Indeterminate Behavior with Determinate Semantics in Parallel Programs*

F. Warren Burton
School of Computing Science, Simon Fraser University
Burnaby, British Columbia, Canada V5A 1S6

burton@cs.sfu.ca

ABSTRACT
A parallel program may be indeterminate so that it can
adapt its behavior to the number of processors available, or
at least so that low level timing issues are removed from the
program.

Indeterminate programs are hard to write, understand,
modify or verify. They are impossible to debug, since they
may not behave the same from one run to the next.

We propose a new construct, a polymorphic abstract data
type called an improving value, with operations that have
indeterminate behavior but simple determinate semantics.
These operations allow the type of indeterminate behavior
required by many parallel algorithms. Operationally, we
may know a lower bound (or upper bound) for an improving
value at any given time. If this bound is sufficient, we
may act on it. Otherwise, the bound may improve as the
computation proceeds.

We define improving values in the context of a functional
programming language, but the technique can be used in
procedural programs as well.

1 INTRODUCTION
A parallel program may be indeterminate so that it can
adapt its behavior to the number of processors available, or
at least so that low level timing issues are removed from the
program.

For example, in a combinatorial search, many different
processes may be searching different subspaces in parallel.
These processes may all access and update a global variable
that gives information on the best solution found so far.
The current value of the variable may be used to determine
if a subspace may be pruned from the search. Since the
processes are not synchronized, the pruning of a particular
subspace may depend on just when a shared variable is read.
This causes indeterminate behavior, although the final result
may be determinate.

Indeterminate programs are hard to write, understand,
modify or verify. They are impossible to debug, since they
may not behave the same from one run to the next.

We propose a new construct, a polymorphic abstract data
type called an improving value, with operations that have

*This work was supported by the Natural Science and Engi-
neering Research Council of Canada.

indeterminate behavior but simple determinate semantics.
These operations allow the type of indeterminate behavior
required by many parallel algorithms. Operationally, we
may know a lower bound (or upper bound) for an improving
value at any given time. If this bound is sufficient, we
may act on it. Otherwise, the bound may improve as the
computation proceeds.

We define improving values in the context of a functional
programming language, but the technique can be used in
procedural programs as well. We will use the notation of
the Miranda functional programming language [7, 8, 9].

In section 2 we will briefly review the concept of specula-
tive evaluation. The improving value abstract data type will
be introduced in section 3. Several examples of the use of
improving values are given in section 4, including a parallel
least-cost search algorithm that is only four lines long. In
these three sections we will limit our attention to two types
of computing devices: Those with a single processor and
those with an infinite number of processors. Since machines
of the second type are not yet on the market, in section 5
we will consider how to make best use of a limited number
of processors. The least-cost search algorithm will again be
considered. An axiomatic definition of improving values is
given in section 6, along with several properties of improving
values, and an outline for a simple correctness proof for the
least-cost search algorithm. Section 7 is the conclusion.

2 SPECULATIVE EVALUATION
Often it is necessary for a good parallel algorithm to per-
form more work in total than would be performed by a good
sequential algorithm for the same problem. This is because
the best sequential algorithm may not have a very high po-
tential for parallelism. In other cases, the sequential algo-
rithm may have a high potential for parallelism, but only if
some work is done before it is known to be required. We are
interested in algorithms of this second kind.

Recall that, by definition, any problem in NP can be
solved by a nondeterministic Turing machine in polynomial
time. Consider any NP-complete problem, and any nonde-
terministic polynomial time algorithm to solve the problem
where none of the nondeterministic choices lead to a non-
terminating computation. A sequential algorithm for this
problem can be produced by simulating the nondetermi-

1Miranda is a trademark of Research Software Ltd.
spec-or $a \ b = \text{spec} \ (\text{or} \ a) \ b$

where

or $a \ b = a \lor b$

\[
\begin{align*}
\text{start} \ [] & = [] \\
\text{start} \ (x:xs) & = \text{spec} \ (\text{spec cons} \ x) \ (\text{start} \ xs) \\
\text{where} \\
n \text{cons} \ a \ b & = a:b
\end{align*}
\]

Figure 1: Two useful functions for initiating speculative computation.

other sequential algorithm can solve this problem in polynomial time in the worst case, assuming $P \neq \text{NP}$.

With unbounded parallelism, we can solve any NP-complete problem in polynomial time by considering all of the alternatives of the nondeterministic choices in parallel. Since no sequential algorithm has a worst case polynomial time solution, all NP-complete problems have a high potential for parallelism. On the other hand, the backtracking algorithm could go directly to the solution, solving the problem as quickly as with unbounded parallelism, in the best case. Hence we have an example of a problem where we can gain speed through the use of parallelism, in the average case, but only if we are willing to perform some work that may not be required.

The use of speculative evaluation [1, 2, 3] has been proposed for problems such as this. A speculative computation is a computation that may or may not be required later. For example, while considering one alternative in a backtracking algorithm, speculative computations may be exploring other alternatives.

All computation that is not speculative is called mandatory. If a speculative computation is found not to be required, then it may be terminated. If a speculative computation is found to be required, then it must be upgraded to mandatory. Mandatory computation must be favored over speculative computation, at least to the extent that some mandatory computation is always progressing. In the degenerate case of a single processor, no speculative computation will ever be performed. With unbounded parallelism, all speculative computation will be performed, at least until it is found to be not required.

A single function spec is sufficient to introduce speculative evaluation. The type of spec is

\[
\text{spec} :: (\star \rightarrow \star \star) \rightarrow (\star \rightarrow \star \star)
\]

and semantically it is the identity function restricted to functions. That is

\[
\text{spec} \ f \ x = f \ x
\]

Operationally, spec will initiate the speculative evaluation of its second argument before applying its first argument to its second.

Two useful functions are defined in Fig. 1. The function spec-or is defined in terms of the conditional or operator, V. When applied, spec-or will evaluate its first argument as a mandatory computation and its second argument as a speculative computation. If the first argument returns \text{True} then the speculative computation may be terminated by the implementation. If the first argument evaluates to \text{False} then the speculative evaluation must be upgraded to mandatory if it has not already completed. This function could be useful in a simple backtracking algorithm that returns a boolean result. The function start will initiate the speculative evaluation of all of the elements in the list to which it is applied. In Miranda, the colon is an infix cons operator.

Examples of speculative algorithms may be found in [2].

3 IMPROVING VALUES

Often in combinatorial search algorithms, bounds are maintained to help decide when pruning is possible. For example, in a branch-and-bound search for a least cost solution, a program may keep the value of the best solution found so far. If it is possible to establish that a subspace cannot return a better solution, it can be pruned. Alpha-beta search, and similar algorithms, also maintain bounds. In all of these cases, the values of the bounds change monotonically as the computation progresses. We will limit our attention to lower bounds. Upper bounds can be handled in a similar manner.

We proposed the use of the polymorphic abstract data type improving for a lower bound that may improve (become a tighter bound) over time. The type signature is given in Fig. 2.

The functions make and break are type transfer functions, and minimum and spec-max compute the minimum and maximum of improving values, respectively, with some added laziness.

We will be able to use the result returned by spec-max before its second argument has been evaluated if its first argument provides sufficient information, just as we are able to use the result returned by spec-or before its second argument has been evaluated, provided its first argument is \text{True}. In general, once the first argument of spec-max has been evaluated, we will have a lower bound on the result. If this is not sufficient, we can wait for the bound to improve, which will happen when the value of the second argument is available.

For example,

\[
\text{break} \ (\text{minimum} \ (\text{make} \ 5) \ (\text{spec-max} \ (\text{make} \ 7) \ \bot)) = 5
\]

Here \bot denotes an undefined value or a nonterminating computation. We cannot test a value for equality with \bot, since the halting problem is undecidable. In practice, we are not really concerned with handling nonterminating computations or undefined values. However, if any subexpression of an expression can be replaced by \bot without changing the value of the expression, then the subexpression need not be evaluated.

In practice, it is better to parameterize the improving abstract data type with an order relation. This will permit improving values to be used for upper bounds as well as lower values.

341
improving * == [*]

make a = [a]
break x = last x

spec max x0 y0 = spec (monotonic-append x0) y0

minimum x y = short-merge x y

monotonic-append x y = x : dropwhile (<= x) y

short-merge [] [] = []
short-merge [ ] [ ] = []
short-merge [ ] (y:y's) = []
short-merge (x:x's) [ ] = []
short-merge (x:x's) (y:y's) = x:short-merge x y's, if x < y
= x:short-merge x y's, if x = y
= y:short-merge x:x's y, if x > y

Figure 3: A simple implementation of improving.

bounds. This also will support applications that require order relations other than those built into the language. This change is straightforward, but complicates the presentation slightly, so will be omitted. In this paper, for simplicity, we will rig all of our examples so that the "<" operator, which is provided for all types in Miranda, gives the desired ordering.

An improving value can be represented by a strictly increasing list of approximations. The final element of the list will be the true value on the previous list elements approximate. Infinite lists and partial lists will be considered in section 6. We will assume that for any two values a and b, the value of a < b is defined.

With this representation, the improving abstract data type can be implemented as shown in Fig. 3. The function make produces a singleton list, and break returns the last element of a list. The spec max function starts the speculative evaluation of its second argument and then appends its arguments, removing any values in the second list that are less than or equal to the final element of the first list. This insures that the list of approximations remain strictly increasing. On the other hand, minimum merges two lists, removing duplicates in order to maintain strict monotonicity. In addition, the merge ends as soon as either list ends. For example,

minimum (make 4) (spec max (make 3) (make 5))

will cause the lists [4] and [3, 5] to be merged to produce [3, 4], with the 5 thrown away. As a second example,

minimum (make 4) (spec max (make 5) (make 3))

will result in the two singleton lists [4] and [5] being merged to produce the singleton list [4]. The 3 will be discarded by spec max since it would be out of order, and the 5 will be discarded by minimum as before.

We will sometimes refer to the elements of a list representing an improving value as a progress report. Each progress report gives some new information about the final value of an improving value.

It is easy to see that every element of a list representing an improving value, except for the final element of the list, must at some time have occurred in a left argument of an application of spec max. This property is trivially true of the singleton lists produced by make, and is preserved by minimum and spec max. Often it is useful to use the expression spec max a b in situations where b is the desired result, but a is known to be a lower bound on the result. In this case, a is a progress report.

If the abstract type improving is implemented as a language primitive, then a more efficient implementation than the one described above is possible. Each improving value is represented by a pair consisting of the best approximation found so far and a flag indicating whether this is a final value. Each improving value has an associated process that will update the approximation as required and supply the latest value to other processes upon request. This saves storing more than one value and saves other processes the cost of examining out of date values before coming to the most recent value. In some cases this can significantly reduce the amount of communication between processes.

4 EXAMPLES

In this section we will consider parallel versions of three search algorithms.

4.1 Least-Cost Search

With many combinational problems it is necessary to search a solution space for the best solution to the problem. We will assume that "best" means "smallest" and call the measure of a solution the cost of the solution. Horowitz and Sahni present a least-cost search algorithm for this problem [6]. Assuming the costs of leaves and lower bounds for other nodes are all distinct, this algorithms expands exactly the same nodes as the A* algorithm in the case of a single processor, which is optimal [6].

We will assume that the solution space is organized as a tree with all possible solutions at leaves. For any node, node, is leaf node is a bool indicating whether or not the node is a leaf. For any leaf node, cost node is the cost of the solution. If the node is not a leaf, then children node is a list of the children of node and lower bound node is a lower bound on the least cost solution to be found in the subtree rooted at node. Finally, nil node is a special node such that nil node < node for any node, node, that may be encountered during a search. Otherwise nodes are ordered arbitrarily.

Let us first consider a simple exhaustive search algorithm to solve this problem. Such an algorithm is given in Fig. 4. The function returns an ordered pair, consisting of the cost of the best solution together with that solution. Note that if cost1 < cost2 then (cost1, node1) < (cost2, node2) in Miranda. We should note that "<" is an infix function composition operation and foldl1 is a function that will apply a binary function to elements of a list, reducing the list to a single value (e.g. foldl1 (+) x's will compute the sum of the elements of the list x's). We will use mini and maxi for binary minimum and maximum functions, respectively. The function map will apply a function individually to elements of a list, producing a new list.

Fig. 5 gives an equivalent (but slightly less efficient) version of this search. We know that lower bound root is a lower bound on the cost of the node returned by searching the subtree rooted at root and, in case it is a tight lower bound, nil node is less than any other node. It follows that
search root
= (cost root, root), if is_leaf root
= (foldrl mini.map search.children)root, otherwise

Figure 4: An exhaustive search algorithm.

(lower_bound root, nil-node) <
(foldrl mini.map search.children) root

so the two algorithms must return the same result.

A minor further modification, to introduce improving values, yields the least-cost search algorithm in Fig. 6. Notice how the search of each subtree starts by providing a progress report in the form of a lower bound on the best solution to be found in the subtree. If a better solution has been found anywhere in the tree, this progress report is sufficient for the subtree to be pruned.

In the case of a single processor, where no speculative evaluation is performed until it becomes mandatory, only those nodes with a lower bound less than or equal to the cost of the optimal solution can ever be expanded. Hence, in the sequential case this algorithm is optimal with respect to the number of nodes expanded, assuming all costs and lower bounds are distinct. With an unbounded number of processors, all paths are searched in parallel, at least until a least cost solution is found.

4.2 Breadth-First Search

Fig. 7 shows a parallel breadth-first search algorithm. The algorithm takes a node as a parameter and returns a pair consisting of the depth of a least deep solution and the solution itself. We assume that we are given two functions: is_solution, which determines whether a node is a solution to the problem of interest, and children, which will generate a list of the children of a node. We do not require that a solution be a leaf node. As with the least-cost search algorithm, we assume the existence of a node nil_node. We also assume the existence of a value infinity which is greater than the depth of any reasonable search. This simplifies the algorithm and allows us to use the built-in order relation "<". Notice that we have used the function start defined in Fig. 1 to initiate the parallel searching of all subtrees when the subtree root has been found not to be a solution. A subtree is pruned whenever a progress report is sufficient to eliminate it from consideration. That is, if the local search is at a greater depth than a know solution, it is terminated.

4.3 Alpha-Beta Search

As a final example we will consider a parallel alpha-beta search algorithm for searching a game tree, based on the sequential alpha-beta search algorithm given in [5]. For simplicity, the algorithm returns the value of the best move, not the move itself. Again start is used to initiate the searching of subtrees. The algorithm is given in Fig. 8.

We assume the existence of three functions. The function is_leaf determines whether a node is a leaf. For leaf nodes eval computes the value of the node to the player whose turn it is, and for other nodes children computes the children of the node (of which we assume that there is at least one.)

The function scan is a commonly used function, similar to foldrl, but returning a list of “partial sums” rather than just the final “sum”. It is defined by

\[
\text{scan}\ f\ a\ z\ =\ a :\ \text{scan}'\ f\ a\ z\ \\
\text{where}\ \\
\text{scan}'\ f\ a\ []\ =\ [] \\
\text{scan}'\ f\ a\ (x:xs)\ =\ scan\ f\ (f\ a\ x)\ xs
\]

The function zip2 maps two lists into a list of corresponding pairs, ending as soon as either list ends.

The expression (alpha_beta node alpha beta) searches for the value of the best move for the player whose turn it is, subject to the constraint that only moves with value between alpha and beta are considered. We know that by making a different move, we can get to a position with value at least alpha, and also know that our opponent can keep us from getting to a position of value greater than beta by making a different move earlier. The initial call is of the form

\[
\text{alpha-beta root} \ (-m)\ m
\]

where m is chosen such that for any position, p,

\[-m < \text{eval}\ p < m.\]

The list alphas is the list of alpha values that result after the search of each child. The recursion allows each element of this list to be used as a bound in the computation of the next element.

Parallel algorithms for minimax searching is an active area of research. This simple parallel alpha-beta algorithm is probably not the best solution to the problem. Our notation makes it easier to understand and verify algorithms, but fundamental problems of finding the best algorithm for a given problem remain.

5 PRIORITIES

If all computation is mandatory, then scheduling is not a difficult problem, assuming a shared memory of sufficient size, so we do not need to worry about communication between processors or running out of memory. There are simple scheduling algorithms [4] that are, in the worst case, within a factor of two of being optimal, where we measure the quality of a scheduling algorithm by the time required to finish all computation.

This is not true with speculative computation. If we have n processors and a potential for parallelism that is much higher than n, then it is possible for some problems to get a speedup approaching n if all processors do mandatory work or speculative work that will later become mandatory almost all of the time. On the other hand, if one processor does mandatory work and all other processors spend almost all of their time on speculative computation that will prove to be unneeded, then the speedup may not be much greater than one. Clearly, given a limited number of processors, we prefer to do that speculative work that is most likely to be required later. In general, it is not possible for the implementation to determine which speculative computations are the most worthwhile.

The solution to this problem is to let the programmer specify priorities. We can introduce a new function, priority of type num -> * -> * . The semantics of priority are

\[\text{priority}\ n\ x = \bot, \text{if } n = \bot\]
\[= x, \text{otherwise}\]

If priority is called within a speculative computation, it initiates a new speculative computation with priority n, where n is any number, to compute x. The higher the value
search root
= (cost root, root), if is_leaf root
= maxi (lower_bound root, nil_node) ((foldr1 mini.map search.children) root), otherwise

Figure 5: A modified exhaustive search algorithm.

search = break.search'

search' root
= make (cost root, root), if is_leaf root
= spec_max (make (lower_bound root, nil_node)) ((foldr1 minimum.map search'.children) root), otherwise

Figure 6: A least-cost search algorithm.

breadth_first = break.search 0
search depth root
= make (depth, root), if is_solution root
= make (infinity, nil_node), if kids = []
= spec_max progress_report solution, otherwise

where
kids = children root
progress_report = make (depth, nil_node)
solution = foldr1 minimum (start [search (depth + 1) k | k <- kids])

Figure 7: A parallel breadth-first search algorithm.

alpha_beta root alpha beta
= eval root, if is_leaf root
= break (minimum (make beta) best), otherwise

where
best = (foldr1 spec_max.map make) alphas
alphas = spec (scan maxi alpha) (start searches)
s searches = [− (alpha_beta child (− beta) (− new_alpha))] (child, new_alpha) <- zip2 (children root) alphas]

Figure 8: An alpha-beta search algorithm.
of $n$, the higher the priority of the speculative computation. All speculative computations started with `$spec$` have higher priority than any started by `$priority$`. The priority of a speculative computation cannot be changed, except that the speculative computation may become mandatory. In particular, when one speculative computation finds it needs the result of another, possibly lower priority, speculative computation in order to proceed, it must wait for the second speculative computation to finish (unless, of course, the waiting computation becomes upgraded to mandatory.) Of course, if `$priority$` is invoked by a mandatory computation, then the resulting computation will immediately become mandatory, since it would not have been invoked if its result were not needed.

As an example, we could modify the least-cost search algorithm in Fig. 6 by changing the subexpression

\[(foldr1 \text{minimum.map search}.\text{children}) \text{root}\]

to

\[(\text{priority }-(\text{lower_bound root})).
foldr1 \text{minimum.map search}.\text{children}) \text{root}\]

so that nodes that are more promising (have a smaller lower bound on the least cost solution) are expanded with higher priority. (Of course, we would want to factor out the computation of `$\text{lower_bound root}$` and compute it only once.)

6 FORMAL PROPERTIES

We will assume that the values used in improving values are finite in size and come from a flat domain. That is, values are either completely defined or are `$\bot$`. An example of a nonflat domain would be the domain of lazy lists. For example, the lazy list `$1::L$` has `$1$` as its first element, but any attempt to evaluate the tail of the list will result in a nonterminating computation or an undefined result.

If we allow values from a nonflat domain to be improving values, we have some awkward special cases. For example, with lists ordered in the obvious way,

\[
\text{break (minimum (make (1::L)))} \\
\text{(spec.max (make (1::L)) (make (2::L))))} \\
= \bot,
\]

since the comparison of `(1:1)` with `(1:1)` will not terminate. On the other hand,

\[
\text{mini (1:1) (maxi (1:1) (2:1))} = 2:1.
\]

In this case, `$maxi$` is applied first and the nonterminating comparison does not arise. Similar problems can arise with infinite lists. We would like the improving value operations to be at least as well defined as the corresponding operations on ordinary values.

Sometimes it will be useful to have improving values where values come from a flat subdomain of a nonflat domain. For example, we might want to use improving lists in a context where we know that all lists will be finite and well defined. In section 4, we used improving ordered pairs. The domain of ordered pairs is not flat, since it includes elements such as `(1, \bot)`. However, we only used fully defined ordered pairs. In cases such as these, we must restrict ourselves to a flat subdomain. If type `$\text{improving}$` is implemented as a primitive in a language, the restriction to finite, fully defined values can be enforced by requiring `$\text{make}$` to fully evaluate its argument. The results in the remainder of this section depend on values being finite and either fully defined or completely undefined before `$\text{make}$` is applied.

With the implementation of type `$\text{improving}$` given in Fig. 3, a number of different lists may all represent the same abstract object. Recall that lists are strictly increasing sequences of improving approximations. If `$a$` and `$b$` are any two well defined values with `$a < b$` then it is not possible to distinguish `$a:b:2` from `$b:2` provide `$b:2` is a valid representation for an improving value. Since `$\text{spec.max}$` and `$\text{minimum}$` both examine list elements sequentially, lists of the form `$zs ++ \bot ++ ys$`, where "$++$" is an infix append operator, cannot be generated, although lists of the form `$zs ++ \bot` can be produced. Finally, while both `$\bot$` and `$\bot$` are possible representations for improving values, `$a:1` is not a possible representation. For example, `$\text{spec.max (make a)} (\text{make } \bot)` will produce an improving value represented by `$a:1$`, even though `$\text{make } \bot$` produces `$\bot$`.

With these observations, we can divide representations into equivalence classes. We will let `$a` represent the class of all finite, fully defined lists with final value `$a$`. If `$zs` is a member of the equivalence class `$a` then we will represent the equivalence class that includes `$zs ++ \bot` by `$a:?$`. We will read `$a` as "exactly `$a$", and `$a:?$` as "at least `$a$". Both `$\bot$` and `$\bot$` belong to the same equivalence class which we will represent by `$\bot$`.

Finally, we have two cases to consider with infinite lists. An infinite list having no upper bound on the size of its elements is in an equivalence class represented by `$\infty$`. An infinite list where the elements have a least upper bound, `$a`, is a member of the equivalence class `$a` which is read "almost `$a$". For example `$\text{foldr1 spec-max (map make [1..])}$` will generate `$\infty$`.

Axioms for `$\text{improving}$` values are given in Fig. 9. In this figure, `$a$` and `$b$` may represent any values and `$z$` may represent any improving value.

With these axioms, we can prove a number of interesting properties. These include:

\[
\text{make (mini a b) } = \text{minimum (make a) (make b)} \quad (1)
\]

\[
\text{make (maxi a b) } \sqsupseteq \text{spec.max (make a) (make b)} \quad (2)
\]

\[
\text{mini (break a) (break b) } \sqsubseteq \text{break (minimum a b)} \quad (3)
\]

\[
\text{make (break a) (break b) } = \text{break (spec.max a b)} \quad (4)
\]

\[
\text{break.make } = \text{id} \quad (5)
\]

\[
\text{make.break } \sqsubseteq \text{id} \quad (6)
\]

We can use these properties to prove the correctness of efficient combinatorial search algorithms using `$\text{improving}$` values. To show that an implementation meets its specification, we show that `$\text{specification} \sqsubseteq \text{implementation}$`. That is, where the specification is defined, the implementation must agree with it, but the implementation may be stronger. For example, we may take an exhaustive search algorithm as a specification for a more efficient search algorithm that avoids searching unnecessary subspaces. In these cases, the implementation must exceed the specification, because nonterminating computations in the pruned portion of the search space will be unencountered by the implementation, but would cause the specification to fail to terminate.

If we take the algorithm in Fig. 4 as a specification for the more efficient least-cost search algorithm of Fig. 6, then we can easily prove the least-cost search algorithm correct. First we recall that the algorithm in Fig. 5 is equivalent to the one in Fig. 4. Using properties 6, 1, 2 and 5 above, it is easy to show that the least-cost search algorithm meets its specification.
7 CONCLUSION

We have seen that the polymorphic abstract data type improving allows us to express various combinatorial algorithms in a manner that is simpler than most previous expressions, yet at the same time introduces parallelism into the problem. Furthermore, the type improving has an axiomatic specification from which we can derive several important properties, which in turn can be used to prove the correctness of programs using the type.

As an example, we presented a four-line function for a parallel least-cost search that is optimal on a single processor and can make good use of any number of processors. A correctness proof of the function was outlined and can be easily completed by the reader.

References


