Technical Report

UCAM-CL-TR-176
ISSN 1476-2986

Number 176

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Implementing aggregates in parallel functional languages

T.J.W. Clarke

August 1989

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

Implementing Aggregates in Parallel Functional Languages

T. J. W. Clarke *

August 13, 1989

Abstract

Keywords: non-deterministic merge, I-structures, aggregates, A-threads

Many constructions which are difficult to write efficiently in pure functional languages have as underlying semantics an aggregate. An aggregate is a collection of individual elements whose order does not matter, it can thus be constructed functionally using a commutative associative combining operator. Equivalent and more efficient implementations for aggregates exist which are operational. A new construction, the A-thread, an aggregate specified operationally which introduces provably local data indeterminacy, is defined. Operational specification of an aggregate, in which each element is specified by a separate function call, does not necessarily destroy referential transparency in a functional language. Aggregates defined using joins on partial orders allow early termination if an operational implementation is used: Arvind's 'I-structures' and Burton's 'improving values' are examples of this.

Functional languages are of interest as models of concurrent computation because they have simple denotational semantics which allows highly concurrent algorithms to be written. Nevertheless there is now much evidence [10] which indicates that pure functional semantics is too restrictive for use as a general concurrent programming language.

*The author wishes to thank The SERC, for a Research Fellowship, Queens' College, Cambridge for a non-stipendiary Fellowship, both held during the period that this work was done.

Extension	determinism	referential transparency	reference
Non-deterministic merge	no	no	[8]
Message passing functions	no	yes	[10]
Oracles	no	yes	[4]
I-structures	yes	no	[3]
Improving values	yes	yes	[5]
Bulk array operators	yes	yes	[9]

Figure 1: Extensions to pure lambda calculus for parallel computation

Many different extensions have been proposed which result in either better implementation of an existing functional construction, or greater semantic expressiveness, or both. Figure 1 lists some of these proposed extensions, together with their properties, and source references. The use of these extensions provides increased expressiveness at the cost of more complicated semantics.

Message passing functions and oracles provide a semantic extension to functional languages by allowing indeterminacy generated by different run-time evaluation orders to be seen by the programmer; however they preserve referential transparency.

I-structures, improving values, and bulk array operators do not introduce indeterminacy: this group of extensions provide efficient implementations in different cases where algorithms do not seem to have any efficient functional implementation.

In this paper we will investigate *aggregates*—data values which are constructed from multisets of individual elements. Thus the value of an aggregate does not depend on the order in which its elements are specified, and for any aggregate there is a corresponding commutative and associative (or aggregating) operation, which can be used to generate the aggregate of a union from the aggregates of its constituent parts.

We will be concerned with the efficient implementation of aggregates, and find that functional implementations of aggregate are unduly restrictive. The properties of operational implementations of aggregate will be investigated, and their use in functional languages found to underlie the second group of functional language extensions listed above.

The investigation here follows from my previous work in [7, pages 5–9] which considered the type of algorithm which can be expressed by an arbitrary sequential thread of global updates running through an otherwise functional concurrent program, and which is examined further in Section 3. Such algorithms are merely efficient implementations of certain types of aggregates.

In this paper we will see how operational algorithms represent an efficient implementation of semantics which may be expressed in the lambda calculus with no

extension. The efficiency results from a recognition that associative commutative (or aggregating) operations impose less constraints on evaluation order than can ever be represented in the lambda calculus.

I will start by examining the implementation of simple aggregating operations. In Sections 2 and 3 operations which perform aggregation using explicit local indeterminacy, for example in unique tag allocation, will be investigated. These correspond to sequential threads of updates. In section 4 the relationship between aggregates and other work extending functional languages for concurrent computation will be explored. Arvind's I-structures and Burton's improving values are both considered. Finally the operational specification of aggregates in functional languages is shown to introduce no loss of referential transparency, providing only that aggregate elements are implicitly allowed to 'piggyback' on primitive data objects.

1 Implementing aggregation

The simplest type of aggregate arises when objects are combined using an associative commutative operator (for example plus). Consider a recursive functional program to sum the leaves of a binary tree whose leaves are integers, by recursively summing subtrees and returning the sum of the results. This specifies, for any implementation, a tree of plus operations which must be evaluated to get the answer; this tree is the same shape as the datastructure to be summed.

We know that because plus is both commutative and associative, any tree of plus operations which has the same items at its leaves will give the same answer (indeed this rearrangement is regularly done by optimising compilers [1]), the functional specification of this computation has therefore over-specified its evaluation order.

The Church-Rosser Theorem states that any set of reductions of a lambda expression will, if it results in normal form, give the same answer. There is a corresponding (stronger) freedom reduction order for the evaluation of any associative commutative set of operations. In order to state this exactly it will be convenient to use multisets—sets with elements each of which has an positive integer indicating its multiplicity. Set union and difference are computed elementwise by adding and subtracting multiplicities, with the proviso that a non-positive multiplicity signifies no resultant element. Thus multisets represent collections whose elements need not all be different.

Let A be a multiset of objects, and $\circ$ an associative commutative operation. Thus the evaluation of any tree of $\circ$-operations with leaves A leads to the same answer. Now for any two elements $a, b \in A$ define a $\circ$ reduction

$$A \rightarrow (A - \{a, b\}) \cup \{a \circ b\}.$$

All sequences of $\circ$-reductions of A ending in an irreducible (single element) multiset are the same length, since each reduction decreases by 1 the size of A.

Furthermore all have the same result since any such sequence corresponds to a tree of o-operations whose leaves are A.

Thus aggregates represent computations which can be implemented concurrently with great freedom of evaluation. Note that the evaluation of an aggregate can be either sequential (a linear tree) or highly concurrent (a well-balanced binary tree). In the next section we consider aggregates whose sequential implementation can be effectively optimised.

Another implementation problem in functional languages concerns the simultaneous construction of multiple independent aggregates. It is important to recognise here that the operations on any one aggregate are sparse, and not perform redundant (trivial) aggregate combinations.

One example of this is the construction of a histogram from a data set. Each bar of the histogram is a separate aggregate, incremented by a small, dynamically chosen, number of the data items. Functional implementation of this requires an implementation of sparse vectors and sparse vector addition with considerable overheads.

Thus there are two distinct and fundamental problems when constructing aggregates in a functional language. Firstly, a functional aggregate does not adequately represent the real freedom of evaluation order in this operation. Secondly, aggregate intermediate results must be combined at every function which calls multiple aggregate generating subfunctions. This is wasteful if the generation of individual aggregate elements occurs infrequently.

2 Aggregates using local indeterminacy

In this section we will consider programming techniques which are commonly used in imperative concurrent programs and which at first seem incompatible with functional semantics. They correspond to the use of an object which is operated on throughout the program and whose internal state thus forms a sequential thread of execution. For an important class of such operations the meaning of the program is independent of the order in which operations are executed, so the operation introduces no constraint on concurrency other than the thread of sequential state updates of the object's state.

Let us consider a simple example of such an operation. The program of Figure 2 labels each leaf of a tree with a unique integer tag in such a way that the integers form a continuous interval [0,n], using a procedure assign_tag with internal state an integer which is incremented whenever a new tag is needed.

This program is written for an impure functional language, and is not referentially transparent. If all possible computations are shared (as is the case in applicative order evaluation) it gives the required result, and still does this if executed in a parallel evaluation order which preserves sharing. However individual tag values depend on the order in which the calls to assign_tag happen, so such a parallel evaluation order leads to program indeterminacy.

```
local current_tag = ref(0)
in
 fun assign_tag() =
   let val v = ! current_tag
   in
       ( current_tag := v+1;
         v
       )
   end
end;

datatype 'a tree = leaf of 'a | node of 'a tree * 'a tree;

datatype 'a tagged_tree =
               tagged_leaf of tagtype * 'a
             | tagged_node of 'a tagged_tree * 'a tagged_tree;

fun tag_tree( node(t1, t2)) =
          tagged_node(tag_tree(t1), tag_tree(t2))
        | tag_tree( leaf(x)) = tagged_leaf( assign_tag(), x);
```

Figure 2: Sequential tag distibution

How might this algorithm be expressed in a pure functional language? The program in Figure 3 uses a binary tagged-tree combining operation, merge_trees, and implements this algorithm without any use of side-effects.

This implementation has no sequential bottleneck, like assign_tag above, on the other hand it is much less efficient because merge_tagged_trees is O(n), and is called once for each node of the tree. It is purely functional and so is both referentially transparent and determinate.

It is natural to look for a way of combining the good properties of both these implementations. The indeterminacy of the sequential implementation can be hidden by restricting the operations which can be applied to tag values: for a typical use of tags there might be tag_equal which tests two tags for equality. The result of this is independent of the tag indeterminacy; if we provide a proof of this as part of the definition of the A-thread, and encapsulate tags so that no other operation can inspect them, we have a construction which introduces strictly local indeterminacy.

Indeterminate values which can be localised in this way to a few data types do not lead to the semantic problems of unconstrained indeterminacy. If we examine the equivalent functional program merge_trees we see that the resulting tag values, while determinate, can only be discovered from global analysis of the entire program. They bear no simple relationship to the program structure and so are no clearer than those in the truly indeterminate program.

```
datatype 'a tree =
   leaf of 'a
 | node of 'a tree * 'a tree;

datatype 'a tagged_tree =
   tagged_leaf of int
 | tagged_node of 'a tagged_tree * 'a tagged_tree;


fun tag_tree( leaf(x)) = tagged_leaf(x, 1)
  | tag_tree( node(x,y)) =
        merge_tagged_trees( tag_tree x, tag_tree y)

and merge_tagged_trees( x, y) =
    let
     val n = largest_tag_in x
    in
     tagged_node(x , add_to_tags( y, n+1))
    end

and largest_tag_in x = ....   (* return largest tag in tree x *)

and add_to_tags( x, n) = .... ;(* return copy of x with n added
                                  to each tag *)
```

Figure 3: Functional tag distribution

The function merge_trees is almost an aggregating operation. If tag values are encapsulated it is externally both associative and commutative, and tag distribution is another example of an aggregating operation. The operational implementation in tag_assign takes sequential time $O(\log n)$ times less than the equivalent functional implementation.

Tradeoffs between sequential and concurrent implementation

We now contrasted three distinct implementations of unique tag distribution:

1. As pure functional aggregate: combine in a fixed tree using a merge operation
2. As a rewrite rule defined aggregate: combine in an arbitrary tree using a merge operation
3. As a sequentially defined aggregate: combine in an arbitrary sequence using an update operation with internal state.

The sequential implementation is most efficient, but introduces a sequential thread of execution which limits total concurrency. One strategy to manage this would be for one processor would to act as a server performing global tag allocation. Alternatively by using snoopy cacheing of the internal state variable of the allocator, allocation could be done locally.

These two strategies both impose a sequential thread on execution, but are optimal in different circumstances. The cost of tag allocation may be estimated by considering the bandwidth (or frequency) at which tags may be allocated, and the latency introduced between calls of tag_assign and use of the resulting as an argument to tag_equal.

If the frequency (or bandwidth) of tag allocations exceeds that which can be processed this sequence will form a critical execution bottleneck. Otherwise the latency of tag allocation will reduce available program concurrency.

If the program is such that all tags are assigned a long time before they are used then this latency will not limit execution. Otherwise, it will have the effect of reducing available program concurrency. Let us in this case distinguish between bandwidth and latency limited computation, which will be defined as follows. Suppose a multiprocessor has N processors. At time t let

$$\begin{aligned} n(t) &= \text{No. executable threads of computation} \\ L(t) &= n(t)/N \end{aligned}$$

Whenever the system loading $L(t) < 1$ computation will be called latency limited, if $L(t) > 1$ the computation will be called bandwidth limited. A fuller discussion of this may be found in [6].

When computation is sufficiently bandwidth limited the latency of tag assignment does not restrict performance, so assignment bandwidth is the most important determinant of performance. Conversely when computation is latency limited, tag assignments which lie on the computation's critical path do limit performance.

When tag assignment is made from a single processor inter-processor communication latencies affect assignment latency, but not assignment bandwidth. When assignment is done locally communication latencies affect assignment bandwidth and latency, but only in the case that the state variable has to be swapped between processors. Consequently local tag assignment will never be optimal for bandwidth limited computation, and may (where successive tag assignments are done by the same processor) be optimal for latency limited computation.

If we assume that computation is bandwidth limited only tag assignment bandwidth need be considered, and sequential assignment will prove a performance bottleneck when this is insufficient. We may eliminate this bottleneck without abandoning the advantages of sequential allocation by using an implementation which combines sequential allocation nodes in a global allocation tree of a shape determined by communication topology and depth by the desired maximum allocation bandwidth.

Consider a tree whose leaves assign tags locally, and along the arcs of which assignment and allocation messages are sent. Every assignment request must pass

up the tree to the root, an appropriate allocation is then made and the correct tag value passed down to the requesting leaf. The sequential bandwidth bottleneck at the root of the tree is avoided by allowing each node of the tree to combine simultaneous requests into a single request for contiguous multiple tags. The resulting tags are then apportioned appropriately amongst the requests.

This strategy has been used in multiprocessors for resource allocation, and implemented in hardware using a combining network [2]. We see here that it is an example of global communication in concurrent programs which corresponds semantically to an associative commutative operation with provably local indeterminacy.

3 A-threads

Algorithms which construct aggregates sequentially using a set of calls to a function with side-effects, but in such a way that the order in which these function calls happen does not matter, are often used in languages like LISP. The unique tag distribution example identifies such an algorithm, where the global structure (next_tag counter), is only used to mediate globally coherent tag allocation.

An aggregate like those considered in the first section can also be implemented in this way. The aggregate value is held as a global variable, updated by every function call which denotes a new aggregate element, and finally read. In this case data is communicated from threads of a concurrent program upwards to some eventually accessible global structure, in the other communication occurs from a global structure down to the concurrent threads.

It is possible to combine these types of communication, for example in a global symbol table which is constructed in lexical analysis and returns tokens consisting of numeric references into the table. I will call the structure which mediates this two-way communication an A-thread, and note that the structures discussed in the previous two sections are special cases of it.

Informally we define an A-thread to be a function with side-effects as follows:

- An A-thread A is initialised creating a new globally updatable variable A_s of type S ref with initial value ref s_0' the A-function of A, f of type $X \to R$. f will be defined by two functions without side-effects:

$$f_{value}\ X \times S \to R$$

$$f_{update}\ X \times S \to S$$

  such when f is called with argument x and A_s containing s, it returns value $f_{value}(x, s)$ and sets A_s equal to $f_{update}(x, s)$.

- After no more calls of f are possible (this could be determined by keeping reference counts of the value of f) the final value of A_s becomes accessible to the program using the A-thread.

In order for an A-thread to be well defined it is necessary for calls to f to be semantically commutative; all functions with arguments of types R or S (except those in the definition of f) must return identical values in a given program for every permutation of calls to f. This requirement can be fulfilled by making R, S encapsulated data types and proving determinacy of the result of every externally accessible function using them under permutations of calls to f.

It is not difficult to show that every A-thread is in fact (if the encapsulated data types R, S are never directly inspected, an aggregation.

Suppose

4 Early termination of aggregate construction

So far we have considered concurrent construction of a global data object from a multiset of elements, and seen that a pure functional representation of this operation is insufficient to express the freedom of evaluation order intrinsic in this operation.

A related question to consider is at what time the globally constructed object can be read. In the simple case of a global sum this must be after all aggregate computation has happened, but in general this is too restrictive.

One particular class of associative commutative operations is that of joins on partial orders. The join of a, b is the minimal x such that $x \geq a$ and $x \geq b$. If the partial order has maximal elements then once these are reached no further construction can change them: any maximal element may therefore be read before aggregate computation terminates.

Burton [5] has investigated an extension to functional language semantics which he calls an *improving values*, in which a join generated aggregate may be used as the argument to a meet function, with optimal timing. Thus the meet operation with argument an aggregate will return a value as soon as the aggregate is as big as its other argument.

Another related extension is Arvind's use of I-structures [3], top level vectors which have values determined by assignment operations. The order in which valid assignments are made to an I-structure does not affect its value, so this is clearly an example of an aggregate. However multiple identical assignment operations result in an error, so I-structures can't be represented by joins. We will see in the next section that this stops them from being referentially transparent.

5 Aggregates and referential transparency

An aggregate is constructed by specifying the elements of a multiset. If this is done using explicit multiset union operators in a functional language the construction is clearly referentially transparent. We have seen that in some cases an implementation of aggregates using a function with the side effect of creating a new aggregate element is more efficient than the functional equivalent. If aggregates are treated

specially in the language a functionally specified aggregate can be implemented operationally. Let us now see whether the operational specification of an aggregate, which seems very natural when writing for an applicative order functional language, can be used without destroying referential transparency.

Within a lambda calculus the set of function applications $S(p)$ on which a given primitive value p is strict is well defined, however different evaluation orders will result in applications in $S(p)$ being evaluated a different number of times. If the sequential operation constructing an aggregate is idempotent, or, equivalently, the aggregate is dependent only on its set (not multiset) of elements, then the operational representation of aggregation can be incorporated into the language without destroying referential transparency. It is equivalent to a purely functional program in which the domain of primitive objects has been extended by a Cartesian product with the set of possible aggregate values, and all operations strict on primitive objects are redefined so as to combine and propagate the aggregate values in their arguments to their results.

A rich source of aggregates which allow referentially transparent operational specifications are partial order joins, which result in an element addition operation which is idempotent. Burton's improving values are a special case of this class of aggregates. Arvind's I-structures are however not referentially transparent because multiple identical writes to a single slot result in an error. This could be changed, as Arvind has observed in [3], and then I-structure construction would be similar to logic variable unification. There remains a problem in the representation of I-structures as aggregates. In order to enable efficient concurrent computation an I-structure slot can be read as soon as it has a value assigned, a subsequent (different) assignment then results in a global error. In order for this to be a true aggregate the result of a slot (error or given value) should not be available for reading until no possiblity of changing it exists, a delay which would make I-structure use very inefficient.

The solution is another implementation optimisation: after one value has been assigned to a slot all resulting computation must be scheduled eagerly, on the assumption that a subsequent error is exceptional. However the computation can only be printed out when it is known that no error will happen (after all possible I-structure construction). This optimisation can itself be implemented with another aggregate which has two possible elements, the error indication, together with a suitable error message value, and the value which a normal computation would compute.

Finally the use of an aggregate to encapsulate eager computation raises another implementation issue: the killing of this computation whenever it is known that it will not contribute to the aggregate result. A general resolution of this would include efficient eager OR computation, in which the true termination of one term of an OR clause kills all other terms. This is a subject for further research.

Where the aggregating operation is not idempotent it is still possible to preserve referential transparency and use a function f to denote elements of an aggregate.

This, as in the case of idempotent operations, is done by making the calls to f associated with the computation of every inspectable data value explicit.

Suppose that the domain of inspectable data values, D, is defined by:

$$D = P + D \times D$$

Let elements of A be multisets of calls to f. We define an extended domain:

$$D' = P \times A + (D' \times D') \times A$$

If $x = (d, a) \in D'$ we call a the A-component of x, and d the value of x. Now we transform a lambda expression over D into an equivalent one over D' by extending all primitive functions over D to new functions over D' with the same values and A-components as follows:

- The A-component of the result of a primitive strict function is the union of the multisets of its arguments.
- The A-component of a data constructor (like cons) is the union of the A-components of its arguments.
- Conditional expressions return as A-component that of the expression which is selected.
- A new primitive, eval

This gives us a well defined and referentially transparent notion of what a function call of f is. It is one in which calls to f are never shared, so:

```
let f1 = ....
in f2 = (f1, f1)
end
```

will define a function f2 with precisely twice the number of calls to f which f1 has.

Thus this extension of the lambda calculus is perfectly consistent but is unusual in its treatment of shared data objects.

6 Conclusions

In this paper we have looked in great detail at the mechanics of aggregation in concurrent functional languages. An aggregate is a global structure whose meaning depends on a multiset of (independently specified) elements. In functional languages aggregates are constructed concurrently using trees of commutative associative operations. In imperative languages aggregates can also be generated concurrently from a sequence of atomic update operations, where each update adds an element to the multiset being constructed.

The functional implementation of aggregating (commutative associative) operations results in an unnecessarily restrictive evaluation order. An arbitrary tree of operations will lead to the same result. The imperative implementation is one where the tree of operations is linear, so that each operation adds just one element to the aggregate. This may allow simpler implementation than an arbitrary aggregate merge. Also, trivial operations, in which an aggregate is combined with an empty aggregate, need never be performed in an imperative implementation. This leads to much greater efficiency when aggregates are constructed sparsely (that is, the number of aggregate elements is much smaller than the number of functions which could potentially generate elements).

We have seen that some aggregate operations, A-threads, can be more efficiently implemented using provably local indeterminate values and an arbitrarily ordered but sequential thread of updates. In special case of unique tag allocation it is possible to choose an implementation which freely combines linear and arbitrary shaped trees of operations. This technique has proved so useful that it is supported by special communications hardware in some multiprocessor designs.

A special class of aggregates have unusual termination properties, in that the value of the aggregate can be inferred before aggregate construction is completed. These correspond approximately to a number of functional language extensions found useful to express concurrent computation.

Finally we have shown that operational specification of an aggregate need not destroy referential transparency. If aggregates are explicitly identified, an functional specification can always be transformed into the equivalent operational implementation, with its advantages of efficiency, freer evaluation order, and early termination. However an operational specification may be syntactically easier, especially when the aggregate is sparse.

It is not surprising that aggregates play such an important rôle in concurrent functional languages. They are precisely the operations which can be unambiguously and globally specified without giving potentially concurrent operations a predefined order.

It would be interesting to determine exactly what class of extensions are necessary to make concurrent functional languages truly general purpose. Aggregates cover the cases where either global updates, or evaluation-order related indeterminacy, is used when in a way which is not intrinsic to the semantics of the computation.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

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

[3] Rishiyur S. Nikhil Arvind and Keshav K. Pingali. *I-structures: Data structures for Parallel Computing.* Computation Structures Group Memo 269, M.I.T., 1987.

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

[5] F. Warren Burton. *Indeterminate Behaviour with Determinate Semantics in Parallel Programs.* Technical Report CSS/LCCR TR 98-03, Simon Fraser University, 1989.

[6] T. J. W. Clarke. The D-RISC—an architecture for use in multiprocessors. In *Proc. Functional Programming Languages and Computer Architectures 1987, Oregon, USA.*, pages 16–33, Springer Verlag, 1987.

[7] T. J. W. Clarke. *General Theory Relating to the Implementation of Concurrent Symbolic Computation.* Technical Report 174, The University of Cambridge Computer Laboratory, August 1989.

[8] P. Henderson. *Purely Functional Operating Systems*, pages 177–189. Cambridge University Press, 1982.

[9] R. M. Keller. *FEL (Functional Equation Language) Programmer's Guide.* Technical Report AMPS Technical Memorandum No. 7, Dept of Computer Science, University of Utah, 1983.

[10] William Stoye. *The Implementation of Functional Languages using Custom Hardware.* Technical Report 81, University of Cambridge Computer Laboratory, December 1985.