Implementing Declarative Parallel Bottom-Avoiding Choice[1]

André Rauber Du Bois   Robert Pointon   Hans-Wolfgang Loidl   Phil Trinder

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.

E-mail: {dubois,rpointon,hwloidl,trinder}@cee.hw.ac.uk

Abstract

Non-deterministic choice supports efficient parallel speculation, but unrestricted non-determinism destroys the referential transparency of purely-declarative languages by removing unfoldability and it bears the danger of wasting resources on unncessary computations. While numerous choice mechanisms have been proposed that preserve unfoldability, and some concurrent implementations exist, we believe that no compiled parallel implementation has previously been constructed. This paper presents the design, semantics, implementation and use of a family of bottom-avoiding choice operators for Glasgow parallel Haskell. The subtle semantic properties of our choice operations are described, including a careful classification using an existing framework, together with a discussion of operational semantics issues and the pragmatics of distributed memory implementation. The expressiveness of our choice operators is demonstrated by constructing a branch and bound search, a merge and a speculative conditional. Their effectiveness is demonstrated by comparing the parallel performance of the speculative search with naive and 'perfect' implementations. Their efficiency is assessed by measuring runtime overhead and heap consumption.

1. Introduction

The importance of non-deterministic bottom-avoiding choice has long been recognised [13] both for chosing alternative actions in reactive systems and for controlling parallel speculation. This paper focusses on the latter application, and a good example is a branch and bound search where parallel threads explore subspaces. Each thread is *speculative* because it may not find a solution, and hence its value may not be used. Such searches can be formulated as a non-deterministic choice between the values produced by the speculative threads.

The semantics of McCarthy's choice operator [13] are deceptively simple: `x amb y` returns either `x` or `y`, typically selecting the first to terminate. However, naively introducing non-deterministic choice into a declarative language is problematic because it destroys the unfolding property, stating that any function call may be replaced by the function body with the argument substituted for the formal parameters [20] [1]. For example in most languages, e.g. call-by-value languages or languages with sharing,

```
(\x-> x+x) (3 amb 5)
```

makes a single choice and has value 6 or 10, but `(3 amb 5) + (3 amb 5)` makes two choices and has value 6, 8 or 10.

Several means of introducing bottom-avoiding choice while preserving semantic properties like unfoldability have been proposed, e.g. [2, 9, 18, 22] but very few truly parallel implementations have been constructed. Implementations are required to explore performance implications and subleties of the parallel operational semantics. There are some concurrent choice implementations that interleave the execution of speculative threads on a single processor [5] but we are aware of just one other truly parallel choice implementation, and that is for an interpreted language [18].

We present a family of bottom-avoiding choice operators for Glasgow parallel Haskell (GPH) [21]. The operators are unique in combining the preservation of unfoldability with a compiled parallel implementation. The choice operators are adaptions of McCarthy's `amb` that preserve unfoldability using an approach that extend proposals by Hughes and O'Donnell [9] and Wadler [22]. That is, each choice injects its result into a monad with restricted operations. The `ND` monad has the added convenience of syntactic support in Haskell, and hence the type of `ambND :: a -> a -> ND a`

[1]This work is supported by EPSRC (grant GR/R88137) and the Austrian Academy of Sciences (APART 624).

[1]This is often stated as "losing referential transparency" [19]

Non-deterministic choice has very subtle semantics, and the numerous variants have been classified by semantic properties [20]. Section 3.1 carefully characterises the properties of our choice operators within this classification. We explore the implications of the non-standard evaluation order implied by introducing parallel speculation into a non-strict language, e.g. searching lazy infinite data structures The choice operations are implemented as Glasgow Distributed Haskell (GDH) [17] skeletons within GPH, and execute in a distributed memory environment. Section 3.2 outlines the operational semantics issues and requirements for an efficient implementation.

In Section 4 the expressiveness of our choice operators is demonstrated by constructing a branch and bound search, a merge and a speculative conditional. Their effectiveness is demonstrated by comparing the parallel performance of the speculative search with naive and 'perfect' implementations. Their efficiency is assessed by measuring runtime overhead and heap consumption.

2. Purely-Functional Parallel Search

This section considers parallel searching in a purely-functional language. The example problem is to find a path to an exit through a maze.

2.1. GPH

GPH(*Glasgow Parallel Haskell*) [21] is a modest extension of Haskell, using the parallel combinator `par` to specify parallel evaluation. The expression ``p `par` e`` (here we use Haskell's infix operator notation) has the same value as `e`. Its dynamic effect is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. The sequential composition operator `seq` is used to control sequencing. The expression ``e1 `seq` e2`` has the value of `e2`. Its dynamic behavior is to evaluate `e1` to *weak head normal form* (WHNF) before returning `e2`.

Higher-level coordination is provided using *evaluation strategies*, which are higher-order polymorphic functions that use the `par` and `seq` combinators to specify the degree of evaluation of expressions. For example:

```
using :: a -> Strategy a -> a
using x s = s x 'seq' x

parMap f xs = map f xs 'using' parList rnf
```

`parMap` applies the function `f` in all the elements of the list `xs` in parallel. This parallel map function is implemented using the `parList` and `rnf` strategies. The `parList` function evaluates the elements of a list in parallel to the degree specified by its argument, in this case, to normal form using the `rnf` strategy. `parList` and `rnf` have a straightforward implementation using `par` and `seq`. For a discussion on how to implement efficient parallel programs using GPH, the reader should refer to [11].

2.2. Simple Parallel Search

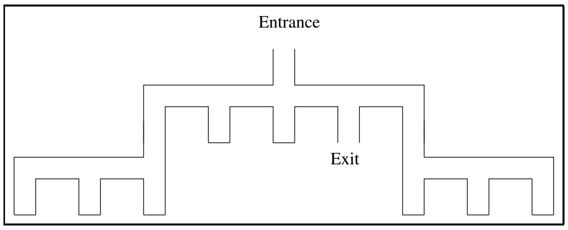

Figure 1. A maze as a tree

We represent a maze as a tree with exactly one path from the entrance (root) to the exit (one leaf) Figure 1. The following algorithm is a sequential solution to this search problem:

```
search ::(a->Bool) -> (a->Bool) -> (a->[a])
             -> [a] -> a -> [a]
search isExit cont succ solutions node
  |isExit node   = node:solutions
  |cont node =  control (map
     (search isExit cont succ
         (node:solutions)) (succ node))
  | otherwise     = []

control = firstSolution

firstSolution :: [[a]] -> [a]
firstSolution [] = []
firstSolution ([]:xs) = firstSolution xs
firstSolution (x:xs)  = x
```

The `search` is a simple depth first search of the tree. It takes 5 arguments: a function `isExit` that determines whether we have found a solution to the problem; a function `cont` that determines if it is possible to continue the search in that branch; a function `succ` that gives the children of the current node; a list that is the path to the current node;and the starting `node` for the search. The top-level `control` function directs the search, and for a sequential search `firstSolution` is used to specify a lazy sequential approach, stopping when the first solution is found.

A naive parallel version is obtained from the sequential algorithm by introducing a control function that evaluates the elements of the input list of `firstSolution` in parallel.

```
control xs = firstSolution
                (xs  'using' parList rnf)
```

In this simple parallel version parallelism is introduced in every recursive call to the `search` function. To avoid generating fine grained tasks near the leaves of the tree this kind of function is usually implemented with an extra threshold parameter, `th`, that specifies to what level parallelism should be introduced in the program. Once this parameter reaches 0 the sequential `control` function is used.

```
searchp::Int ->(a->Bool)->(a->Bool)->
                (a->[a])->[a]->a->[a]
searchp th isExit cont succ solutions node
 | isExit node       = node:solutions
 | cont node         = control th (map
    (searchp (th-1) isExit cont succ
        (node:solutions)) (succ node))
 | otherwise             = []

control th xs = if (th>0) then
  (firstSolution (xs  `using` parList rnf))
  else firstSolution xs
```

2.3. A 'Perfect' Fuel Version

Our parallel implementation of depth first search works well for balanced search problems. But in general many search spaces are unbalanced, and if one thread finds a solution we do not want the other threads to continue their searches, consuming resources. In the parallel version of the `search` function presented above tasks are created to solve the problem but we have to wait for all the threads to finish searching before printing the result. We would prefer to start working with the results produced by the child tasks as soon as they become available.

Assuming that we know the level of the tree in which the solution of the problem can be found, we could implement a search function with an extra parameter, like a threshold, that tells the function how deep it should search. When this value, the *fuel*, reaches zero, the thread stops searching and returns the values produced so far, and these are checked to see if any thread has produced a solution. This is a special case of general fuel algorithms, and is expressed in the following function.

```
searchf::Int->Int->(a->Bool)->(a->Bool)
            ->(a->[a])->[a]->a->[a]
searchf 0 th isExit cont succ sol node
  | isExit node  = node:sol
  | otherwise    = []
searchf fuel th isExit cont succ sol node =
      control th (map (searchf (fuel-1)
      (th-1) isExit cont succ(node:sol))
      (succ node))
```

When the fuel of the function is zero, it tests all the children produced so far, trying to find a solution. All other types of nodes are discarded.

This parallel fuel program is 'perfect', rather than realistic, because the programmer usually does not know in which level of the tree the solution is, but it serves to give a good estimate of the best possible search time.

2.4. Measurements

All measurements use up to 8 nodes of a Beowulf cluster, consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128Kb cache, 128MB of DRAM and 5.7GB of IDE disk. The workstations are connected with a 100Mb/s fast Ethernet switch.

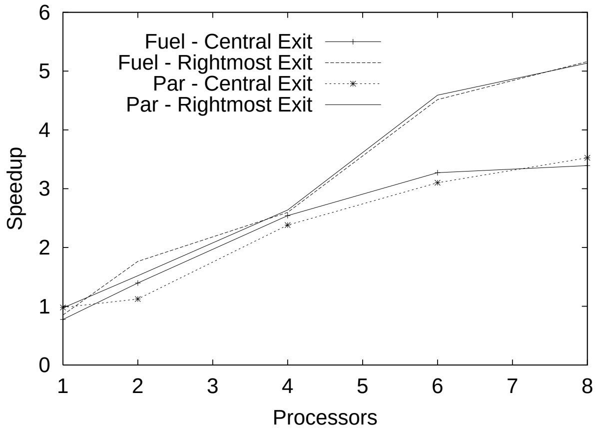

Figure 2. Maze Search Speedups

The mazes are trees with a branching factor of 7 for the root and 6 for all other levels. The depth of the root's subtrees is 10, but the subtree containing the exit has a depth of 9. We use 3 different mazes as input: one with an exit in the *rightmost* branch, one with an exit in the rightmost branch of the 5^{th} subtree from the root (we call it *center* exit), and another with an exit in the rightmost branch of the 3^{rd} subtree from the root (we call it *left* exit). The runtimes on 1 processor for the simple parallel and the perfect fuel version differ significantly: 140.60s vs 35.56s for the rightmost input (Table 1 summarises the runtimes for all parallel versions). This is explained by the total number of nodes visited in each version, using a depth first search. The perfect fuel version visits around 70 million nodes while the parallel program visits 370 million nodes.

Figure 2 shows the relative speedups for the simple parallel and the perfect fuel version of the search program using two different mazes. Note that, although the overall runtime differs significantly, the speedup is more or less the same for both programs. This shows that we achieve good utilisation of the parallel machine by generating independent processes using a `parList` in the `control` function. However, since the combination of all results in `firstSolution` requires a traversal of the entire subtree for all generated processes, far more work is done in the simple parallel version, resulting in a far higher total runtime. To achieve performance similar to the perfect fuel version, we would like to retain the structure of the parallelism and only modify the combination phase, aborting all unnecessary computations once an exit has been found.

3. Parallel Bottom-Avoiding Choice Operators

A great advantage of non-deterministic choice for searches is that the first result can be selected from parallel speculative search threads. Moreover, our `ambND` primitive kills the unsuccessful threads, avoiding wastage of computational resources. In fact we require a family of choice operators: to nest one choice within another we require `ambNDND :: ND a -> ND a -> ND a`; and it is sometimes useful to know which alternative has been successful, and this is supported by `ambEitherND :: a -> b -> ND (Either a b)`, where the `Either` type wraps the result in left or right to indicate the chosen alternative.

3.1. Semantic Properties

Non-deterministic choice has very subtle semantics, and we loosely follow Søndergaard and Sestoft's classification [20] to characterise the properties of our family of choice operators. There are so many alternative semantics that space precludes comprehensive coverage, but we briefly compare with major alternatives. For simplicity we characterise `ambND`, only distinguishing between it and other members of the family where relevant.

Locally angelic: `ambND` is angelic, or bottom avoiding [8]: it may terminate even when one of its branches is divergent, i.e.

$$\texttt{ambND}\, x \perp\ = x \quad \texttt{ambND} \perp x\ = x$$

However `ambND` is not Hoare's globally angelic choice [10]: the following fragment may diverge because zero is chosen, where an angelic choice would always terminate with value two[2].

```
do
  x <- ambND 1 0
  return (2/x)
```

Strong: a program containing `ambND` may return one of several different results [20]. For example different executions of a search program may return alternative solutions, determined by the resources available to different speculative threads. This is in contrast to *weak* non-determinism where the result, although computed non-deterministically is always the same. Under certain circumstances `ambND` can be used to implement weak non-determinism, as illustrated in Section 4.3.

Singular: as GPH is lazy, `ambND` is evaluated at most once, making a single choice, e.g. it returns either 6 or 10 in the following classic example [3]. A language with call-by-value semantics would naturally support plural choice and return 6, 8 or 10 [6].

```
do
  x <- ambND 3 5
  return ((\y -> y+y) x)
```

Distinguishing Exceptions: Haskell supports exceptions, but it is not generally possible to distinguish between divergence and an exception. It is, however, possible to return one of the exceptions if both branches of `ambND` raise one. This significantly complicates the implementation, requiring a process to count the exceptions returned and is omitted from the implementation given in Section 3.4

Unbounded: We do not guarantee that every computation that can be used as an argument to `ambND` has a finite number of possible results. We have chosen this property because we do not want to restrict the class of computations in Haskell that might be input to `ambND`.

3.2. Implementation Properties

In this section we discuss key requirements for the implementation of a non-deterministic choice. We cover both operational properties and requirements for an efficient implementation.

Lethal: `ambND` terminates *all* child threads, including any nested threads. This potentially saves resources, e.g. terminating unsuccessful searches. Our implementation kills all child threads, using Haskell's asynchronous exception mechanism, and does not incure additional runtime-system cost locate the child threads. Furthermore, computations of a terminated speculative thread are retained by automatically freezing a partial computation, i.e. converting it into a heap structure that represents the partially evaluated data. This introduces a limited amount of overhead in freezing the computation, to avoid wasting an unbounded amount of computation done by the thread.

Fair: `ambND` has a simple notion of fairness in using round-robin scheduling for all threads, both speculative and mandatory. Hence, computations of similar duration have similar chances of being selected. In contrast many other

[2]Haskell `do` notation combines a sequence of monadic operations and is analogous to sequential composition in imperative languages

implementations assign priorities to speculative threads, typically lower priority than mandatory threads. Others have variable priorities often based on the depth of nested speculation, e.g. [12, 15].

Synchronous: `ambND` blocks until a result is in weak head-normal form, whereas for the operators that allow nesting (such as `ambNDND`), it blocks while the nested `ambND` operators are evaluating.

State preserving: The initial temptation may be to use the `IO` monad to carry the results of the choice operator. Thus allowing non-deterministic I/O operations; for example:

```
do
  x <- ambEitherIOIO (writefile f "foo")
                     (writefile f "bar")
  case x of
    (Left _)  -> putStrLn "foo"
    (Right _) -> putStrLn "bar"
```

However, while the result reported by `x` is the I/O action that finished first, this does not guarantee that other I/O actions were not performed. Unlike pure computations, the world state is not a transaction that can be reversed. The new `ND` monad is introduced to avoid such problems and represents non-deterministic choice between computations rather than between IO interactions.

3.3. GDH

Glasgow Distributed Haskell (GDH) [17] is a distributed functional language and a superset of both GPH and Concurrent Haskell [16]. It supports the creation and distribution of stateless and stateful (pure and impure) threads. Stateless threads are created by the GPH primitives, and stateful or IO threads are created using the `forkIO :: IO () -> IO ThreadId` command of Concurrent Haskell. In contrast to GPH, GDH schedules threads fairly. GDH allows the creation of remote threads using `rforkIO`, analogous to `forkIO` except with an additional argument identifing the processor where the thread must be created. All threads, stateless and stateful, synchronise and communicate implicitly to share stateless values, but I/O threads can communicate and synchronise explicitly using shared mutable variables (`MVars`), of Concurrent Haskell, and by the use of asynchronous exceptions.

3.4. Implementation

The ND monad is defined with an empty state as follows:

```
newtype ND a = ND (() -> IO a)

instance Monad ND where
 return a     = ND $ \info -> return a
 (ND a) >>= f = ND $ \info ->  do
                  v <- a info
                  case f v of
                    ND a -> a info
```

Since the `ND` monad is directly implemented using the `IO` monad the monad laws hold [22].

The choice operators are implemented using the following private generic GDH primitive that chooses the first computation to complete from a list of computations.

```
ambListIOIO :: [IO a] -> IO a
ambListIOIO [] = error "no result"
ambListIOIO js = do
  mv <- newEmptyMVar
  let
   care = do
      ts <- mapM (\j -> rforkAnyIO (work j)) js
      cr <- catchAllIO (takeMVar mv)
                       (\e-> return (Fail e))
      mapM killThread ts
      case cr of
        Okay r -> return r
        Fail e -> throw e
    work job = do
      r <- job
      tryPutMVar mv (Okay r)
      return ()
  blockAsyncExceptions care
```

`ambListIOIO` uses two features of Concurrent Haskell: firstly an MVar (`mv`) is used to hold the result and only the first value written to it, by `tryPutMVar`, will succeed. Secondly exceptions are used to kill all child threads (`mapM killThread ts`) and these exceptions are recursively caught by the child threads and passed down to their children (`catchAllIO...throw...`).

`ambListIOIO` works as follows: an MVar (`mv`) is created to hold the result, exceptions are then blocked while `rforkAnyIO` is used to create a remote parallel thread to evaluate each job in `js`. Exceptions are automatically reenabled by the `takeMVar` which then waits either for a result or an exception. Upon either of these events all child threads are killed and the result is either returned or the exception is thrown again. Meanwhile, each parallel child thread is executing `work`: evaluating the given `job` before attempting to put the result into the MVar.

The family of choice operators are easily implemented using `ambListNDND` (a direct wrapping of `ambListIOIO` in the ND monad), for example:

```
ambNDND :: ND a -> ND a -> ND a
ambNDND x y = ambListNDND [x,y]

ambND :: a -> a -> ND a
ambND x y = ambNDND (seqND x) (seqND y)
  where seqND w = w `seq` return w
```

Table 1. Maze Search Runtimes (in seconds)

	Sequential	1 processor Par	1 processor Fuel	1 processor ambND	8 processors Par	8 processors Fuel	8 processors ambND
Left Exit	48.28	50.01	14.21	35.12	18.74	6.25	4.57
Central Exit	92.60	94.03	26.28	39.02	26.11	6.01	4.89
Rightmost Exit	137.32	140.60	35.56	39.00	26.48	5.86	5.85

4. Speculative Evaluation with `ambND`

4.1. Maze Search Example

The maze program needs minor modifications to use the `ambND` operator: `ambNDND` is folded (using `foldr1`) over the list of search tasks in the top level of the tree.

```
searchamb ::(a->Bool) -> (a->Bool) ->
             (a->[a]) -> [a] -> a -> ND [a]
searchamb isExit cont succ solutions node
 = foldr1 ambNDND (map workND js)
   where
     js = map (search isExit cont succ
       (node:solutions)) (succ node)
```

The `workND` function forces the evaluation of the tree using the `rnf` strategy and raises an exception if the value computed is not a solution to the problem.

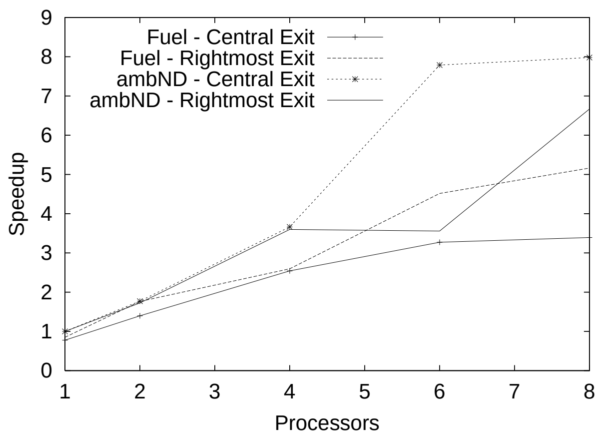

Figure 3. Maze Search Speedups with `ambND`

Figure 3 compares the relative speedups of the `ambND` with the perfect fuel version. Whereas all parallelism in the perfect fuel version is conservative, i.e. the total amount of work by all processors is the same as on one processor, the `ambND` version stops the search after finding the first solution and can therefore achieve even super-linear speedup. Since a central exit is found earlier than a rightmost exit, the improvement is most pronounced for the latter input.

Table 1 allows a more detailed comparison of the runtimes. Most importantly we observe the best parallel runtimes for the `ambND` program for all three mazes. In fact, we can even use an infinite maze as input, and achieve similar speedup figures.

As expected, the sequential times of all `ambND` executions are roughly the same, matching the time of the perfect fuel version on a maze with a rightmost exit. From an operational point of view, the fair scheduling of `ambND` threads induces a breadth first search in the tree resulting in similar execution times for all mazes. The higher runtime of the `ambND` version for the rightmost exit can be attributed to overhead of the `ambND` primitive and more expensive fair scheduling required by GDH. However, on eight processors the `ambND` program is in fact faster than the perfect fuel program because it also avoids maintaining an extra fuel parameter. In assessing the overhead for the `ambND` primitive we measured a call of `ambND () ()` to take about $420\mu s$ (including function call and system startup time). Although this is significantly higher than the overhead of creating potential parallelism in GPH (adding a pointer to an array), it is low enough to justify its use even for massive parallelism.

In general the reduction of runtime in the `ambND` version comes at the expense of increased resource consumption due to fair scheduling in GDH. In particular, separate measurements show an increase in heap residency proportional to the number of generated threads (80 in these measurements) on a single processor. However, in an 8 processor execution the startup delay introduced by communication latency reduces the number of concurrently running threads on each PE. As a result the average heap residency of the `ambND` version on 8 PEs is within a factor of 4 of the residency for the parallel fuel version.

4.2. Fair Merge

Fair Merging is a classical operation in reactive systems that combines two potentially infinite streams into a single stream, and our implementation works as follows. The `ambEitherND` applies the `bang` function to both input lists, which forces the evaluation of the first list element. `ambEitherND` tags the first element with `Left` or `Right` to indicate which list it came from enabling the `case r` to remove it from the appropriate input list. A special case exists for when one of the input lists is empty meaning that the whole of the remaining list may be returned. To construct the result list lazily the `lazyConsND` function is used.

`MergeND` preserves unfoldability because the result of the merge is returned in the `ND` monad. The potentially disastrous `unsafePerformND` (of type `ND a -> a`) is used safely as the unsafe result is bound to just one variable and, with just a single laziness-preserving cons interposed, injected back into the `ND` monad.

The fairness of `mergeND` is readily demonstrated in practice; it is fair because `ambND` is fair with an even chance of chosing arguments of similar computation cost, and `ambEitherND` choses between a pair of list constructors that have similar costs, unlike list elements.

```
mergeND :: [a] -> [a] -> ND [a]
mergeND sx sy = do
 let
  bang []    = Nothing   -- empty stream
  bang (h:t) = Just(h,t)

  lazyConsND h jt =
           return (h:(unsafePerformND jt))
 r <- ambEitherND (bang sx) (bang sy)
 case r of
  (Left  (Just(h,t))) ->
        lazyConsND h (mergeND t sy)
  (Left   Nothing)    -> return sy
  (Right (Just(h,t))) ->
        lazyConsND h (mergeND sx t)
  (Right  Nothing)    -> return sx
```

4.3. Speculative conditionals

Speculative conditionals are a parallel language primitive, and we consider a conditional that introduces speculative parallelism by evaluating the true branch in parallel with the condition. The `coord` function uses `ambND` to evaluate the conditional (`c`) and true (`t`) branch in parallel. Care must be taken to guarantee that the conditional is always evaluated (`(rnf t) 'seq' c`, and if the condition is true to wait until the true branch has completed rather than discard evaluation to date (`(rnf c) 'seq' (if c then (t 'seq' ())`. The `demanding` function forces the evaluation of its second argument, a strategy application, and returns its first argument. `nondet` is a function used to force the evaluation of `coord` and it has type `nondet :: ND () -> ()`. `optimisticIf` demonstrates how `ambND` can be used to express *weak* non-determinism [20], i.e. the conditional always returns the branch selected by the predicate.

```
optimisticIf :: Bool -> a -> a -> a
optimisticIf c t f =(if c then t else f)
              'demanding' (nondet coord)
  where coord = ambND
         ((rnf t) 'seq' c 'seq' ())
         ((rnf c) 'seq' (if c then
                         (t 'seq' ()) else ()))
```

5. Related Work

Language Constructs: Non-deterministic constructs have long been considered important for either reactive systems, e.g. to merge streams of events in an operating system, or for parallel computation, e.g. to achieve speculative parallelism. Such constructs have been studied in [6, 13] and their implementations in [1].

Hughes and O'Donnell [9] define an abstract data type of sets to represent the possible result values of a non-deterministic operator. Since these sets can only be treated in their entirety, unfoldability is maintained. Furthermore, the operations permitted on the set type allow a cheap implementation, which identifies a set with any representative for the set. Non-deterministic choice then amounts to ordinary set union. More recent work proposes to use a monadic programming style, similar to our `ND` monad [22]. In [5] an operator `chooseIO :: a -> b -> IO Bool` provides bottom-avoiding choice with implicit killing of redundant threads. It is implemented in Concurrent Haskell [16] for use in reactive systems, whereas we work in a parallel system. The `spec` primitive implemented in a simulator by Checkland [2] behaves similarly to GPH's `par`, but treats its threads speculatively: a priority scheduler with automatic upgrading of relevant and killing of irrelevant threads is used.

Implementations: Most closely related to our work is [18], which extends the strict, dynamically typed language KiR with a speculative case construct, generating parallelism for all branches of the case. A fair priority scheduler is implemented by maintaining step counts in an interpreted environment, and unnecessary threads are automatically killed. However, as a system for high performance computation such an interpretive system is unrealistic and, as the authors admit, their main contribution lies in controlling the resource consumption in the presence of speculative computation. Performance measurements are reported on the π-RED^{+} graph reducer running on a 32-processor nCube machine. Other closely related implementations of speculative parallelism are [12, 15]. Both implement lazy functional languages using a parallel graph reduction machine and use a priority scheduler with annotations to mark expressions for which speculative threads should be generated. The aim is to generate additional local parallelism to hide the communication latency in a system with implicit parallelism. The usage of an annotation, rather than a language primitive, makes it hard to build abstractions such as our merge construct. Another implementation of speculative parallelism, based on the runtime-system for

GPH and supporting priorities and automatic thread killing is currently under development [4].

Semantics: Early studies of the semantic properties of McCarthy's `amb` in a referentially transparent language often focused on the impact of different parameter passing styles [7]. Søndergaard and Sestoft [20] give a detailed discussion of semantic properties of functional languages in the presence of non-deterministic constructs and present both singular and plural semantics for `amb`. Hughes and Moran [8] present a singular semantics for normal-order λ-calculus with `amb`, capturing the notion of fairness, and a more detailed discussion of the semantic properties is given in Moran's PhD thesis [14].

6. Conclusions

We have implemented and measured a family of parallel bottom-avoiding choice operations that preserve unfoldability, and carefully described their semantic and operational properties. The operations support the efficient implementation of speculative parallel algorithms, and have been demonstrated in several examples and shown to deliver good speedup. The `ambND` primitives are the first example of using GDH to construct parallel skeletons for GPH.

Our design and implementation use a number of features from the Haskell family of languages. Type classes are used to define the `ND` monad. GDH's distributed exceptions are used to kill threads that may reside on a remote processor. Properties of the GDH implementation utilised are the ability to create mandatory, placed computations, a fair scheduler and the automatic freezing of computations to avoid wasting computation when killing threads.

Future directions include the usage of bottom-avoiding non-determinism in reactive GDH programs, and improvements of the implementation, e.g. by using a priority scheduler. An operational semantics for the choices, possibly based on [14], would facilitate proof of semantic properties.

References

[1] F. Burton. Nondeterminism with Referential Transparency in Functional Programming Languages. *Computer Journal*, 31(3):243–247, 1988.

[2] I. Checkland. *Speculative Concurrent Evaluation in a Lazy Functional Language*. PhD thesis, Department of Computer Science, University of York, 1995.

[3] W. Clinger. Nondeterministic Call by Need is Neither Lazy Nor by Name. In *LFP'82 — Conf on LISP and Functional Programming*, pages 226–234, Pittsburgh, PA, 1982.

[4] J. Dermoudy. *Effective Runtime Management of Parallelism in a Functional Programming Context*. PhD thesis, University of Tasmania, 2002. Under assessment.

[5] S. Finne and S. Peyton Jones. Programming Reactive Systems in Haskell. In *Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 50–66, Ayr, Scotland, Sept. 1994. Springer-Verlag.

[6] P. Henderson. *Functional Programming — Application and Implementation*, chapter Non-deterministic Primitives and Backtrack Problems. Prentice-Hall, 1980.

[7] M. Hennessy and E. Ashcroft. Parameter Passing Mechanisms and Nondeterminism. In *Symp on the Theory of Computing*, Boulder, CO, 1977.

[8] R. Hughes and A. Moran. Making Choices Lazily. In *FPCA'95 — Intl Conf on Functional Programming Languages and Computer Architecture*, pages 108–119, La Jolla, CA, June 1995. ACM Press.

[9] R. Hughes and J. O'Donnell. Expressing and Reasoning about Nondeterministic Functional Programs. In *Glasgow Workshop on Functional Programming*, pages 308–328. Springer-Verlag, 1989.

[10] J. Kennaway and C. Hoare. A Theory of Nondeterminism. In *Intl Colloquium on Automata, Languages, and Programming*, LNCS 85, pages 338–350. Springer-Verlag, 1980.

[11] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency – Practice and Experience*, 11:701–752, 1999.

[12] J. Mattson Jr. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction*. PhD thesis, University of California, San Diego, Feb. 1993.

[13] J. McCarthy. A Basis for a Mathematical Theory of Computations. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[14] A. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University, 1998.

[15] A. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*. PhD thesis, University of Tasmania, 1991.

[16] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL'96 — Symp on Principles of Programming Languages*, pages 295–308. ACM Press, Jan. 1996.

[17] R. Pointon, P. Trinder, and H.-W. Loidl. The Design and Implementation of Glasgow distributed Haskell. In *IFL'00 — Intl Workshop on the Implementation of Functional Languages*, LNCS 2011, pages 53–70. Springer-Verlag, 2000.

[18] R. Schröder and W. Kluge. Organizing Speculative Computations in Functional Systems. In *IFL'00 — Intl Workshop on the Implementation of Functional Languages*, LNCS 2011, pages 214–230. Springer-Verlag, 2001.

[19] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[20] H. Søndergaard and P. Sestoft. Non-determinism in Functional Languages. *Computer Journal*, 35(5):514–523, 1992.

[21] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.

[22] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.