J. Functional Programming 3 (2): 153–169, April 1993

Improving intervals[1]

W. KEN JACKSON AND F. WARREN BURTON
School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6
(*e-mail*: jackson@cs.sfu.ca)

Abstract

We show how *improving values* (Burton, 1991) can be extended to handle both upper and lower bounds. The result is a new data type, called *improving intervals*. We give a simple implementation of improving intervals that uses a list of successively tighter bounds to represent a value. A program using improving intervals can be evaluated as a parallel program using speculative evaluation. The utility of improving intervals is demonstrated through two programs: parallel alpha-beta and parallel branch-and-bound

Capsule review

Lazy evaluation permits many search problems to be nicely split into one phase that builds the entire tree of possibilities and a second one that prunes this tree to find the goal. An example of this is alpha-beta pruning in game theory.

These pruning searches often use a (tightening) upper and lower bound (alpha and beta) to determine which nodes should be visited. This paper describes how this particular concept can be encapsulated into an abstract data type, called improving intervals. Using this data type it is no longer necessary to actually refer to the upper and lower bounds during the search; all the pruning happens internally in the abstract data type, and again relies heavily on lazy evaluation.

The main advantage of using the improving intervals is that the implementation of them can actually be made parallel without changing the semantics of the data type. This can speed up searches substantially. The paper describes how to implement the data type both in a parallel and non-parallel way. It also shows some properties that the implementation has to fulfil to be correct, and that their implementation has them.

1 Introduction

Search algorithms often use bounds to prune parts of the search space that cannot lead to an optimal solution. For example, a branch-and-bound algorithm for a minimization problem might prune a subspace whose lower bound exceeds the cost of the best solution found so far. The alpha-beta search algorithm uses upper and lower bounds to prune parts of a game tree that cannot contain the best move. Such algorithms can easily be implemented in a functional style by introducing new

[1] This work was supported by the Natural Science and Engineering Research Council of Canada.

arguments to keep track of the bounds (Bird and Hughes, 1987; Bird and Wadler, 1988).

An alternative in a lazy functional language is to represent a value by a lazy sequence of bounds and to modify the functions used in the program to handle these sequences of bounds (Hughes, 1989; Burton, 1991). Burton's approach encapsulates functions on sequences of lower bounds in an abstract data type called *improving values*. A program using improving values can be evaluated as a parallel program in which improving values have non-deterministic behaviour (that allows them to adapt their behaviour to the number of processors available) but deterministic semantics. In this paper, we describe a simple extension to improving values, called *improving intervals*, that encapsulates operations on both lower and upper bounds. A program using improving intervals can also be evaluated as a parallel program and we consider parallel programs throughout the paper. However, improving intervals have deterministic semantics and can also be used to simplify sequential programs.

Extending improving values with upper bounds complicates the semantics. Rather than giving complex exact semantics for improving intervals, we give an approximate semantics that constrains the operations on improving intervals but does not specify them exactly. Although these semantics are not exact, they are sufficient for proving programs correct.

We begin by reviewing the improving values abstract data type. In section 3 we present our extension to this data type and provide an implementation for it. Section 4 demonstrates the utility of our extensions through examples: a branch-and-bound program and an alpha-beta program. Related work and some problems are described in section 5. Throughout the paper, Miranda[2] (Turner, 1986) is used for our programming notation.

2 Improving values

An improving value represents a value as a monotonically increasing sequence of lower bounds. For example, the sequence [*3*, *5*, *10*] represents the value *10* but first gives the lower bounds 3 and 5 on the value. The sequence may be implemented as a lazy list, so if one of the lower bounds in the list gives sufficient information then not all elements of the list need to be computed.

The operations on improving values are encapsulated in the *improving* abstract data type whose signature is

```
abstype improving *
with
    make     :: * → improving *
    break    :: improving * → *
    spec_max :: improving * → improving * → improving *
    minimum  :: improving * → improving * → improving *.
```

The functions *make* and *break* are type transfer functions. The function *spec_max* evaluates its first argument to produce lower bounds on the result before evaluating

[2] Miranda is a trademark of Research Software Ltd.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

its second argument. That is *spec_max* (*make 5*) $\perp$ reduces to the partial list $5:\perp$ and represents a value that is at least 5. So, *spec_max* is strict in its first argument but is not strict in its second argument. On the other hand, *minimum* keeps evaluating the argument with the least lower bound until one of the arguments is completely evaluated. *minimum* is strict in both arguments but does not have to fully evaluate each argument. For example

$$minimum\,(make\,3)\,(spec\_max\,(make\,5)\,\perp) \quad = \quad (make\,3)$$

because the lower bound of 5 from the second argument is sufficient to determine that the minimum is 3.

The name *spec_max* comes from the idea that its second argument may be evaluated as a speculative computation. A speculative computation is a computation whose result may or may not be required later. A computation whose result is certainly required is called a 'mandatory computation'. On a parallel machine, speculative computations may be evaluated in parallel with mandatory computations if enough processors are available. If the result of a speculative computation is required, some time is saved by performing the speculative computation in parallel; if the result is not required then a processor is wasted doing work that is not required. We use the annotations *spec* and *priority* (Burton, 1985) to initiate and manage speculative computations. Semantically, *spec* is just the identity function, that is

$$spec\,f\,x \quad = \quad f\,x$$

and therefore the use of *spec* in a program cannot affect the semantics of the program, only its behaviour. Operationally, when an expression $spec\,f\,x$ is evaluated, the evaluation of x is started as a speculative computation before the mandatory evaluation of the application $f\,x$. To make good use of available resources, priorities should be set on speculative computations. The *priority* annotation is semantically defined as

$$\begin{aligned} priority\,p\,e \quad &= \quad e, \quad \textbf{if } p \neq \perp \\ &= \quad \perp, \quad \textbf{otherwise} \end{aligned}$$

where p is a numeric value and e is any expression. Operationally, when *priority* $p\,e$ is evaluated as a speculative computation it has the effect of setting the priority for the evaluation of e to p; when it is evaluated as a mandatory computation the *priority* annotation has no effect.

The second argument of *spec_max* may be evaluated as a speculative computation because its result is required only if the first argument fails to give a sufficient bound. If sufficient processors are available then the evaluation of the second argument may be done in parallel with evaluation of the first argument. Thus a program written using improving values can be evaluated as a parallel program.

The speculative aspect of *spec_max* means that operations on improving values have non-deterministic behaviour because the precise behaviour depends on the number of processors available. However, improving values have a simple deterministic semantics since the final result is the same regardless of the behaviour. Therefore, a program written using improving values is a parallel program that can adapt its behaviour to the number of processors while correctness can be established using the simple deterministic semantics.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

A limitation with the improving abstract data type is that it deals exclusively with lower bounds. It is, therefore, awkward to use improving values for algorithms like alpha-beta that use both upper and lower bounds. Many branch-and-bound algorithms also use both upper and lower bounds. This paper shows that the improving abstract data type can be extended to handle both upper and lower bounds. Instead of representing a value as a sequence of lower bounds, we use a sequence of upper and lower bounds. The details of this approach are given in the next section.

3 Improving intervals

This section describes a new abstract data type, called *improving intervals*, capable of dealing with both upper and lower bounds. An improving interval represents a value as a sequence of upper and lower bounds that bound an interval containing the value. As with improving values, the sequence is implemented as a lazy list so, for example, the list

$$[LB\,3, UB\,10, LB\,5, EX\,7]$$

represents the value 7. The list can also be considered as representing a sequence of successively smaller intervals that contain the final value – this is the reason for the name *improving intervals*. We consider an improving interval to denote the limit of this sequence of successively smaller intervals.

Improving intervals extend the improving abstract data type by adding two new functions: *spec_min* and *maximum*. The complete signature of improving intervals is shown in Fig. 1. The *spec_min* function is used to introduce upper bounds. The

abstype *improving_i* * **with**

make :: $* \rightarrow$ *improving_i* *

break :: *improving_i* * $\rightarrow$ *

minimum :: *improving_i* * $\rightarrow$ *improving_i* * $\rightarrow$ *improving_i* *

maximum :: *improving_i* * $\rightarrow$ *improving_i* * $\rightarrow$ *improving_i* *

spec_min :: *improving_i* * $\rightarrow$ *improving_i* * $\rightarrow$ *improving_i* *

spec_max :: *improving_i* * $\rightarrow$ *improving_i* * $\rightarrow$ *improving_i* *

Fig. 1. Signature of improving intervals

expressions *spec_min* (*make 5*) $\perp$, for example, represents a value that is at most 5. The *maximum* function combines bounds in a manner similar to *minimum*. In additions, *spec_max* and *minimum* are modified to handle both upper and lower bounds.

An improving interval may be represented by a lazy list of bounds that improve monotonically. A bound is a value tagged with *LB*, *UB* or *EX* to indicate if the value is a lower bound, an upper bound, or an exact value so the implementation types for improving intervals are

$$\begin{aligned} \textit{improving\_i}* \quad &== \quad [\textit{bnd}*] \\ \textit{bnd}* \quad &::= \quad EX* \mid LB* \mid UB*. \end{aligned}$$

Successive bounds in the list must be tighter than previous bounds to reflect the fact

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

that bounds improve monotonically. Since no bound is tighter than an exact value, there can be at most, one exact value in the list and any list with an exact value will be finite with the exact value as its last element. Conversely, every partial or infinite list cannot contain an exact value. Also note that the empty list is not a valid representation but that some partial lists, such as $LB\,3{:}\perp$, are valid.

The implementation of *make* and *break* are particularly simple

$$\begin{aligned} make\,a &= force\,[EX\,a] \\ break\,x &= a\ \textbf{where}\,(EX\,a) = last\,x \end{aligned}$$

make produces a singleton list where the element is tagged with *EX*. The function *force* ensures that *a* is fully defined (the reasons for using *force* are described later). The function *break* returns the value represented by the last bound in the list which must be the exact value if the list is finite. If the list is partial or infinite, then *break* will return $\perp$.

For the other functions, we describe only the implementation of *spec_max* and *maximum* since *spec_min* and *minimum* are the respective duals. To get the dual, change each *LB* to *UB* and *vice versa*; change *min2* to *max2* and *vice versa*; and turn around any use of $<$, $>$ or $\geqslant$.

Consider *spec_max*: when the evaluation of the first argument produces a lower bound, this lower bound is also a lower bound on the result of the *spec_max* application. Hence, a lower bound from the first argument is reproduced as output. If this lower bound is sufficient then further evaluation of the first argument or the second argument is not required. Otherwise the first argument and possibly the second argument are further evaluated to improve the bound as necessary. The implementation for *spec_max*, using some auxiliary functions, is shown below

$$\begin{aligned} spec\_max\,x\,y &= spec\,(sx\,x)\,y \\ \\ sx\,[EX\,a]\,y &= LB\,a{:}mxfilter\,a\,y \\ sx\,(LB\,a{:}x)\,y &= LB\,a{:}\,sx\,x\,y \\ sx\,(UB\,a{:}x)\,y &= sx\,x\,y \\ mxfilter\,a\,[EX\,b] &= [EX\,(max2\,a\,b)] \\ mxfilter\,a\,(LB\,b{:}y) &= mxfilter\,a\,y, \quad \textbf{if}\,a \geqslant b \\ &= LB\,b{:}y, \quad \textbf{if}\,a < b \\ mxfilter\,a\,(UB\,b{:}y) &= [EX\,a], \quad \textbf{if}\,a \geqslant b \\ &= UB\,b{:}mxfilter\,a\,y, \quad \textbf{if}\,a < b. \end{aligned}$$

The definition uses the annotation *spec* to initiate the speculative evaluation of the second argument. The auxiliary function *sx* does the real work by reproducing lower bounds from its first argument as output until an exact value in the first argument is found. Note that upper bounds are not reproduced since an upper bound from the first argument alone does not give an upper bound on the result. Once an exact value is found, *mxfilter* starts evaluating the second argument and produces tighter bounds from the second argument as output. The function *max2* is a Miranda builtin function that returns the maximum of two arguments.

The *maximum* function combines bounds from its arguments by reproducing as

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

output a bound from one of the arguments. It behaves somewhat like a merge but evaluation terminates once an exact value can be determined. The following code defines *maximum* by case analysis

$$
\begin{aligned}
&\mathit{maximum}\, x\, y \;=\; \mathit{mx}\, x\, y\\
&\mathit{mx}\,[\mathit{EX}\, a]\,[\mathit{EX}\, b] &&= [\mathit{EX}\,(\mathit{max2}\, a\, b)]\\
&\mathit{mx}\,[\mathit{EX}\, a]\,(\mathit{LB}\, b{:}y) &&= \mathit{LB}\, a{:}\mathit{mxfilter}\, a\, y, \quad \textbf{if}\, a \geqslant b\\
&&&= \mathit{LB}\, b{:}y, \quad \textbf{if}\, a < b\\
&\mathit{mx}\,[\mathit{EX}\, a]\,(\mathit{UB}\, b{:}y) &&= [\mathit{EX}\, a], \quad \textbf{if}\, a \geqslant b\\
&&&= \mathit{LB}\, a{:}\mathit{mxfilter}\, a\,(\mathit{UB}\, b{:}y), \quad \textbf{if}\, a < b\\
&\mathit{mx}\,(\mathit{LB}\, a{:}x)\,[\mathit{EX}\, b] &&= \mathit{mx}\,[\mathit{EX}\, b]\,(\mathit{LB}\, a{:}x)\\
&\mathit{mx}\,(\mathit{LB}\, a{:}x)\,(\mathit{LB}\, b{:}y) &&= \mathit{LB}\,(\mathit{max2}\, a\, b){:}\mathit{mx}\, x\, y\\
&\mathit{mx}\,(\mathit{LB}\, a{:}x)\,(\mathit{UB}\, b{:}y) &&= \mathit{LB}\, a{:}\mathit{mx}\, x\,(\mathit{UB}\, b{:}y), \quad \textbf{if}\, a \leqslant b\\
&&&= \mathit{LB}\, a{:}x, \quad \textbf{if}\, a > b\\
&\mathit{mx}\,(\mathit{UB}\, a{:}x)\,[\mathit{EX}\, b] &&= \mathit{mx}\,[\mathit{EX}\, b]\,(\mathit{UB}\, a{:}x)\\
&\mathit{mx}\,(\mathit{UB}\, a{:}x)\,(\mathit{LB}\, b{:}y) &&= \mathit{mx}\,(\mathit{LB}\, b{:}y)\,(\mathit{UB}\, a{:}x)\\
&\mathit{mx}\,(\mathit{UB}\, a{:}x)\,(\mathit{UB}\, b{:}y) &&= \mathit{UB}\, b{:}\mathit{mx}\,(\mathit{UB}\, a{:}x)\, y, \quad \textbf{if}\, a < b\\
&&&= \mathit{UB}\, a{:}\mathit{mx}\, x\, y, \quad \textbf{if}\, a = b\\
&&&= \mathit{UB}\, a{:}\mathit{mx}\, x\,(\mathit{UB}\, b{:}y), \quad \textbf{if}\, a > b.
\end{aligned}
$$

The auxiliary functions *mx* does the real work by merging bounds from its arguments into a single list of bounds. Notice how the equation

$$\mathit{mx}\,[\mathit{EX}\, a]\,(\mathit{UB}\, b{:}y) \;=\; [\mathit{EX}\, a], \quad \textbf{if}\, a \geqslant b$$

can produce an exact value without fully evaluating its second argument. This is how the pruning of unneeded computation takes place. If both arguments to *mx* produce an upper bound then *mx* reproduces only the larger of these bounds and demands further evaluation of the argument that produced the larger bound. This behaviour corresponds to expanding the node with the greatest upper bound in algorithms like best-first search. Figure 2 shows a sequential reduction of an expression.

3.1 Semantics of improving intervals

We have been unable to formulate a set of axioms for improving intervals that is clearer or simpler than the implementation. Instead, we give some properties that bound the behaviour of improving intervals, but do not specify them exactly. These properties are simple and are useful in proving the correctness of programs using improving intervals.

Recall that an improving interval denotes the limit of a sequence of successively smaller intervals. That is, the list $(\mathit{LB}\,3{:}\mathit{UB}\,10{:}\mathit{UB}\,7{:}\mathit{LB}\,5{:}\perp)$ denotes the interval $[5, 7]$. Let V be the base domain of values with a total ordering $\leqslant$. We assume that there are two elements $-\infty$ and ∞ that are the minimal and maximal elements of V with respect to $\leqslant$. The domain of intervals is the set

$$\{[a, b] \mid a, b \in V \quad \text{and} \quad a \leqslant b\}$$

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

$$
\begin{aligned}
& mx\,(sm\,(make\ 10)\,(make\ 7))\,(sm\,(make\ 5)\ \bot) \\
\Rightarrow\ & mx\,(sm\,[EX\ 10]\,(make\ 7))\,(sm\,(make\ 5)\ \bot) \\
\Rightarrow\ & mx\,(UB\ 10 : mnfilter\ 10\,(make\ 7))\,(sm\,(make\ 5)\ \bot) \\
\Rightarrow\ & mx\,(UB\ 10 : mnfilter\ 10\,(make\ 7))\,(UB\ 5 : mnfilter\ 5\ \bot) \\
\Rightarrow\ & UB\ 10 : mx\,(mnfilter\ 10\,(make\ 7))\,(UB\ 5 : mnfilter\ 5\ \bot) \\
\Rightarrow\ & UB\ 10 : mx\,[EX\ 7]\,(UB\ 5 : mnfilter\ 5\ \bot) \\
\Rightarrow\ & UB\ 10 : [EX\ 7] \\
\Rightarrow\ & [UB\ 10, EX\ 7]
\end{aligned}
$$

Fig. 2. Sequential reduction of an expression

with $\bot_{ii} = [-\infty, \infty]$ and where the ordering $\sqsubseteq_{ii}$ is defined by

$$[a, b] \sqsubseteq_{ii} [c, d] \quad \text{iff} \quad a \leqslant c \quad \text{and} \quad d \leqslant b.$$

We use variables a, b, c, d for values while the variables x, y, z are used for improving intervals.

Conceptually, an interval bounds an (as yet) unknown value. Given two intervals, i_1 and i_2 that bound the values a and b, the maximum of i_1 and i_2 is an interval that bounds the value $max2\,a\,b$. It is easy to define such a function:

$$
\begin{aligned}
max_{ii}\ \bot_{ii}\ \bot_{ii} &= \bot_{ii} \\
max_{ii}\ \bot_{ii}\ [a, b] &= [a, \infty] \\
max_{ii}\ [a, b]\ \bot_{ii} &= [a, \infty] \\
max_{ii}\ [a, b]\ [c, d] &= [max2\,a\,c, max2\,b\,d].
\end{aligned}
$$

A minimum function could be defined in a similar manner. However, this function is not directly implementable because the second two equations imply that max_{ii} must be non-strict yet bottom avoiding in both its arguments. This means that both arguments must be evaluated in parallel. Such functions are called *non-sequential* and cannot be implemented directly in standard functional languages (such as Miranda). On the other hand, the functions *spec_max* and *maximum* can be implemented directly but only approximate max_{ii}, that is

$$
\begin{aligned}
spec\_max &\sqsubseteq_{ii} max_{ii} \\
maximum &\sqsubseteq_{ii} max_{ii}.
\end{aligned}
$$

The *make and break* functions are just type transfer functions and should satisfy the following:

$$make\ \bot = \bot \tag{1}$$

$$break\ \bot = \bot \tag{2}$$

$$(break \,.\, make)\,a = a \tag{3}$$

$$(make \cdot break)\,x \sqsubseteq_{ii} x. \tag{4}$$

The use of $\sqsubseteq_{ii}$, rather than $=$, is necessary in (4) because improving intervals are a richer domain than values; for example

$$(make \,.\, break)\,(spec\_max\ a\ \bot) = \bot \sqsubseteq_{ii} spec\_max\ a\ \bot.$$

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Properties (1) and (2) are not really necessary since they follow from (3) and (4) assuming that *make* and *break* are monotonic.

We assume that specifications for programs using improving intervals will be written using *min2* and *max2*. With this assumption, program correctness is easy to show as long as the operations on improving intervals obey the following properties:

$$min2\,a\,b = break\,(minimum\,(make\,a)\,(make\,b)) \quad (5)$$

$$max2\,a\,b = break\,(maximum\,(make\,a)\,(make\,b)) \quad (6)$$

$$min2\,a\,b = break\,(spec\_min\,(make\,a)\,(make\,b)) \quad (7)$$

$$max2\,a\,b = break\,(spec\_max\,(make\,a)\,(make\,b)). \quad (8)$$

It is easy to see that the above properties do not precisely specify the implementation because they make no distinction between *spec_max* and *maximum* while the implementation does impose some differences.

The above do not hold for the implementation when the values might be partially defined. For example, the expression

$$max2\,(1:\bot)\,(min2\,(1:\bot)\,(0:\bot))$$

reduces to $(1:\bot)$ in Miranda. However, applying (6) and (7) gives the expression

$$break\,(maximum\,(make\,(1:\bot))\,(spec\_min\,(make\,(1:\bot))\,(make\,(0:\bot))))$$

which reduces to $\bot$ because the test $(1:\bot) = (1:\bot)$ is evaluated. To avoid this difficulty, we assume that values are fully defined and use *force* in the implementation of *make* to ensures this.

The following properties follow from monotonicity, property (4) and properties (5)–(8) by substituting break x for a and *break* y for b

$$min2\,(break\,x)\,(break\,y) \sqsubseteq break\,(minimum\,x\,y) \quad (9)$$

$$max2\,(break\,x)\,(break\,y) \sqsubseteq break\,(maximum\,x\,y) \quad (10)$$

$$min2\,(break\,x)\,(break\,y) \sqsubseteq break\,(spec\_min\,x\,y) \quad (11)$$

$$max2\,(break\,x)\,(break\,y) \sqsubseteq break\,(spec\_max\,x\,y). \quad (12)$$

These properties extend easily to lists of improving intervals. For example, for any non-empty list *xs*, it can be shown that

$$max\,(map\,break\,xs) \sqsubseteq break\,(foldl1\,spec\_max\,xs) \quad (13)$$

using (12) and structural induction on the list. We will use this property in the next section to prove the correctness of an alpha-beta program.

The bounds on the behaviour of *spec_max* can be made explicit by applying *make* to both sides of 12 to give

$$make\,(max2\,(break\,x)\,(break\,y)) \sqsubseteq_{ii} (spec\_max\,x\,y)$$

and previously, we had that

$$spec\_max\,x\,y \sqsubseteq_{ii} max_{ii}.$$

These inequalities serve as an inexact semantics for *spec_max*; a semantics that bounds *spec_max* but does not precisely define it.

The following theorem shows that our implementation satisfies the above properties:

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Theorem 3.1
The above implementation satisfies properties (1)–(8).

Proof
Recall that properties (1) and (2) follow from monotonicity, (3) and (4). We show the proofs only for the max versions, since the others can be done by taking the duals

(3) Consider $(break.make)\,a$
Case $a = \bot$. $break\ (make\ \bot) = \bot$
Case $a \neq \bot$. $break\ (make\ a) = break\ ([EX\ a]) = a$

(4) Consider $(make.break)\ x$
Case $x = \bot$. $make\ (break\ x) = \bot$
Case x is finite. Assume that $x = xs +\!\!+ [EX\ a]$

$$(make.break)\,x \quad = \quad make\,a \quad = \quad [EX\,a]$$

and $xs +\!\!+ [EX\,a] = [EX\,a]$ since they both denote the same value.
Case x is not finite. $make\ (break\ x) = make\ \bot = \bot \sqsubseteq_{ii} x$

(6) **Case** $a = \bot$ or $b = \bot$.
Then $max2\,a\,b = \bot$ and $maximum\ (make\,a)\,(make\,b) = \bot$
Case $a \neq \bot$ *and* $b \neq \bot$. Assume that $max2\,a\,b = a$

$$\begin{aligned} break\,(maximum\,(make\,a)\,(make\,b)) \quad &= \quad break\,([EX\,(max2\,a\,b)]) \\ &= \quad a \end{aligned}$$

(8) **Case** $a = \bot$. Then $max2\,a\,b = \bot = break\,(spec\_max\ \bot\ (make\,b))$
Case $b = \bot$. Then $max2\,a\,b = \bot = break\,(spec\_max\,(make\,a)\ \bot)$
Case $a \neq \bot$ and $b \neq \bot$. Assume that $max2\,a\,b = a$

$$\begin{aligned} &break\,(spec\_max\,(make\,a)\,(make\,b)) \\ &\quad = \quad break\,(spec\,(sx\,(make\,a))\,(make\,b)) \\ &\quad = \quad break\,(sx\,(make\,a)\,(make\,b)) \\ &\quad = \quad break\,(UB\,a : mxfilter\,a\,(make\,b)) \\ &\quad = \quad break\,(UB\,a : [EX\,a]) \\ &\quad = \quad a \end{aligned}$$

□

4 Examples using improving intervals

In this section, we consider how improving intervals can be used in search programs. First, we consider an alpha-beta program that uses just *spec_min* and *spec_max*, then we consider a branch-and-bound program.

4.1. Alpha-Beta search

The alpha-beta algorithm is used to find the minimax value of a game tree in two player games. We start by defining the minimax value of a game tree then give an alpha-beta program, using improving intervals that meets this specification.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Nodes in a games tree represent positions and alternate levels are specified as max-nodes or min-nodes corresponding to a player's move or the opponent's move. We use a type *gtree* * for game trees, defined by

$$\begin{aligned} gtree* &== mxnode* \\ mxnode* &::= MXnode*\,[mnnode*] \\ mnnode* &::= MNnode*\,[mxnode*]. \end{aligned}$$

A leaf node will have an empty list of children. The type *gtree* * is polymorphic, but we assume that it is instantiated with a number that indicates the value of the game position represented by that node (the value is usually determined by applying a static evaluation function to the game position). The *minimax* value of a game tree is the value of the best position in the game tree and is specified recursively as: the value of leaf nodes, the max of all children for *MXnodes*, or the min of children for *MNnodes*. The function *minmax*, defined below, computes the minimax value of a game tree

$$\begin{aligned} &minmax :: gtree* \to * \\ &minmax = minmax\_max \\ \\ &minmax\_max :: mxnode* \to * \\ &minmax\_max\,(MXnode\,a\,kids) \\ &\qquad = a, \quad \textbf{if}\, kids = [] \\ &\qquad = max\,(map\,minmax\_min\,kids), \quad \textbf{otherwise} \\ \\ &minmax\_min :: mnnode* \to * \\ &minmax\_min\,(MNnode\,a\,kids) \\ &\qquad = a, \quad \textbf{if}\, kids = [] \\ &\qquad = min\,(map\,minmax\_max\,kids), \quad \textbf{otherwise}. \end{aligned}$$

A simple alpha-beta program is constructed from the definition of *minmax* by replacing *max* with *foldl1 spec_max* and *min* with *foldl1 spec_min*. The resulting program is

$$\begin{aligned} &alphabeta :: gtree* \to * \\ &alphabeta\,t = break\,(ab\_max\,t) \\ \\ &ab\_max :: mxnode* \to improving* \\ &ab\_max\,(MXnode\,a\,kids) \\ &\qquad = make\,a, \quad \textbf{if}\, kids = [] \\ &\qquad = foldl1\,spec\_max \quad (map\,ab\_min\,kids), \quad \textbf{otherwise} \\ \\ &ab\_min :: mnnode* \to improving* \\ &ab\_min\,(MNnode\,a\,kids) \\ &\qquad = make\,a, \quad \textbf{if}\, kids = [] \\ &\qquad = foldl1\,spec\_min\,(map\,ab\_max\,kids), \quad \textbf{otherwise}. \end{aligned}$$

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

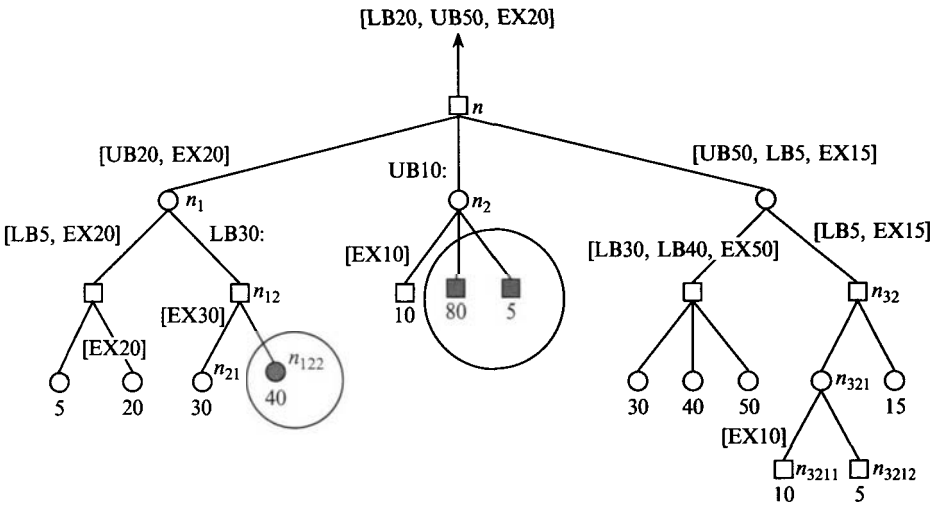

Fig. 3. Sequential operation of the alpha-beta algorithm

At a leaf node the value of the node is returned as an improving interval by using *make* and at the root the value of the game tree is converted from an improving interval to a value by using *break*. The handling of bounds and pruning is encapsulated in the operations on improving intervals.

Figure 3 shows a game tree and the improving intervals that are computed at each node. *MXnodes* are drawn with a square box, while *MNnodes* are drawn with a circle. The circled portions of the game tree are pruned during the search. When executed sequentially, the program uses a depth-first search strategy because *spec_min* and *spec_max* fully evaluate their first argument before considering their second argument. Node n_{122} is pruned because the lower bound of 30 from node n_{121} is sufficient to determine that the value of node n_1 is exactly 20. Other nodes are pruned in a similar manner.

When more than a single processor is available, speculative computation could be used to explore other subtrees in parallel. For example, the subtree rooted at n_3 might be evaluated as speculative computation in parallel with the mandatory evaluation of n_1.

However, the program misses an opportunity for pruning the node labelled n_{3212}. The reasoning that supports the pruning of this node is that we have a lower bound of 20 at the root node; to get a better result at the root each of the nodes: n_3, n_{32} and n_{321} would have to be greater than 20; but we know n_{321} must be less than 10 without examining n_{3212}; therefore n_{3212} can be pruned. This is known as a *deep cutoff*. Deep cutoffs are difficult to achieve in a parallel alpha-beta algorithm because deep cutoffs rely on bounds found by evaluating the left siblings of a node. In a parallel setting, if we wait for these bounds then we reduce the amount of parallelism; if we do not wait then we miss the opportunity for a cutoff. Akl *et al.* (1982) discuss this point further.

To show the correctness of this program, we prove that *alphabeta* meets the specification, *minmax*, by using the properties defined in section 3.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Theorem 4.1

For any game tree t, minmax t $\sqsubseteq$ *alphabeta t.*

Proof

Let t be any game tree. If t is an infinite tree then $minmax\ t = \bot$ and the result trivially holds. Otherwise, t is finite and we use induction on the height of t to show that $minmax\_max \sqsubseteq break\,.\,ab\_max$. A dual proof shows that $minmax\_min \sqsubseteq break\,.\,ab\_min$.

Case $t = MXnode\ a\ []$.

$$\begin{aligned} minmax\_max\ (MXnode\ a\ []) &= a && (minmax\_max\,.\,2)\\ &= (break\,.\,make)\ a && (3)\\ &= break\ (ab\_max\ (MXnode\ a\ [])) && (ab\_max\,.\,1) \end{aligned}$$

Case $t = MXnode\ a\ kids, kids \neq []$.

$$\begin{aligned} &minimax\_max\ (MXnode\ a\ kids)\\ &= max\ (map\ minimax\_min\ kids) && (minmax\_max\,.\,2)\\ &\sqsubseteq max\ (map\ (break\,.\,ab\_min)\ kids) && \text{(Induction Hypothesis)}\\ &= max\ (map\ break\ (map\ ab\_min\ kids)) && (map)\\ &\sqsubseteq break\ (foldl1\ spec\_max\ (map\ ab\_min\ kids)) && (13)\\ &= break\ (ab\_max\ (MXnode\ a\ kids)) && (ab\_max\,.\,2) \end{aligned}$$

Therefore

$$minmax\ t \;=\; minmax\_max\ t \;\sqsubseteq\; break\ (ab\_max\ t) \;=\; alphabeta\ t \qquad \square$$

This shows that *alphabeta* is correct with respect to *minmax*. However, it does not show that *alphabeta* does all the pruning that the standard alpha-beta algorithm does. In fact, we know from the previous discussion that *alphabeta* misses the deep cutoffs. The pruning behaviour of improving intervals can only be understood by carefully examining the operational behaviour of the specific implementation. Understanding pruning is difficult in the sequential case and is even more difficult in the parallel case because of speedup anomalies (Lai and Sahni, 1984). The issue of operational correctness is largely ignored in functional programs and remains an open problem. Neither Bird and Hughes (1987) nor Hughes (1989) discusses the operational correctness of their algorithms.

We likely need to control the speculative computations using priorities. A reasonable strategy in parallel alpha-beta search is to:

1. Give priority to searching nodes at greater depth in the tree because new bounds are found only at nodes at the bottom of the tree.
2. Give priority to left children because game trees are usually ordered such that better moves are to the left.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

The following function assigns a priority to a node n in a manner consistent with the above rules:

$$priority\,(n) \quad = \quad priority\,(parent\,(n)) + 1 + r(n)\,b^{d(n)}$$

where $d(n)$ = depth of the tree rooted at n

b = maximum branching factor of the tree

$r(n)$ = number of right siblings of n.

We assign priorities to nodes by pairing each node with its priority. Then to make *alphabeta* use the priorities, we define *alphabeta* in terms of two new function *ab_max'* and *ab_min'* where *ab_max'* is defined as

$$ab\_max'\,(MXnode\,(p, a)\,kids) \quad = \quad priority\,p\,(ab\_max\,(MXnode\,a\,kids)).$$

If a node is evaluated with a speculative computation then the priority will be set to p, otherwise it is evaluated with a mandatory computation and the *priority* annotation has no effect.

4.2 Branch-and-bound with upper and lower bounds

Branch–and–bound is often used for combinatorial optimization problems such as the 0/1 knapsack problem. Burton (1991) gives a least cost branch-and-bound program that uses improving values, but this program uses only lower bounds and for many problems, including the 0/1 knapsack problem, there is a simple upper bound as well as a lower bound. We can use improving intervals to write a simple branch-and-bound program that uses both upper and lower bounds.

A branch-and-bound algorithm searches a tree for solutions and uses bounds to prune parts of the tree that cannot lead to a solution. We use *spec_max* to introduce a lower bound, *spec_min* to introduce an upper bound, and *minimum/maximum* to combine the bounds. We assume the existence of functions *upper_bound* and *lower_bound* that return the upper and lower bound of a node. The function *lcsearch'* implements a least-cost branch-and-bound algorithm for a minimization problem

lcsearch' root

= *make* (*cost root*, *root*), **if** *is_leaf root*

= *spec_min ubound* (*spec_max lbound subtree_search*), **otherwise**

where

lbound = *make* (*lower_bound root*, *nil_node*)

ubound = *make* (*upper_bound root*, *nil_node*)

subtree_search = (*foldr1 minimum . map lcsearch' . children*) *root*.

The least-cost behaviour comes about because *minimum* demands further evaluation of the argument that produced the smallest lower bound. The only difference from

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Burton's program is the addition of *spec_minubound*(...) to introduce the upper bound. In a parallel setting, speculative computation could be used to search some of the subtrees in parallel.

5 Discussion

In this section, we consider some related work on search programs and some of the problems associated with improving intervals.

5.1 Related programs

It is an interesting exercise to derive an alpha-beta program that uses extra arguments to maintain the bounds, from the specification of minimaxing (Bird and Hughes, 1987; Bird and Wadler, 1988). However, the resulting program is not surprising. For example, the program (Bird and Wadler, 1988)

$$
\begin{array}{lcl}
gtree & ::= & Node\ num\ [gtree] \\
bmx\ a\ b\ (Node\ x\ ts) & = & max2\ a\ (min2\ x\ b), \quad \textbf{if}\ ts = [] \\
 & = & cmx\ a\ b\ ts, \quad \textbf{otherwise} \\
 & & \\
cmx\ a\ b\ [] & = & a \\
cmx\ a\ b\ (t : ts) & = & a', \quad \textbf{if}\ a' = b \\
cmx\ a\ b\ (t : ts) & = & cms\ a'\ b\ ts, \quad \textbf{otherwise} \\
 & & \textbf{where}\ a' = -bmx\ (-b)\ (-a)\ t
\end{array}
$$

uses the arguments a and b to record the current bounds. The function *cmx* recursively searches a list of sibling nodes, and at each step in the recursion a is the bound found by searching siblings to the left of the current node. Note that the test $a' = b$ in *cmx* sets up a data dependency between the result of the *cmx* application and the value of a'. This data dependency forces the search to proceed in a sequential left-to-right manner. Therefore, this program has no potential for searching the subtrees in parallel. The advantage of no parallelism is that, unlike our program, deep cutoffs are performed correctly.

We might consider parallelising the above program by using a *parallel_if* that starts the speculative computation of the then and else parts in parallel with the computation of the test. In that case, the tree is expanded in parallel, but none of the nodes can be evaluated because of the data dependencies. It follows that no pruning would occur in the subtrees that are expanded speculatively. In contrast, with improving intervals subtrees are evaluated in parallel and some pruning can occur during the speculative evaluation of a subtree.

Hughes (1989) defines two functions, *max'* and *min'*, and uses them in an alpha-beta program. The *max'* function returns an increasing list of lower bounds, and the *min'* function returns a decreasing list of upper bounds. Improving intervals generalize this idea by allowing upper and lower bounds in the same list. This seems important for algorithms like branch-and-bound, where a node may have both a lower and upper bound. Surprisingly, combining upper and lower bounds also makes

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

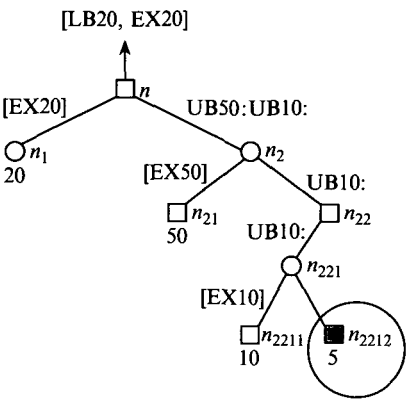

Fig. 4. A missed pruning opportunity in Hughes' approach

a difference in the alpha-beta program. Figure 4 shows an example, where our approach prunes more of the search space because we are able to return the upper bound *UB 10* from the max node n_{22} which allows node n_{2212} to be pruned. Hughes' program cannot do this pruning because there is no way to represent an upper bound on a max node.

5.2 Opportunities for improvement

There are some problems associated with using improving intervals especially related to efficiency. One occurs as a result of the list representation of improving intervals. Recall that in our implementation of improving intervals, a value is represented by a list of successively better and better bounds. However, at any point in time, it is really only the tightest bounds currently in the list that are important. For example, the lists $LB3{:}LB5{:}UB10{:}UB8{:}\perp$ and $LB5{:}UB8{:}\perp$ have the same information content. Hence, the list representation consumes more space than is necessary, and time is wasted examining the out-of-date values in the list. A better representation, in a language that allows update-in-place, would be a pair consisting of the tightest lower bound and the tightest upper bound found so far. This pair would be updated-in-place when better bounds become known.

Parallel search algorithms are often difficult to analyse and empirically evaluate because of speed-up anomalies that may occur (Lai and Sahni, 1984). Often, anomalies occur because a parallel algorithm searches a different space than a sequential algorithm. Search overhead is incurred when the parallel algorithm searches a larger space than the sequential one, and in particularly bad cases the parallel algorithm may run more slowly than the sequential algorithm. There is a lot of work on developing (imperative) parallel search algorithms that try to avoid as much search overhead as possible (Akl *et al.* 1982; Li and Wah, 1990; Powley *et al.* 1989). To do so, these algorithm carefully control the order in which nodes are searched. Efficient parallel search algorithms that use improving intervals must do the same. It is not clear whether priorities alone allow enough control to express such algorithms.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

A serious problem arises because *spec_max* and *spec_min* are not symmetric in their arguments. For example

$$spec\_max\,[LB\,5, EX\,10]\,(UB\,3:\bot) \quad = \quad [LB\,5, EX\,10]$$

but

$$spec\_max\,(UB\,3:\bot)\,[LB\,5, EX\,10] \quad = \quad \bot.$$

In terms of parallel search, a bound found by evaluating the left subtree can be used for pruning, but a bound found by evaluating the right subtree is ignored until evaluation of the left subtree completes. We would often like some non-sequential behaviour so that both arguments to *spec_max* would evaluate in parallel and as soon as a bound, from either argument, is computed it may be used for prune parts of the tree. The typical example of a non-sequential function is the parallel-or function that evaluates its two arguments in parallel and returns true as soon as one of the arguments returns true, even if the other argument is $\bot$. We would like a parallel-max (call it *pmax*) with similar behaviour. If

$$pmax\,(make\,5)\,\bot \quad = \quad LB\,5:\bot$$

and

$$pmax\,\bot\,(make\,5) \quad = \quad LB\,5:\bot$$

then we can use the $LB\,5$ regardless of which argument it came from. However, using a non-sequential function like *pmax* requires parallel evaluation so that a search tree of height n would create $O(2^n)$ processes when evaluated. A compromise is to use a function that behaves like *pmax* when enough processors are available and otherwise behaves like *spec_max*. Such a function is non-deterministic because its results can be different on different runs, but it is only partially non-deterministic because it is at least as well defined as *spec_max*. We are currently working on the further development of these partially deterministic functions (Burton and Jackson, 1990).

6 Conclusions

This paper has shown how to extend Burton's improving values to handle both upper and lower bounds. We introduced two new functions, *spec_min* and *maximum*, that are used to introduce and handle upper bounds. A simple implementation of improving intervals uses lists of successively tighter bounds to represent a value. Improving intervals are useful for writing programs, like alpha-beta search or branch-and-bound, that make use of both upper and lower bounds.

References

Akl, S., Barnard, D. and Doran, R. 1982. Design, analysis and implementation of a parallel tree search algorithm. *IEEE Trans. Pattern Analysis & Machine Intelligence*, 4(2): 192–203.

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice Hall.

Bird, R. S. and Hughes, J. 1987. The alpha-beta algorithm: An exercise in program transformation. *Inf. Process. Lett.*, 24:53–57.

Burton, F. W. 1985. Speculative computation, parallelism, and functional programming. *IEEE Trans. Comput.*, 34(12):1190–1193.

https://doi.org/10.1017/S0956796800000678 Published online by Cambridge University Press

Burton, F. W. 1991. Encapsulating nondeterminacy in an abstract data type with deterministic semantics. *J. Functional Programming*. 1(1):3–20.

Burton, F. W. and Jackson, W. K. 1990. Partially deterministic functions. *IV Higher Order Workshop*, Springer-Verlag Workshops in Computing, pp 1–10, Banff, Canada 10–14 September, Graham Birtwistle (Ed.).

Hughes, J. 1989. Why functional programming matters. *Comput. J.*, 32(2):98–107.

Lai, T.-H. and Sahni, S. 1984. Anomalies in parallel branch-and-bound algorithms. *Commun. ACM*, 27(6):594–602.

Li, G.-J. and Wah, B. W. 1990. Computational efficiency of parallel combinatorial or-tree searches. *IEEE Trans. Softw. Eng.*, 16(1):13–31.

Powley, C., Ferguson, C. and Korf, R. E. 1989. Parallel heuristic search: Two approaches. In Kumar, V., Gopalakrishnan, P. and Kanal, L. N., editors, *Parallel Algorithms for Machine Intelligence and Vision*, pp. 42–65. Springer-Verlag.

Turner, D. A. 1986. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166.