} y$ to terminate are:

| Condition | $a \leq_{T'} b$ |
|---|---|
| $x = a_\bullet$ and $a \leq clb(y)$ | **true** |
| $y = b_\bullet$ and $b < clb(x)$ | **false** |
| $x = a_\bullet$ and $y = b_\bullet$ and $a \not\leq b$ | **false** |

These cases follow from the $\perp$ avoiding semantics of $\leq$. Thus if $clb(x) \leq clb(y)$ and $x$ finishes a result will be returned, if $clb(y) < clb(x)$ the termination of $y$ will return a result, and if $x$ and $y$ are incomparable both must finish before a result can be determined. Therefore in suspensive evaluation of $y \leq_{T'} y$ the currently greater *clb* (if there is one) may be suspended.

Suspensive evaluation for ERAs over non-flat domains is similar, only now the update of a *clb* may not terminate immediately. The associativity of $\diamond$ ensures that the *clb* is independent of the order in which elements are used to update the *clb*. Product ERAs have a particularly simple implementation because each component of the *clb* can be updated separately as the corresponding elements finish.

Monotonic functions between ERAs lead to new conditions for ERA element evaluation. If any element of an ERA value $a$ is a monotonic function of another ERA value $b$ we say that $a$ depends on $b$, and that $b$ is an ERA-element of $a$. Active evaluation of any ERA value must entail active evaluation of all its ERA-elements. Whenever *clbs* of these ERA-elements are updated the *clb* of the dependent ERA is also updated.

The algorithm will be given here in a form useful only for po-ERAs, since the algorithm for general ERAs is more complicated. In an ERA which is not a po-ERA, whenever a *clb* is updated, elements derived from the old *clb* value must be removed from *clbs* as elements derived from the updated clb are added.

**Algorithm 1** *Let*

$$\boldsymbol{a}_j = (in\ k_{j1}) \diamond \ldots \diamond (in\ k_{jn_j}) \diamond (f'_{j(n_j+1)}\boldsymbol{a}_{q(j,n_j+1)}) \diamond \ldots \diamond (f'_{jm_j}\boldsymbol{a}_{q(j,m_j)})$$

*be a possibly recursive set of equations defining ERA values* $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_J$, *with* $\boldsymbol{a}_j \in T'_j$. *The numbers* $n_j, m_j$ *define the number of elements of form* $k_{ji}$ *and ERA-elements in the equation for* $\boldsymbol{a}_j$. *Each* $q(j,i)$ *indicates which ERA value is the argument of the function* $f'_{ji} : T'_{q(j,i)} \to T'_j = \theta_j f_{ji}$. *for some* $f_i : T \to_M T$.

```
∀ j. (c_j := ι_{T_j}; A_j := [1, m_j])
while ∃ j. A_j ≠ ∅ do
begin
  choose j, i ∈ A_j such that if i ≤ n_j, k_ji ≠ ⊥
  A_j := A_j − {i}
  if (i > n_j) then
    x := c_j ◇ (f_ji c_j)
  else
   x := c_j ◇ k_ji
  endif
  if x ≠ c_j then
    c_j := x; ∀ k,l. if q(k,l) = j then A_k := A_k ∪ {l}
  endif
  return c_j
end
```

*The choice of $j,i$ in the algorithm must be fair, so no possible choice is unchosen for more than a finite time. Choices may be made in parallel, as long as the clb updates are atomic.*

This algorithm defines the implementation of recursive systems of ERA values in which the iterative update of *clbs* implements the recursion. In a recursive system we would like to know that this iteration is well behaved, so that if a least fixed point for a recursion exists the iteration will find it whatever order is chosen for *clb* updates.

**Theorem 1** *Let $a_j$ be as defined above for $j \in [1, J]$. Then a solution for $a_j$ exists and the least such solution will be found by Algorithm 1.*

The result covers non-terminating ERA recursions whose value is a limit element $a_\uparrow$ ($a \in T_{lim}$). In this case the algorithm will not terminate, resulting in $\perp$ as required for *out* $a_\uparrow$. However the algorithm will find within finite time any lower bound for $a_\uparrow$, so $\leq_{T'}$ operations will terminate correctly.

The proof of this will be sketched here. It rests on the inductive hypothesis that after $n$ updates the set of clbs is less than or equal to the desired least fixed point, $P$. After any one more update it is then easy to prove that this condition still holds. Certainly it holds at the beginning so the *clbs* are bounded from above by $P$. Equally if the algorithm terminates the termination condition means that the found *clbs* satisfy the ERA defining equations. Finally increasing *clb* values form an ascending sequence which cannot be stationary for more than a finite number of consecutive elements before the algorithm terminates. Therefore either the fixed point is a limit value, in which case the non-termination of the algorithm is expected, or the algorithm must terminate.

The comparison of *clb* values before and after every update requires that these be of a type that admits equality, however this is not strictly necessary. All that is required is to be able to tell, given $x, y \in T$, whether $x \diamond y = x$. For the write-once location ERA this depends only on whether or not $y = \iota$, so write-once locations of functions can be used in an ERA recursion even though the functions themselves cannot be compared.

## Threaded Aggregates

One class of simple aggregates exists for which the most efficient implementation is particularly interesting. In procedural languages global resource allocation is a frequently

encountered concurrent operation. A typical example of this is in the distribution of unique and contiguous tags to each distinct node of a data structure. In declarative languages this might be modelled by a `cons_with_merged_tags` function which takes two tagged structures and returns a copy of each with tags relabelled.

If we now define an abstract datatype *Tag* with a set of operations which hide the indeterminacy introduced by different tag labellings the function `cons_with_merged_tags` becomes both commutative and associative: it therefore makes an aggregate. If this function were defined as an aggregate in the first place it might be possible to use an implementation as efficient as the global procedure `tag_allocate` which has a persistent local variable and returns the next tag every time it is called:

```
current_tag_value = ref 0;

fun tag_allocate()=
let val x = ! current_tag_value
in
   ( current_tag_value := x+1; x)
end;
```

You may now be wondering what might be the purpose of this, since we have converted an inefficient but manifestly concurrent declarative algorithm into an explicitly sequenced (and therefore less concurrent) implementation. For evaluation on one processor this algorithm is much more efficient than the corresponding merge algorithm. It is not difficult to modify the algorithm so that no sequential bottlenecks exist by a process called *combination.* The idea is that if two simultaneous tag requests have to be processed they are combined into a single request and forwarded for further processing. This allows a tree of processing agents to distribute tags uniquely without any sequential bottlenecks. This operation has been found by designers of concurrent hardware to be so important that it is implemented directly in special *combining networks*, for example in IBM's RP3 [AH88].

The principle which this illustrates is that there are determinate operations whose essentially associative character allows highly concurrent execution even while the most efficient implementation of the operation is both sequential and apparently indeterminate.

We will call this type of computation a *threaded aggregate.* Threaded aggregates are a way of implementing operations such as `cons_with_merged_tags`. These operations are associative and commutative outside an encapsulated datatype, and therefore define aggregates. Further investigation of threaded aggregates is beyond the scope of this paper.

# 6 Examples

## Array aggregates

An array aggregate could in principle be implemented as a pure functional array of aggregates, with the usual $\log n$ overhead whenever an element is added. An implementation which uses in-place update of each aggregate does not have this overhead, and may be specified in a program by using an abstract datatype $array[T']$ with functions:

$$\begin{aligned}
&\textit{inject: } Int \rightarrow T' \rightarrow array[T'] \\
&\textit{project: } Int \rightarrow array[T'] \rightarrow T' \\
&\diamond_a : array[T'] \times array[T'] \rightarrow array[T'] \\
&\textit{collect_indices} : array[T'] \rightarrow List[Int]
\end{aligned}$$

The operation $\diamond_a$ is the array aggregate $\diamond$ operation, which combines elements with the same index using $\diamond$ from $T'$. *Collect_indices* returns a list sorted by index, of all indices for which the value of the component is not $\iota_{T'}$.

*Inject* may be implemented by adding its second argument to the aggregate indexed by its first argument, using in-place update as described above. All other aggregates in the array are unaffected by operation. Arbitrary-sized sparse arrays are used in this definition although in a practical implementation fixed size arrays might be used. If a fixed size array aggregate has an out-of-bounds index any resulting error indication must be well behaved: a set of *all* offending indices must be returned rather than the first bad index to be written.

The type of an array of aggregates has a natural interpretation as the product of its base types. It is sufficient for the arrays presented here not to distinguish between an array all of whose elements are $\perp$, and $\perp$. Note however that an empty array, all of whose elements are $\iota_{T'}$, is distinct from this.

Because of this structure the evaluation of an array aggregate can treat each component separately and lazily not evaluate components which are not demanded. This leads to a better implementation (remember all ERA elements are evaluated fairly in parallel so 'lazy' evaluation does not alter semantics) in which the function *project n a* demands evaluation first of all indices $i$ of elements *inject i b* and only then for $i = n$ demands evaluation of $b$. However this is only optimal if indices $n$ are guaranteed to terminate. Otherwise the *inject* $\perp$ $\iota$ would cause non-termination of the aggregate when it need not. The cost of deciding which elements to evaluate in an array is avoided if parallel order evaluation is used.

## Graph Reachability problem

Using an array of boolean aggregates the graph reachability algorithm can be defined as a function `reachable_from`:

```
fun mapa f [] = inject 0 ι
|       f (h :: t)= (f h) ◇a (mapa f t);

fun list_to_array = mapa (fn n=> inject n true);

(* next is a list of (vertex, vertex list) pairs which represents the
   arcs of a graph.
   r is a list of nodes.
   function returns a list representing all nodes
   reachable from r *)
fun reachable_from next r=
let
   fun add_vertices m (v,l) = mapa l (fn n => inject n (project v m));
   val m = (list_to_array r) ◇a (mapa (add_vertices m) next)
in collect_indices m
end
```

## The histogram problem

Using an array of $\langle Int, +, 0\rangle$ aggregates it is possible to write an efficient solution to the histogram problem mentioned in Section 2.

```
datatype tree = node of Tree * Tree
              | leaf of Int;

fun histogram (leaf n) = inject n (in_Int' 1)
    histogram (node t1 t2)= t1 ◇_a t2;

fun column n hist = out_Int' (project n hist);
```

The difference between this and a solution of the problem without aggregates is only one of implementation—the aggregate acts as an annotation which the compiler can use to optimise implementation. In this case all *inject* operations within recursive calls of *histogram* can be turned into in-place updates of a result value (or values on a multiprocessor as we saw on page 15). The $\diamond_a$ operations within recursive calls of `histogram` have no associated run-time cost.

The use of aggregates as implementation hints contrasts with the previous example, where the aggregate expresses different semantics.

## Expressing Side Effects

Exceptions can be added to a pure language by extending all functions to operate on a direct sum of their normal type and an exception type. Exception values can then be propagated unchanged through outer functions until intercepted by a special exception handler. In impure functional languages, like ML [Mil84], exceptions can have an operational semantics, in which a defined evaluation order is used to ensure that only one exception (the first to be evaluated), is returned as the result of an exceptional computation.

Williams and Wimmers [WW88] have proposed a language FL in which arbitrary side-effects are incorporated in a functional program by pairing every language value with a history which is updated by functions which have side effects. The consequent plethora of histories is resolved by defining a particular but fixed order in which history updates are threaded together, so that the effect is the same as an operational semantics with a depth-first call tree evaluation order. This behaves like an operational semantics, with side effects turned into explicit data value updates. The title of [WW88] is "Sacrificing simplicity for convenience: where do you draw the line?", a question which we will now try to answer. If an applicative evaluation order is used to define a depth-first call-tree traversal by which side effects are sequenced the order in which side-effects happen is at best unrelated to the declarative program semantics, and at worst obscure. In a lazy language any feasible order in which side effects could be sequenced would be more obscure, since evaluation order is less easy to predict from the static structure of the program. On the other hand the inspection of 'side-effects' which are produced during evaluation of an expression but which have no dependency on a sequential order is much safer. Exceptions, as they are commonly used, are an example of this because only one exception is likely to occur in a calculation. The condition that side-effects be order independent is exactly that they form an aggregate, and this is where we propose the line should be drawn.

Aggregates can be propagated through a functional language, carrying the most general sort of order-independent side-effect, by pairing all language values with an aggregate value and redefining (overloading) all functions to propagate the aggregate component whilst acting normally on the value component. Propagation is exactly by strict dependence, so the aggregate part of an evaluated expression is the aggregate of all side effects raised during its evaluation.

Overloading can be implemented as syntactic device, however it allows 'constructor-functions' to be be written which have the effect of adding an element to the aggregate component of an expression. The aggregate can be picked up by an outer function which acts as an aggregate handler, like an exception handler. One difference from exceptions is that the normal value of a program is not automatically thrown away when an aggregate element is constructed, so side-effects can be handled together with normal values.

Suppose that the domain of a lazy functional language is of the form:

$$D = B \oplus (D \to D)_\perp$$

where $B$ is a set of flat constants. In a lazy language tuples, conditionals, data constructors can all be implemented using functions. For simplicity suppose this to be the case. The overloaded domain, $D'$ can be written:

$$D' = (A \otimes B) \oplus (D' \stackrel{new}{\to} D')_\perp)$$

where A is an aggregate datatype. Functions are redefined so as to propagate aggregate components from an applied function to its result, so that the aggregate component of an evaluated expression is the aggregate of the aggregate components of all functions applied in its evaluation. This can be represented by a new operator *apply'* which handles the aggregate components of an expression and replaces normal application in all expresions.

Let $\pi_a$ and $\pi_d$ be projections of $D'$ onto aggregate and domain components, so:

$$D' = \pi_a D' \otimes \pi_d D'.$$

The definition of the new *apply'* is given below:

$$apply'(\langle f_a, f_d\rangle, g) \;=\; \langle f_a \diamond \pi_a(f_d\, g), f_d\, g)\rangle$$

As a result of this overloading the aggregate component of an expression is the aggregate of the aggregate components of all subexpressions on which the expression is strictly dependent:

$$\pi_a x = \diamond\{\pi_a y;\ \pi_d y = \perp \Rightarrow \pi_d x = \perp\}$$

Overloading a language with an ERA is particularly useful, because this allows side-effects to be obtained from an evaluating expression before it terminates. The restriction to order-independent side-effects is in practice very important. In a lazy language it may not be easy to work out whether a particular subexpression will be evaluated, but this fact is of more significance to a programmer than some arbitrary side-effect combination order. Therefore aggregate side-effects are less likely to cause obscure program errors than more general side-effects. In a concurrent program aggregate side-effects are appropriate because they do not put unnecessary constraints on evaluation order.

One use of ERA overloading is motivated by the use of streams to manage I/O, and uses information about program data dependence explicitly. A number of writers have proposed using angelic merge of concurrently evaluated lazy lists to allow concurrent stream

processing functions to communicate. The unconstrained use of a non-deterministic merge for this purpose has been advocated by Henderson [DHT82]. Other writers, notably Stoye [Sto85] and Augustsson [Aug89], study non-deterministic primitives which are packaged in a form which is referentially transparent. By using ERAs it is possible to move yet further from impure semantics and construct a *pseudo-time* deterministic merge which is both referentially transparent and deterministic. This does not take into account the different evaluation times of functions, which would necessarily[2] be non-deterministic. Instead every lazy list element to be merged has a *pseudo-time*, determined explicitly by the program. By coding pseudo-times as elements of a po-ERA ordered by time, the pseudo-time of an expression can be made dependent on the pseudo-times of subexpressions it is strict on, so that pseudo-times are always causal. Then a *pseudo-time deterministic merge* may be written using $\leq$ which merges lists from interacting functions in an order which avoids blocking.

The use of pseudo-time deterministic merge separates 'real' non-determinism, resulting from unknown function evaluation time, from uses of non-deterministic merge which are implicitly determinate because the merged functions sequence each other. This does not replace real non-determinism, however it does allow a wide class of interactive algorithms to be expressed in a completely determinate language. A full discussion of pseudo-time deterministic merge can be found in [Cla90].

### Single Assignment Programs

It is possible to convert single assignment language programs, such as functional programs using *I-structures* [ANP87], into equivalent determinate aggregate programs. An I-structure is an array of locations with an operational semantics defined by operations *create_I, read_I and write_I.* I-structures are created empty and can then have values written into each of their locations only once. A multiple write of a location results in a global program error. Reads of single locations return the corresponding values as soon as the location has been written: if a location is never written the value of reading it is $\bot$.

I-structures can be represented as ERAs by using a modified product of location orders in which there is only one global error element, so that the $\top_L$ elements for each location are coalesced in the product. The ERA $\diamond$ operation reflects the semantics of multiple writes:

$$\begin{aligned} x \diamond \bot_L &= x \\ x \diamond y &= \top_L \qquad (x,\, y \neq \bot_L) \\ x \diamond \top_L &= \top_L \end{aligned}$$

Each I-structure *write_I* is represented by a single ERA element of the form *inject* $n$ $(in\, x)$. and *create_I* by the empty array. The I-structure operational semantics is then equivalent to a declarative semantics with I-structure aggregate side-effects. I-structure *read_I* operations can be represented by *project* functions from the aggregate, and a functional program using I-structure reads and writes can be represented by a recursively defined I-structure aggregate. These semantics are not identical to those of a program with I-structures, because in order to preserve determinacy in the presence of program errors it is not possible to print any value dependent on an I-structure location until all I-structure computation has finished, and it is known that there is no global error. Otherwise an indeterminate

[2] Unless some form of timed simulation were used to introduce deterministic evaluation time information, for example Friedman's engines, [HF84].

result could be printed. This does not reduce program concurrency because within an ERA recursion values defined only by lower bounds on the ERA order can be used freely.

# 7 Conclusions

We have investigated programming with aggregates, concurrently specified collections of elements which have the property that the meaning of the collection is independent of the order in which the collection is specified. The central thesis of this paper is that aggregates underlie many different types of useful concurrency. There is nothing intrinsically non-declarative about the efficient implementation of aggregate values, which can be written as trees of commutative associative operators. However pure functional languages do not adequately express aggregates and so language extensions are necessary to do this. Thus the notion of aggregates unifies a wide class of functional language implementation techniques and semantic extensions.

A characteristic of all aggregates is the hiding of evaluation-order indeterminacy: we have seen how the requirements for optimal concurrency, together with the existence of an associative commutative operation, must always give rise to this. But different types of aggregate result in different programming techniques.

Single threaded aggregates correspond to server processes in a message passing system where the meaning of the replies is independent of the order in which requests are processed. The indeterminate values are the local state of the server and the replies. All of these are encapsulated so that the indeterminacy cannot be seen by the rest of the system.

ERAs correspond to the imperative use of incrementally constructed shared data structures. Subsequent construction may be influenced by the current value of the shared structure, which during construction has an indeterminate value. This structure is encapsulated with operations which have necessarily determinate results. ERAs can also be used to do recursion on user-defined semantic domains. This may, but does not necessarily, involve speculative evaluation. The graph marking example performs a determinate set of calculations, without speculative evaluation, in an indeterminate order.

Finally aggregates may be used to handle side-effects in a clean way. Side-effects contained in an aggregate have no dependence on evaluation (or any other) sequence, and so are less obscure than other types of side-effect. One example of this is a pseudo-time deterministic merge. This can be implemented with ERAs and used to distinguish between time deterministic and evaluation time deterministic uses of angelic merge in systems which implement message passing operating systems with concurrent streams. The former is deterministic whereas the latter is non-deterministic, with correspondingly greater semantic complexity.

The different types of aggregate which we have reviewed are classified as a family tree in Figure 5. Many different functional language extensions have been proposed as single solutions of particular problems. Some of these have a place within the family of aggregates, others do not.

Recent work from Hughes [HO90] has proposed the use of a non-deterministic, encapsulated, set construction which has semantics intermediate in complexity between that of this work and that of unrestricted non-determinism. This seems interesting, however the proofs which Hughes has made of program correctness, demonstrating the tractability of his semantics, are for programs which can be rewritten without any non-determinism. For example Burton in [Bur89] has given an outline proof of the correctness of a parallel

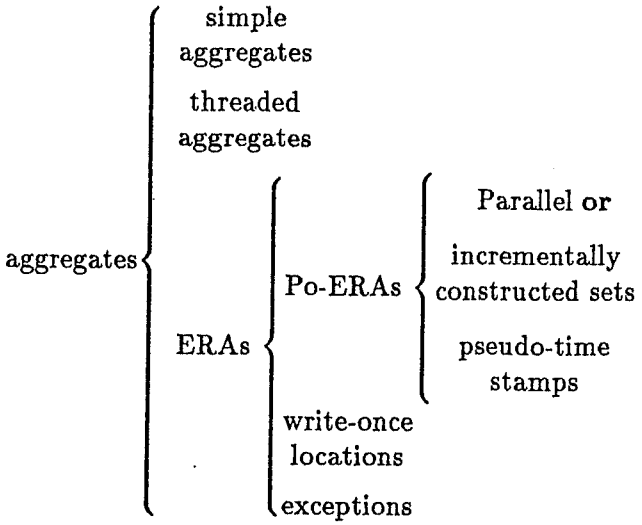


Figure 5: Aggregate family tree

least-cost search algorithm written using his *improving values*. Hughes' work has extra semantic complexity, it has yet to be shown that this cost results in greater expressiveness.

In this paper we have identified a unifying principles in the declarative treatment of the indeterminacy introduced by concurrent computation. The abstract datatypes we have defined are therefore more general than those used by other writers concerned with specific languages extensions. It is not known whether there is a small set of primitive abstract aggregate constructors sufficient to generate all useful aggregates. Without this an aggregate library must be assembled on an ad-hoc basis, with a burden of verification whenever a new aggregates is introduced. Other writers have demonstrated good ways of introducing specific aggregates as functional language extensions. The work of Burton [Bur89] is particularly relevant. His *improving values* are very similar to a po-ERA on total orders. Burton's work could be generalised to partial orders with least upper bounds without any alteration.

Further work is motivated by these results. The area between aggregates and Burton's improving values seems particularly fruitful, and more work here might lead to a clearer picture of the relationship between single assignment languages and pure functions, and a simpler way of introducing ERA recursion.

Threaded aggregates and pseudo-time merge aggregates represent some of the styles of computation used in object-oriented languages, so further study of these may lead to a better understanding of the differences between declarative and object-oriented programs. It has already been shown that object type-inheritance [CW85] can be represented within polymorphic functional type systems.

Finally it would be interesting to find a theoretical computational model which makes explicit use of aggregates, since they underlie much concurrent computation. The work of Moggi [Mog88] on a categorical semantics of computation based on monads may be relevant here.

## References


[AH88] G. S. Almasi and S. L. Harvey. RP3. In *Design and Application of Parallel Digital Processors*, April 1988.

[ANP87] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. Computation Structures Group Memo 269, M.I.T., 1987.

[Aug89] Lennart Augustsson. Functional non-deterministic programming, or, how to make your own oracle. Draft paper, Programming Methodology Group, Chalmers University of Technology, 1989.

[Bur89] F. Warren Burton. Indeterminate behaviour with determinate semantics in parallel programs. Technical Report CSS/LCCR TR 98-03, Simon Fraser University, 1989.

[Cla90] Thomas Clarke. In preparation. Technical report, University of Cambridge Computer Laboratory, 1990.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[DHT82] J. Darlington, P. Henderson, and D. A. Turner, editors. *Purely Functional Operating Systems*, pages 177–189. Cambridge University Press, 1982.

[Gor87] Mike Gordon. A proof generating system for higher-order logic. Technical Report 103, Cambridge University Computer Laboratory, 1987.

[HF84] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *LISP and Functional Programming*, pages 18–24. ACM, 1984.

[HO90] John Hughes and John O'Donnell. Expressing and reasoning about non-deterministic functional programs. Draft paper, Dept. of Computer Science, University of Glasgow, 1990.

[Mil84] R. Milner. A proposal for standard ML. In *1984 ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[Mog88] Eugenio Moggi. Computational lambda-calculus and monads. Technical report, Dept. of Comp. Sci., University of Edinburgh, October 1988.

[Sto85] William Stoye. The implementation of functional languages using custom hardware. Technical Report 81, University of Cambridge Computer Laboratory, December 1985.

[WW88] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line. In *Proc. Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 169–179. ACM, 1988.