J. Functional Programming **1** (1): 1–000, January 1993

Algorithm + Strategy = Parallelism

P.W. TRINDER

Department of Computing Science, University of Glasgow, Glasgow, UK

K. HAMMOND

Division of Computing Science, University of St Andrews, St Andrews, UK

H.-W. LOIDL AND S.L. PEYTON JONES [†]

Department of Computing Science, University of Glasgow, Glasgow, UK

Abstract

The process of writing large parallel programs is complicated by the need to specify both the parallel behaviour of the program and the algorithm that is to be used to compute its result. This paper introduces *evaluation strategies*, lazy higher-order functions that control the parallel evaluation of non-strict functional languages. Using evaluation strategies, it is possible to achieve a clean separation between algorithmic and behavioural code. The result is enhanced clarity and shorter parallel programs.

Evaluation strategies are a very general concept: this paper shows how they can be used to model a wide range of commonly used programming paradigms, including divide-and-conquer, pipeline parallelism, producer/consumer parallelism, and data-oriented parallelism. Because they are based on unrestricted higher-order functions, they can also capture irregular parallel structures.

Evaluation strategies are not just of theoretical interest: they have evolved out of our experience in parallelising several large-scale parallel applications, where they have proved invaluable in helping to manage the complexities of parallel behaviour. These applications are described in detail here. The largest application we have studied to date, Lolita, is a 60,000 line natural language parser. Initial results show that for these programs we can achieve acceptable parallel performance, while incurring minimal overhead for using evaluation strategies.

1 Writing Parallel Programs

While it is hard to write good sequential programs, it can be considerably harder to write good parallel ones. At Glasgow we have worked on several fairly large parallel programming projects and have slowly, and sometimes painfully, developed a methodology for parallelising sequential programs.

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly parallel programs

[†] This work is supported by the UK EPSRC (Engineering and Physical Science Research Council) AQUA and Parade grants.

must also specify *how* the machine should organise the computation. There are many aspects to the parallel execution of a program: threads are created, execute on a processor, transfer data to and from remote processors, and synchronise with other threads. Managing all of these aspects on top of constructing a correct and efficient algorithm is what makes parallel programming so hard. One extreme is to rely on the compiler and runtime system to manage the parallel execution without any programmer input. Unfortunately, this purely implicit approach is not yet fruitful for the large-scale functional programs we are interested in.

A promising approach that has been adopted by several researchers is to delegate most management tasks to the runtime system, but to allow the programmer the opportunity to give advice on a few critical aspects. This is the approach we have adopted for Glasgow Parallel Haskell (GPH), a simple extension of standard non-strict functional language Haskell (Peterson *et al.*, 1996) to support parallel execution.

In GPH, the runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might usefully be evaluated by parallel threads, and since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*. Even with such a simple parallel programming model we find that more and more of such code is inserted in order to obtain better parallel performance. In realistic programs the algorithm can become entirely obscured by the code describing the dynamic behaviour.

1.1 Evaluation Strategies

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the algorithm and the strategy, with graph reduction allowing values defined in the former to be manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the parallel behaviour.

The primary benefits of the evaluation strategy approach are similar to those that are obtained by using laziness to separate the different parts of a sequential algorithm (Hughes, 1983): the separation of concerns makes both the algorithm and the dynamic behaviour easier to comprehend and modify.

Because evaluation strategies are written using the same language as the algorithm, they have several other desirable properties.

- Strategies are powerful: simpler strategies can be composed, or passed as arguments to form more elaborate strategies.
- Strategies can be defined over all types in the language.
- Strategies are extensible: the user can define new application-specific strategies.
- Strategies are type safe: the normal type system applies to strategic code.
- Strategies have a clear semantics, which is precisely that used by the algorithmic language.

Evaluation strategies have been implemented in GPH and used in a number of large-scale parallel programs, including data-parallel complex database queries, a divide-and-conquer linear equation solver, and a pipelined natural-language processor, Lolita. Lolita is large, comprising over 60K lines of Haskell. Our experience shows that strategies facilitate the top-down parallelisation of existing programs.

1.2 Structure of the Paper

The remainder of this paper is structured as follows. Section 2 describes parallel programming in GPH. Section 3 introduces evaluation strategies. Section 4 shows how strategies can be used to specify several common parallel paradigms including pipelines, producer/consumer and divide-and-conquer parallelism. Section 5 discusses the use of strategies in three large-scale applications. Section 6 discusses related work. Finally, Section 7 concludes.

2 Introducing Parallelism

Parallelism is introduced in GPH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p `par` e` (here we use Haskell's infix operator notation) has the same value as `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*. Since the thread is not necessarily created, `p` is similar to a *lazy future* (Mohr *et al.*, 1991). Note that `par` differs from parallel composition in process algebras such as CSP (Hoare, 1985) or CCS (Milner, 1989) by being an asymmetric operation – at most one new parallel task will be created.

Since control of sequencing can be important in a parallel language (Roe, 1991), we therefore introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 `seq` e2` has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`.

Since both `par` and `seq` are projection functions, they are vulnerable to being altered by optimising transformations, and care must be taken in the compiler to protect them. The implementation of the compositions is described more fully in (Trinder *et al.*, 1996).

2.1 Simple Divide-and-Conquer Functions

Let us consider the parallel behaviour of `pfib`, a very simple divide-and-conquer program. If `n` is greater than 1, then `pfib (n-1)` is sparked, and the thread continues to evaluate `pfib (n-2)`. Figure 1 shows a process diagram of the execution of `pfib 15`. Each node in the diagram is a function application, and each arc is the data value, in this case an integer, used to communicate between the invocations. Note that `seq` has a higher precedence than `par`.

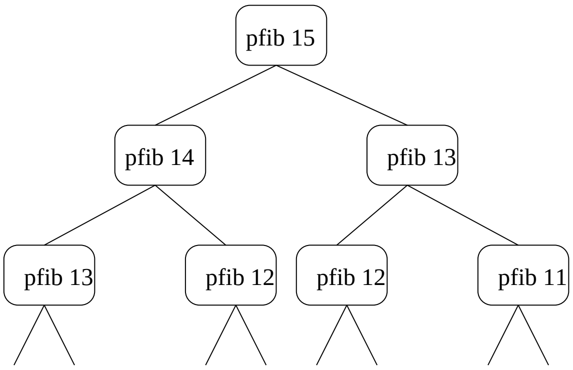

Fig. 1. pfib Divide-and-conquer Process Diagram

```
pfib n
  | n <= 1 = 1
  | otherwise = n1 `par` n2 `seq` n1+n2+1
                where
                  n1 = pfib (n-1)
                  n2 = pfib (n-2)
```

Parallel quicksort is a more realistic example, and we might write the following as a first attempt to introduce parallelism.

```
quicksortN :: [a] -> [a]
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) = losort `par`
                    hisort `par`
                    losort ++ (x:hisort)
                    where
                      losort = quicksortN [y|y <- xs, y < x]
                      hisort = quicksortN [y|y <- xs, y >= x]
```

The intention is that two threads are created to sort the lower and higher halves of the list in parallel with combining the results. Unfortunately `quicksortN` has almost no parallelism because threads in GPH terminate when the sparked expression is WHNF. In consequence, all of the threads that are sparked to construct `losort` and `hisort` do very little useful work, terminating after creating the first `cons` cell. To make the threads perform useful work a forcing function like `forceList` below, can be used. The resulting program has the desired parallel behaviour, and a process network similar to `pfib`, except that complete lists are communicated rather than integers.

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

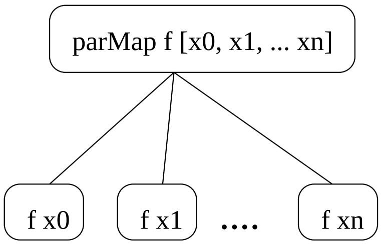

Fig. 2. parMap Process Diagram

```
quicksortF []       = []
quicksortF [x]      = [x]
quicksortF (x:xs)   = (forceList losort) `par`
                      (forceList hisort) `par`
                      losort ++ (x:hisort)
                      where
                        losort = quicksortF [y|y <- xs, y < x]
                        hisort = quicksortF [y|y <- xs, y >= x]
```

2.2 Data-Oriented Parallelism

Quicksort and pfib are examples of (divide-and-conquer) *control-oriented* parallelism where subexpressions of a function are identified for parallel evaluation. *Data-oriented parallelism* is an alternative approach where elements of a data structure are evaluated in parallel. A parallel map is a useful example of data-oriented parallelism; for example the `parMap` function defined below applies its function argument to every element of a list in parallel.

```
parMap :: (a -> b) -> [a] -> [b]
parMap f [] = []
parMap f (x:xs) = fx `par` fxs `seq` (fx:fxs)
                    where
                      fx = f x
                      fxs = parMap f xs
```

The definition above works as follows: `fx` is sparked, before recursing down the list (`fxs`), only returning the first constructor of the result list after every element has been sparked. The process diagram for parMap is given in Figure 2. If the function argument supplied to `parMap` constructs a data structure, it must be composed with a forcing function in order to ensure that the data structure is constructed in parallel.

2.3 Evaluation Degree + Parallelism = Dynamic Behaviour

As the examples above show, a parallel function must describe not only the algorithm, but also some important aspects of how the parallel machine should organise the computation, i.e. the function's dynamic behaviour. In GpH, there are two components to this dynamic behaviour:

- *Parallelism control*, which specifies what threads should be created, and in what order, using `par` and `seq`.
- *Evaluation degree*, which specifies how much evaluation each thread should perform. In the examples above, forcing functions were used to describe the evaluation degree.

Evaluation degree is closely related to strictness. If the evaluation degree of a value in a function is less than the program's strictness in that value then the parallelism is *conservative*, i.e. no expression is reduced in the parallel program that is not reduced in its lazy counterpart. In several programs we have found it useful to evaluate some values *speculatively*. That is, the evaluation-degree may usefully be more strict than the lazy function.

In the examples above, the code describing the algorithm and dynamic behaviour are intertwined, and as a consequence both have become rather opaque. In larger programs, and with carefully-tuned parallelism, the problem is far worse.

3 Strategies Separate Algorithm from Dynamic Behaviour

The driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

3.1 Evaluation Strategies

An *evaluation strategy* is a function that specifies the dynamic behaviour of an algorithmic function. In order to allow evaluation strategies to specify the degree to which the algorithmic function's result should be evaluated, they are parameterised over the result of the algorithmic function. Since a strategy's only purpose is to define dynamic behaviour, it is defined to return the unit type `()`.

```
type Strategy a = a -> ()
```

Strategies Controlling Evaluation Degree The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. This is surprisingly useful, e.g. when evaluating a pair the first element can be evaluated but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GpH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

A *data value* (but not a function value) can also be reduced further to *normal form* (NF) using `rnf`. Since we wish to define only one `rnf` operation for a list of values of any type, the obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for base types like integers and booleans, the default method for `rnf` is `rwhnf`. For constructed types an instance of `NFData` must be declared specifying how to reduce a value of that type to normal form. Such an instance relies on its element type being in class `NFData`. Consider lists and pairs for example.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf

instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

Using Strategies A strategy is applied by the `using` function. The expression ``x `using` s`` is a *projection* on `x`, i.e. it is both a retraction (``x `using` s`` is less defined than `x`) and idempotent (``(x `using` s) `using` s = x `using` s``). The `using` function is defined to have a lower precedence than any other operator.

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

Note that the use of `seq` in the definition above allows some control over the timing of results. For example, the following sequential version of `quicksort` will not return any part of its result until the entire list is sorted. This could be significant if the sort formed part of a pipeline, for example.

```
quicksortFS []      = []
quicksortFS [x]     = [x]
quicksortFS (x:xs)  = losort ++ (x:hisort) `using` rnf
                      where
                        losort = quicksortFS [y|y <- xs, y < x]
                        hisort = quicksortFS [y|y <- xs, y >= x]
```

Combining Strategies Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Strategies are most commonly composed with `seq` or `par`. Many useful strategies are higher-order, for example, `seqList` is a strategy that sequentially applies a strategy to every element of a list. The strategy `seqList r0` evaluates just the spine of a list, and `seqList rwhnf` evaluates every element of a list to WHNF. There are analogous functions for every constructed type.

```
seqList :: Strategy a -> Strategy [a]
seqList strat []     = ()
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

Parallel Strategies A strategy can specify parallelism/sequencing as well as evaluation degree. Strategies specifying control-oriented parallelism use `par` and `seq` to specify which subexpressions of a function are to be evaluated in parallel, and in what order. Quicksort uses divide-and-conquer control-oriented parallelism, and in the following version the evaluation degree is specified by `rnf`. As before, the two subexpressions, `losort` and `hisort` are selected for parallel evaluation:

```
quicksortS (x:xs) = losort ++ (x:hisort) `using` strategy
                    where
                      losort = quicksortS [y|y <- xs, y < x]
                      hisort = quicksortS [y|y <- xs, y >= x]
                      strategy result = rnf losort `par`
                                        rnf hisort `par`
                                        rnf result
```

Strategies specifying data-oriented parallelism must describe the dynamic behaviour in terms of some data structure. For example `parList` is similar to `seqList`, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Strategic functions are particularly elegant when their result is a data structure that describes the parallelism. Parallel map is just such a function:

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

The `strat` parameter determines the dynamic behaviour of each element of the result list, and hence `parMap` is parametric in some of its dynamic behaviour. Such strategic functions can be viewed as a dual to the algorithmic skeleton approach (Cole, 1988). This relationship is discussed further in Section 6.2.

4 Evaluation Strategies for Parallel Paradigms

This section demonstrates the flexibility of evaluation strategies by showing how they express some common parallel paradigms. We cover data-oriented, divide-and-conquer, producer-consumer, and pipeline parallelism.

4.1 Data-oriented Parallelism

In the data-oriented paradigm, elements of a data structure are evaluated in parallel. Complex database queries are more realistic examples of data-oriented parallelism than `parMap`. The basis of one such query is a relation between parts indicating that one part is made from zero or more others. The task is to list all component parts of a given part, including all the sub-components of those components etc. (Date, 1976).

Main Component	Sub-Component	Quantity
P1	P2	2
P1	P4	4
P5	P3	1
P5	P6	8
P2	P4	3

A naïve function `explode` lists the components of a single part, `main`. The full program generates a bill of material relation, as a list of tuples, then explodes a sequence of part numbers before printing the number of parts in each explosion.

```
explode parts main = [p | (m,s,q) <- parts, m == main,
                          p <- (s:explode parts s)]

doQuery lo hi bomSize = map length explodeList
                        where
                          bom = generate bomSize
                          explodeList = map (explode bom) [lo..hi]
```

The program is inherently data parallel because the explosion of one part is not dependent on the explosion of any other part. Constructing the bill of material in memory is atypical of a query program: a more realistic program would read it in from disk. For this reason we do not parallelise the construction `generate`. Once the bill exists, the parts are exploded in parallel. This dynamic behaviour is specified by adding a strategy to `doQuery`:

```
doQuery lo hi bomSize = map length explodeList `using` strat
                        where
                          bom = generate bomSize
                          explodeList = map (explode bom) [lo..hi]
                          strat result = (rnf bom) `seq`
                                         (parList rnf explodeList)
```

It is easy to modify both algorithm and strategy, although changing the algorithm may also entail specifying new dynamic behaviour. It is, however, easy to modify the strategy without changing the algorithm. For example, to calculate the lengths in parallel we simply add `` `seq` parList result`` to the strategy.

4.2 Divide-and-conquer Parallelism

Divide-and-conquer is probably the best-known parallel programming paradigm. The problem to be solved is decomposed into smaller problems that are solved in parallel before being recombined to produce the result. Our example is taken from a parallel linear equation solver that we wrote as a realistic medium-scale parallel program (Loidl *et al.*, 1995), whose overall structure is described in Section 5.4.

Here is the specification of a determinant on a square matrix:

- Given: a matrix $(A)_{1 \leq i,j \leq n}$
- Compute: for some $1 \leq i \leq n$: $\sum_{1 \leq j \leq n} (-1)^{i+j} A_{i,j} det(A')$ where $A' = A$ cancelling row i, and column j

```
sum (l_par `using` parList rnf)
where
  l_par = map determine1 [jLo..jHi]
  determine1 j = (if pivot > 0 then
                    sign*pivot*det' `using` strategyD
                  else
                    0) `using` sPar sign
                   where
                     sign = if (even (j-jLo)) then 1 else -1
                     pivot = (head mat) !! (j-1)
                     mat' = SqMatrixC ((iLo,jLo),(iHi-1,jHi-1))
                                      (map (newLine j) (tail mat))
                     det' = determinant mat'
                     strategyD r =
                       parSqMatrix (parList rwhnf) mat' `seq`
                       det' `par`
                       r0 r
```

For comparison, Appendix A contains sequential and directly parallel versions of this function. At first sight, it may not be obvious that this is a divide-and-conquer program. The crucial observation is that a determinant of a matrix of size `n` is computed in terms of the determinants of `n` matrices of size `n-1`.

The first strategy, `parList rnf` specifies that the determinant of each of the matrices of size `n-1` should be calculated in parallel. There are two strategies in `determine1`. The first, `sPar sign` specifies that the sign of the determinant should be calculated in parallel with the conditional (`sPar` is a strategy corresponding to `par`. i.e. `` x `par` e `` = `` e `using` sPar x``). Only if the pivot is non-zero is the second strategy, `strategyD` used. It specifies that the sub-matrix (`mat'`) is to be constructed in parallel before its determinant is computed in parallel with the result.

4.3 Producer/Consumer Parallelism

In another common paradigm, a process consumes some data structures produced by another process. In a compiler, for example, an optimising phase might consume

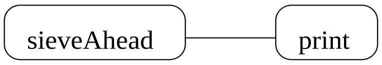

Fig. 3. Producer/Consumer Process Diagram

the parse-tree produced by the parser. The data structure can be thought of as a buffer that the producer fills and the consumer empties.

For simplicity, we will assume that the buffer is represented by a list, and consider just two alternatives: a one-place buffer and an n-place buffer. There are many other possible ways to express producer/consumer parallelism, for example in order to improve granularity the producer could compute the next n-element "chunk" of the list rather than just a single value.

One-Place Buffer In order to fill a one-place buffer, when the head of the buffer-list is demanded, the producer should immediately evaluate the second element. In effect the producer speculatively assumes that the next element in the list will be used in the computation. This gives parallel behaviour because, if there is a free processor, a producer-thread can construct the second element, while the consumer is consuming the first. If good parallelism is to result, then the time to produce an element must be similar to the time to consume it. The second element of the list acts as a one-element buffer. The simple `sieveAhead` function below eagerly produces an extra prime number using Eratosthenes' algorithm. It uses a simple strategy `parListNth` to evaluate the second element of a list in parallel (since Haskell lists are enumerated from 0, the parameter to `parListNth` is 1 rather than 2). The process diagram for producer/consumer parallelism is very simple: a producing process communicating via the buffer with the consumer. Figure 3 show the diagram for a program that prints the result of a `sieveAhead` invocation

```
parListNth :: Int -> Strategy a -> Strategy [a]
parListNth n strat xs
  | null rest = ()
  | otherwise = strat (head rest) `par` ()
  where
    rest = drop n xs

sieveAhead (p:xs) =
  p:(sieveAhead [x | x <- xs, x `mod` p /= 0]) `using` parListNth 1 rwhnf
```

n-Place Buffer To provide an n-place buffer, the producer must initially evaluate n elements, and whenever the head of the buffer-list is demanded, it must evaluate the nth element. In effect the producer eagerly fills an n-element buffer. Evaluating the first n elements of a list in parallel is easily specified by `parListN`, analogous to `parListNth`. Unfortunately constructing the nth element every time the head is demanded cannot be specified by a strategy that is independent of the result. Instead, the strategy for generating the rest of the result must be built into the result. We use a function `fringeList` whose semantics are the identity on lists, but whose dynamic behaviour is to spark the nth element when the first is demanded.

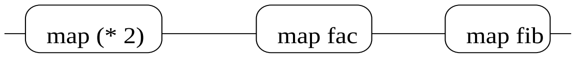

Fig. 4. Pipeline Process Diagram

`seqListNth` is analogous to `parListNth`. As an example, `doExplode` is a database query function that maps an explode function over a range of elements in a list. The list of explosions acts as a three-element buffer.

```
parListN :: (Integral b) => b -> Strategy a -> Strategy [a]
parListN n strat []     = ()
parListN 0 strat xs     = ()
parListN n strat (x:xs) = strat x `par` (parListN (n-1) strat xs)

fringeList :: (Integral a) => a -> Strategy b -> [b] -> [b]
fringeList n strat [] = []
fringeList n strat (r:rs) = seqListNth n strat rs `par`
                            r:fringeList n strat rs

doExplode lo hi bom =
  fringeList 3 rnf result `using` parListN 2 rnf
  where
    result = map (explode bom) [lo..hi]
```

4.4 Pipelines

In pipelined parallelism a sequence of stream-processing functions are composed together, each consuming the stream of values constructed by the previous stage and producing new values for the next stage. The generic `pipeline` combinator uses strategies to describe a simple pipeline, where every stage constructs values of the same type, and the same strategy is applied to the result of each stage.

```
pipeline :: Strategy a -> a -> [a->a] -> a
pipeline strat inp []     = inp
pipeline strat inp (f:fs) =
  pipeline strat out fs `using` sPar (strat out)
  where
    out = f inp

list = pipeline rnf [1..4] [map fib, map fac, map (* 2)]
```

A pipeline process diagram has a node for each stage, and an arc connecting one stage with the next. Typically an arc represents a list or stream of values passing between the stages. Figure 4 gives the process diagram for the example above.

Several of the large applications described in the next section use more elaborate pipelines where different types of values are passed between stages, and stages may use different strategies. For example, the back end in Lolita's top level pipeline is as follows:

```
back_end inp opts
 = r8 `using`  strat
   where
     r1 = unpackTrees inp
     r2 = unifySameEvents opts r1
     r3 = storeCategoriseInformation r2
     r4 = unifyBySurfaceString r3
     r5 = addTitleTextrefs r4
     r6 = traceSemWhole r5
     r7 = optQueryResponse opts r6
     r8 = mkWholeTextAnalysis r7
     strat x = (parPair rwhnf (parList rwhnf)) inp                `par`
               (parPair rwhnf (parList (parPair rwhnf rwhnf))) r1  `par`
               rnf r2                                              `par`
               rnf r3                                              `par`
               rnf r4                                              `par`
               rnf r5                                              `par`
               rnf r6                                              `par`
               (parTriple rwhnf (parList rwhnf) rwhnf) r7          `par`
               ()
```

A disadvantage of using strategies like this over long pipelines is that every intermediate structure must be named (`r1..r8`). Because pipelines are so common we have introduced two special combinators: parameterised sequential and parallel function application. The parameter specifies the strategy that is used on the argument. Therefore, we achieve the separation of algorithm and dynamic behaviour by using strategies only as the second argument to a parameterised function application.

The definition of the new combinators is as follows:

```
infixl 6 $||, $|
($|), ($||) :: (a -> b) -> Strategy a -> a -> b

($|) f s  = \  x -> f x `using` \ _ -> s x `seq` ()
($||) f s = \  x -> f x `using` \ _ -> s x `par` ()
```

We have also defined similar combinators for parameterised function composition. Pipelines can now be expressed more concisely, while retaining textual separation of strategic and algorithmic code.

```
back_end inp opts =
 mkWholeTextAnalysis     $|| parTriple rwhnf (parList rwhnf) rwhnf $
 optQueryResponse opts   $|| rnf $
 traceSemWhole           $|| rnf $
 addTitleTextrefs        $|| rnf $
 unifyBySurfaceString    $|| rnf $
 storeCategoriseInf      $|| rnf $
 unifySameEvents opts    $|| parPair rwhnf (parList (parPair rwhnf rwhnf)) $
 unpackTrees             $|| parPair rwhnf (parList rwhnf)  $
 inp
```

5 Large Parallel Applications

5.1 General

We have written a number of medium-scale parallel programs, and are currently paralleling a large-scale program, Lolita (60K lines). This section discusses the use of evaluation strategies in three programs, one divide-and-conquer, one pipelined and another data-oriented. The methodology we are developing out of our experiences is also described.

To date, parallel programming has been most successful in addressing problems with a regular structure and large grain parallelism. However, many large scale applications have a number of distinct stages of execution, and good speedups can only be obtained if each stage is successfully made parallel. The resulting parallelism is highly irregular. This makes understanding and controlling the dynamic behaviour of a large program hard. A major motivation for investigating our predominantly-implicit approach is that we believe that it is very hard to gain good speedups for large programs with irregular parallelism in languages that require the programmer to control many aspects of parallelism, e.g. thread creation, placement and synchronisation, etc.

In large applications, evaluation strategies are defined in three kinds of modules. Strategies over Prelude types such as lists, tuples and integers are defined in a Strategies module. Strategies over application-specific types are defined in the application modules. Currently, strategies over library types are defined in private copies of the library modules. Language support for strategies which automatically derived strategies over constructed types would greatly reduce the amount of code to be modified and avoid this problem of reproducing libraries.

5.2 Methodology

Our emerging methodology for parallelising large non-strict functional programs is outlined below. The approach is top-down, starting with the top level pipeline, and then parallelising successive components of the program. The first five stages are machine-independent. Our approach uses several ancillary tools, including time profiling (Sansom and Peyton Jones, 1995) and the GranSim simulator (Hammond *et al.*, 1995). Several stages use GranSim, which is fully integrated with the GUM parallel runtime system (Trinder *et al.*, 1996). A crucial property of GranSim is that it can be parameterised to simulate both real architectures and an idealised machine with, for example, zero-cost communication and an infinite number of processors.

The stages in our methodology are as follows.

1. **Sequential implementation.** Start with a correct implementation of an inherently-parallel algorithm or algorithms.
2. **Parallelise Top-level Pipeline.** Most non-trivial programs have a number of stages, e.g. lex, parse and typecheck in a compiler. Pipelining the output of each stage into the next is very easy to specify, and often gains some parallelism for minimal change.

3. **Time Profile** the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.
4. **Parallelise Big Eaters** using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm, otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. divide-and-conquer or data-parallelism.
5. **Simulate First.** Using an idealised simulator like `hbcpp` or GranSim eliminates some of the complexities of a real parallel implementation, like task migration, communication times etc. This is a "proving" step: if the program isn't parallel on an idealised machine it won't be on a real machine. A simulator is often easier to use, more heavily instrumented, and can be run on a workstation.
6. **Simulate Second.** GranSim can be parameterised to closely resemble the GUM runtime system for a particular machine, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.
7. **Real Machine.** The GUM runtime system supports some of the GranSim performance visualisation tools. This seamless integration helps understand real parallel performance.

It is more conventional to start with a sequential program and then move almost immediately to working on the target parallel machine. This has often proved highly frustrating: the development environments on parallel machines are usually much worse than those available on sequential counterparts, and, although it is crucial to achieve good speedups, detailed performance information is frequently not available. It is also often unclear whether poor performance is due to use of algorithms that are inherently sequential, or simply artefacts of the communication system or other dynamic characteristics.

5.3 Lolita

The Lolita natural language engineering system (Morgan *et al.*, 1994) has been developed at Durham University. The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The overall structure of the program bears some resemblance to that of a compiler, being formed from the following large stages:

- Morphology (combining symbols into tokens; similar to lexical analysis);
- Syntactic Parsing (similar to parsing in a compiler);
- Normalisation (to bring sentences into some kind of normal form);
- Semantic Analysis;
- Pragmatic Analysis.

Depending on how Lolita is to be used, a final additional stage may perform a discourse analysis, the generation of text (e.g. in a translation system), or it perform inference on the text to extract the required information.

Our immediate goal in parallelising this system is to expose sufficient parallelism to fully utilise a 4-processor shared memory machine. A pipeline approach is a promising way to achieve this relatively small degree of parallelism (Figure 5). Each stage listed above is executed by a separate thread, which are linked to form a pipeline. The key step in parallelising the system is to define strategies on the very complex intermediate data structures (e.g. parse trees) which are used to communicate between these stages. This data-oriented approach simplifies the top-down parallelisation of this very large system, since it is possible to define the parts parts of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures.

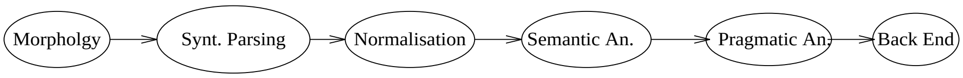

Fig. 5. Overall Pipeline Structure of Lolita

A critical issue for the Lolita system is avoiding the generation of unnecessary work. In order to achieve this, Lolita makes heavy use of laziness, for example when handling ambiguities in the parsing of natural languages. The overall efficiency of the whole system depends on computing only information about the quality of alternative parses, and not the parse trees themselves. This avoids the construction of large superfluous data structures. Consequently, using a strategy that is stricter than necessary may increase the parallelism in the parsing stage but decrease overall performance.

We are currently at the *Simulate First* stage of our parallelising methodology. So far, the pipeline approach has produced an average parallelism between 2.3 and 2.7. Since Lolita was originally written without any consideration for parallel execution, we are fairly satisfied with this amount of parallelism. Amdahl's law gives an upper bound for speedup of about 3 if only 10% of the code is inherently sequential!

Apart from specifying instances of `NFData` for intermediate data structures, to achieve this parallelisation it was only necessary to modify one of about three hundred modules in Lolita and three of the thirty six functions in that module. At this stage, we haven't parallelised any of the sub-algorithms, which also contain significant sources of parallelism.

To achieve more parallelism we plan to consider two parts of the pipeline.

Firstly, both of the first two stages (morphology and syntactic parsing) can be applied to different parts of the text in parallel. So several sentences can be parsed simultaneously. This creates data-parallelism in the first part of the pipeline that will be especially effective in improving performance for large inputs.

Similarly, the semantic and pragmatic analyses can be applied in a data-parallel fashion on different possible parse trees for the same sentence. Such parallelism would not increase the performance of the system but it might improve the quality of the result.

The analyses also produce information that is put into a 'global context' containing information about the semantics of the text. This creates an additional

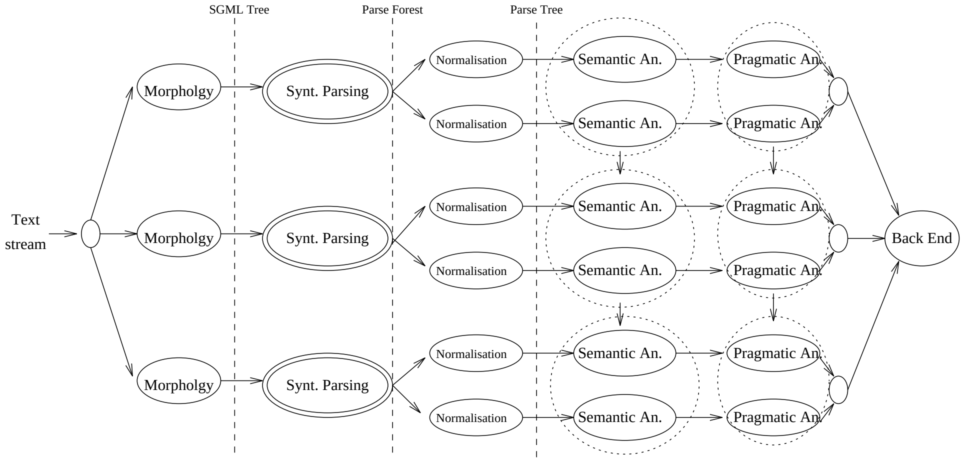

Fig. 6. Detailed Structure of Lolita

dependence between different instances of the analysis. Lazy evaluation ensures that this does not completely sequentialise the analyses, however.

Finally, it seems worthwhile to parallelise the rather expensive syntactic parsing stage itself. Figure 6 shows the more detailed structure that results.

The code of the top level function `wholeTextAnalysis` is given in Figure 7. This clearly shows how the algorithm is separated from the dynamic behaviour in each stage.

The only changes in the algorithm are

1. the use of `parMap` to describe the data parallelism in the parsing stage; and
2. the use of parameterised function applications to describe the overall pipeline structure.

The strategies used in `parse2prag` are of special interest. The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied to a predefined number (`global`) of these parses. The strategy that is applied to the list of these results (`parList (parPair ...)`) demands only the score of each analysis (the first element in the triple), and not the complete parse. This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis.

5.4 Linsolv

Linsolv is a linear equation solver, and a typical example of a parallel symbolic program. It uses the multiple homomorphic images approach which is often used in computer algebra algorithms (Lauer, 1982): first the elements of the input matrix and vector are mapped from $\mathbf{Z}$ into several images $\mathbf{Z}_p$ (where each p is a prime number); then the system is solved in each of these images, and finally the overall result is constructed by combining these solutions using the Chinese Remainder Algorithm. This divide-and-conquer structure is depicted by Figure 8.

```
wholeTextAnalysis opts inp global =
  result
  where
    -- (1) Morphology
    (g2, sgml) = prepareSGML inp global
    sentences  = selectEntitiesToAnalyse global sgml

    -- (2) Parsing
    rawParseForest = parMap rnf (heuristic_parse global) sentences

    -- (3)-(5) Analysis
    anlys = stateMap_TimeOut (parse2prag opts) rawParseForest global2

    -- (6) Back End
    result = back_end anlys opts

-- Pick the parse tree with the best score from the results of
-- the semantic and pragmatic analysis.  This is done speculatively!

parse2prag opts parse_forest global =
 pickBestAnalysis global  $|| evalScores $
 take (getParsesToAnalyse global)       $
 map analyse parse_forest
 where
   analyse pt =   mergePragSentences opts $ evalAnalysis
   evalScores =   parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))
   evalAnalysis = stateMap_TimeOut analyseSemPrag pt global

-- Pipeline the semantic and pragmatic analyses
analyseSemPrag parse global =
 prag_transform            $|| rnf   $
 pragm                     $|| rnf   $
 sem_transform             $|| rnf   $
 sem (g,[])                $|| rnf   $
 addTextrefs global        $|  rwhnf $
 subtrTrace global parse

back_end inp opts =
 mkWholeTextAnalysis     $|| parTriple rwhnf (parList rwhnf) rwhnf $
 optQueryResponse opts   $|| rnf $
 traceSemWhole           $|| rnf $
 addTitleTextrefs        $|| rnf $
 unifyBySurfaceString    $|| rnf $
 storeCategoriseInf      $|| rnf $
 unifySameEvents opts    $|| parPair rwhnf (parList (parPair rwhnf rwhnf)) $
 unpackTrees             $|| parPair rwhnf (parList rwhnf)  $
 inp
```

Fig. 7. The Top Level Function of the Lolita Application

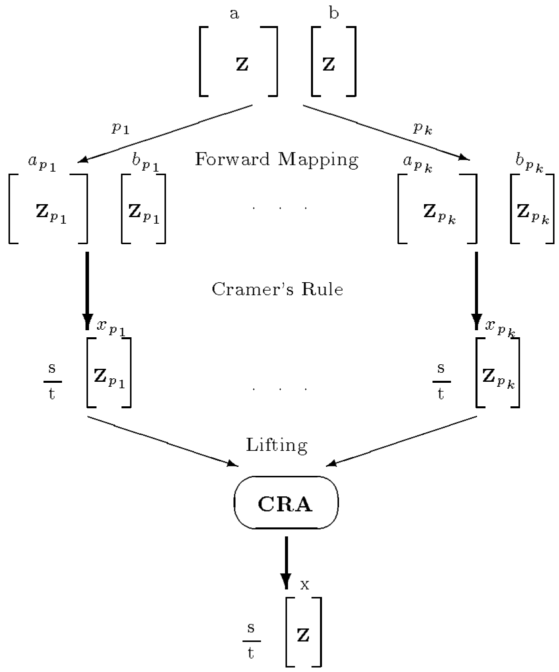

Fig. 8. Structure of the LinSolv algorithm

Strategic code for the matrix determinant part of the solver is given in Section 4.2 (the whole algorithm is discussed in (Loidl *et al.*, 1995)). Precise control of the dynamic behaviour is required at two critical places in the program. This behaviour can be described by combining generic strategies.

- The algorithm is described in terms of an *infinite list* of all solutions in the homomorphic images. An initial segment of the list is computed in parallel, based on an educated guess as to how many homomorphic solutions are needed. Depending on the solutions in the initial segment, a small number of additional solutions are then computed.
- The algorithm only computes the solutions that can actually be used in the combination step. This is achieved by *initially only evaluating the first two elements* of the result list, then checking if the result is useful and if so computing the remainder.

5.5 Accident Blackspots

Given a set of 7500 police accident records, the task is to discover any accident blackspots, i.e. places where a number of accidents occurred. Several criteria are used to determine whether two accident reports are for the same location. Two accidents may be at the same location if they occurred at the same junction number, at the same pair of roads, at the same grid reference, or within a small radius of each other. The problem amounts to partitioning a set into equivalence classes under several equivalence relations.

The GpH implementation (Trinder *et al.*, 1996) has three major phases forming a top-level pipeline. These are: reading and parsing the file of accidents; constructing a combined `sameSite` relation and indices over the `accident` and `sameSite` relations; and forming the partition. Little parallelism is gained from this top-level pipeline (a speedup of 1.2) because the first value cannot be read from the index-trees until all of the tree has been constructed.

So far, the first and last pipeline stages have been adequately parallelised for our 4-processor target machine. The accidents are read in parallel from `n` separate files into a list of lists of accidents:

```
nFiles = 4

main =  readn nFiles []

readn n cts | n > 0 =
    readFile ("/path/accident"++show n)
    (\ioerror -> complainAndDie)
    (\ctsn -> readn (n-1) (ctsn:cts))

readn 0 cts =
  let accidents = map parse8Tuple cts `using` strategy
                  where strategy = parList rnf
```

The partition is parallelised by speculatively computing the equivalence classes of n (20) accidents in parallel. If two or more of the accidents are in the same class, some work is duplicated. The chance of wasting work is small as the mean class size is 4.4, and there are approximately 7500 accidents. Additional parallelism is obtained by removing members of the equivalence classes from the accident set in parallel with determining the equivalence classes (`rnf rest`). The speculation is benign because the amount of work performed by a speculative task is small, and no other threads are sparked.

```
mkPartition :: Set Accident -> IxRelation2 Accident Accident ->
               Set (Set Accident)
mkPartition accs ixRel =
  case (length aList) of
    0         -> emptySet
    n         -> (mkSet matchList `union` mkPartition rest ixRel)
                 `using` strategy
    otherwise -> (singletonSet matches) `union`
                 mkPartition (accs `minusSet` matches) ixRel
    where
      aList = take n (setToList accs)
      matches = mkSet (reachable [chose accs] ixRel)
      matchList = [mkSet (reachable [a] ixRel) | a <- aList]
      rest     = minusManySet accs matchList
      strategy result = parList rnf matchList `par`
                        rnf rest `par`
                        r0 result
```

The middle stage which constructs the indices is harder to implement in parallel. The problem is that the indices are trees, and the top-level pipeline is blocked because the first element (root) of an index-tree cannot be consumed by the following stage until all of the tree has been constructed. Our current solution splits the index into a sequence of trees, reducing the bottleneck.

6 Related Work

There have been many different proposals for ways to specify parallelism in functional languages. Space precludes describing every proposal in detail, instead this section concentrates on the approaches that are most closely related to evaluation strategies, covering purely-implicit approaches, algorithmic skeletons, coordination languages, language extensions and explicit approaches. Some non-functional approaches are also covered. The approach that is most closely related to our work is that using first-class schedules (Mirani and Hudak, 1995), described in Section 6.4.

6.1 Purely Implicit Approaches

Purely implicit approaches include dataflow languages like Id (Arvind *et al.*, 1989) or pH (Nikhil *et al.*, 1993), which is based on Haskell, and evaluation transformers (Burn, 1987). Data parallel languages such as NESL (Blelloch *et al.*, 1993) can also be seen as implicitly parallelising certain bulk data structures. All of the implicit approaches have some fixed underlying model of parallelism. Because evaluation strategies allow explicit control of some crucial aspects of parallelism, the programmer can describe behaviours very different from the fixed model, e.g. speculatively evaluating some expressions.

Evaluation Transformers Evaluation transformers exploit the results of strictness analysis on structured data types, providing parallelism control mechanisms that are tailored to individual strictness properties (Burn, 1987). Each evaluation transformer reduces its argument to the extent that is allowed by the available strictness information. The appropriate transformer is selected at compile time, giving efficient execution at the cost of some increase in code-size (Burn, 1991; Finne and Burn, 1993).

If there are only a small number of possible transformers (as for lists using the standard 4-point strictness domain – see Table 1), repeated work can be avoided by recording the extent to which a data structure has already been evaluated, and then using a specialised transformer on the unevaluated, but needed part of that structure.

One problem with evaluation transformers is that the more sophisticated the strictness analysis, and the more types they are defined on, the greater is the number of evaluation transformers that are needed, and the greater is the code-bloat. Specialised transformers must be defined in the compiler for each type, complicating the provision of transformers over programmer-defined types.

In contrast, since the programmer has control over which strategy is to be used in

Transf.	Meaning	Strategy
E_0	No reduction	`r0`
E_{WHNF}	Reduce to WHNF	`rwhnf`
E_{TS}	Reduce spine of a list	`seqList r0`
E_{HTS}	Reduce each list element to WHNF	`seqList rwhnf`

Table 1. *The Relationship of Evaluation Strategies and Transformers*

a particular context, and since those strategies are programmable rather than fixed, strategies are strictly more general than evaluation transformers. In particular, a programmer can elect to use a strategy that is more strict than the function in order to obtain good performance.

It is possible that in the future, strictness analysis could drive the choice of an appropriate evaluation strategy in at least some circumstances. Indeed we are aware of a relationship between strictness domains and the structure of certain strategies that implement those domains. Use of strictness information in this way would make strategies more implicit than they are at present.

Data Parallelism It has been argued that support should be provided for both task and data parallelism (Subhlok *et al.*, 1993). We have already shown how some kinds of data-oriented parallelism can be expressed using evaluation strategies. Truly data parallel approaches, however, such as NESL (Blelloch *et al.*, 1993; Blelloch, 1996) treat higher-order functions such as *scans* and *folds*, or compound expressions such as list- and array-comprehensions, as single “atomic” operations over entire structures such as lists or arrays.

In effect, functions are applied to each element of the data simultaneously, rather than data being supplied to the functions. This approach is more suitable than control parallelism for massively parallel machines, such as the CM-5. Certain evaluation strategies can therefore be seen as control parallel implementations of data parallel constructs, targetted more at distributed-memory or shared-memory machines than at massively parallel architectures.

Dataflow Many recent dataflow languages are functional, e.g. Id (Arvind *et al.*, 1989), indeed pH (Nikhil *et al.*, 1993) is a variant of Haskell. These languages typically use some evaluation scheme, e.g. *lenient evaluation*, to introduce parallelism implicitly. The evaluation scheme generates massive amounts of fine-grained parallelism, which is often too small to be utilised efficiently by conventional thread technology. The overheads of small grain threads have been addressed by using hardware support. The explicit control provided by evaluation strategies help the programmer to create larger grain threads.

6.2 Algorithmic Skeletons

As defined by Cole (Cole, 1988), algorithmic skeletons take the approach that implementing good dynamic behaviour on a machine is hard. A skeleton is intended to be an efficient implementation of a commonly encountered parallel behaviour on some specific machine. In effect a skeleton is a higher-order function that combines (sequential) sub-programs to construct the parallel application. The most commonly encountered skeletons are pipelines and variants of the common list-processing functions `map`, `scan` and `fold`. A general treatment has been provided by Rabhi, who has related algorithmic skeletons to a number of parallel paradigms (Rabhi, 1993).

Skeletons and Strategies Since a skeleton is simply a parallel higher-order function, it is straightforward to write skeletons using strategies. Both the `parMap` function in Section 3.1 and the `pipeline` function in Section 4.4 are actually skeletons. A more elaborate divide-and-conquer skeleton, based on a Concurrent Clean function (Nöcker *et al.*, 1991) can be written and used as follows.

```
divConq :: (a -> b) -> a -> (a -> Bool) ->
           (b -> b -> b) -> (a -> Bool) -> (a -> (a,a)) -> b
divConq f arg threshold conquer divisible divide
  | not (divisible arg) = f arg
  | otherwise     = conquer left right `using` strategy
    where
      (lt,rt)  = divide arg
      left     = divConq f lt threshold conquer divisible divide
      right    = divConq f rt threshold conquer divisible divide
      strategy = \ _ -> if threshold arg
                          then (seqPair rwhnf rwhnf) $ (left,right)
                          else (parPair rwhnf rwhnf) $ (left,right)
```

It is also possible to use strategies in the opposite way to skeletons†. A skeleton parameterises the control function over the algorithm, i.e., it takes sequential sub-programs as arguments. However, a function using strategies may instead specify the algorithm and parameterise the control information, i.e. take a strategy as a parameter. In fact several of the functions we have already described take a strategy as a parameter, including `parList`, `parMap`, and `pipeline`.

Imperative Skeletons The algorithmic skeleton approach clearly fits functional languages very well, and indeed much work has been done in a functional context. However, it is also possible to combine skeletons with imperative approaches.

For example, the Skil compiler integrates algorithmic skeletons into a subset of C (C-). Rather than using closures to represent work, as we have done for our purely functional setting, the Skil compiler (Botorog and Kuchen, 1996) translates polymorphic higher-order functions into monomorphic first-order functions. The

† such functions are not a true dual, because skeletons are lower level.

performance of the resulting program is close to that of a hand-crafted C- application. While the Skil *instantiation* procedure is not fully general, it may be possible to adopt similar techniques when compiling evaluation strategies, in order to reduce overheads.

6.3 Coordination Languages

Coordination languages build parallel programs from two components: the *computation* model and the *coordination* model (Gelernter and Carriero, 1992). Like evaluation strategies, programs have both an algorithmic and a behavioural aspect. It is not necessary for the two computation models to be the same paradigm, and in fact the computation model is often imperative, while the coordination language may be more declarative in nature. Programs developed in this style have a two-tier structure, with sequential processes developed using the computation language composed using the coordination language.

The best known coordination languages are PCN (Foster and Taylor, 1994) and Linda (Gelernter and Carriero, 1992). Both of these adopt a much lower-level approach than evaluation strategies, however. It is, of course, possible to introduce deadlock with either of these systems.

PCN composes tasks by connecting pairs of communication ports, using three primitive composition operators: sequential composition, parallel composition and choice composition. It is possible to construct more sophisticated parallel structures such as divide-and-conquer, and these can be combined into libraries of reusable templates.

Linda is built on a logically shared-memory structure. Objects (or *tuples*) are held in a shared area: the Linda *tuple space*. Linda processes manipulate these objects, passing values to the sequential computation language. In the most common Linda binding, C-Linda, this is C. Sequential evaluation is therefore performed using normal C functions.

SCL Darlington et al. integrate the coordination language approach with the skeleton approach, providing a system for composing skeletons, SCL (Darlington *et al.*, 1995). SCL is basically a data-parallel language, with distributed arrays used to capture not only the initial data distribution, but also subsequent dynamic redistributions.

SCL introduces three kinds of skeleton: *configuration*, *elementary* and *computational* skeletons. Configuration skeletons specify data distribution characteristics, elementary skeletons capture the basic data parallel operations as the familiar higher-order functions *map*, *fold*, *scan* etc. Finally, computational skeletons add control parallel structures such as *farms*, *SPMD* and iteration. It is possible to write higher-order operations to transform configurations as well as manipulate computational structures etc. An example taken from Darlington et al., but rewritten in Haskell-style, is the `partition` function, which partitions a (sequential) array into a parallel array of `p` sequential subarrays.

```
partition :: Partition_pattern -> Array Index a ->
             ParArray Index (Array Index a)

partition (Row_block p) a = mkParArray [ ii := b ii | ii <- [1..p] ]
  where b l = array bounds [ (i,j) := a ! (i+(ii-1)*l/p, j)
                           | i <- [1..l/p], j <- [1..m] ]
        bounds = ((1,l/p), (1,m))
```

A similar integration is provided by the P^{3}L language (Danelutto *et al.*, 1991), which provides a set of skeletons for common classes of algorithm.

Control Abstraction Another approach which has certain parallels with evaluation strategies has been described by Crowl and Leblanc (Crowl and Leblanc, 1994), who work with explicitly parallel imperative programs (including explicit synchronisation and communication, as well as explicit task creation).

Like evaluation strategies, the control abstraction approach also separates parallel control from the algorithm. Each control abstraction comprises three parts: a prototype specifying the types and names of the parameters to the abstraction; a set of control dependencies that must be satisfied by all legal implementations of the control abstraction; and one or more implementations.

Each implementation is effectively a higher-order function, parameterised on one or more closures representing units of work that could be performed in parallel. These closures are invoked explicitly within the control abstraction. Implementations can use normal language primitives or other control abstractions.

In our purely functional context, Crowl and Leblanc's control dependencies correspond precisely to the evaluation degree of a strategy. Their requirement that implementations conform to the stated control dependencies is thus equivalent in our setting to requiring that strictness is preserved in any source-to-source transformation involving an evaluation strategy. This is, of course, a standard requirement for any transformation in a non-strict functional language.

Compared with the work described here, that on control abstractions is much lower level, relying on a meta-language to capture the essential notions of closure and control dependency that can be directly encoded in our GPH-based system. We also avoid the complications caused by explicit encoding of synchronisation and communication, though perhaps at some cost in efficiency.

Crowl and Leblanc have applied the technique in a prototype parallelising compiler. They report good performance results compared with hand-coded parallel C, though certain optimisations must be applied by hand. This lends confidence to our belief that evaluation strategies could also be applied to imperative parallel programs.

Finally, there is a clear relationship between control abstraction and skeleton-based approaches. In fact, control abstractions could be seen as an efficient implementation technique for algorithmic skeletons.

6.4 Parallel Language Extensions

Rather than providing completely separate languages for coordination and computation, several researchers have instead extended a functional language with a small, but distinct, process control language. In its simplest form (as with GpH), this can be simply a set of annotations that specify process creation etc. More sophisticated systems, such as Caliban (Kelly, 1989), or first-class schedules (Mirani and Hudak, 1995) support normal functional expressions as part of the process control language.

Annotations Several languages have been defined to use parallel annotations. Depending on the approach taken, these annotations may be either hints that the runtime system can ignore, or directives that it *must* obey. In addition to specifying the parallelism and evaluation degree of the parallel program (the *what* and *how*), as for evaluation strategies, annotation-based approaches often also permit explicit placement annotations (the *where*).

An early annotation approach that is similar to that used in GpH was that of Burton (Burton, 1984), who defined three annotations to control the reduction order of function arguments: strict, lazy and parallel. In his thesis (Hughes, 1983), Hughes extends this set with a second strict annotation (`qes`), that reverses the conventional evaluation order of function and argument, evaluating the function body before the argument. Clearly all these annotations can be expressed as straightforward evaluation strategies, or even directly in GpH.

These simple beginnings have led to the construction of quite elaborate annotation schemes. One particularly rich set of annotations was defined for the Hope^{+} implementation on ICL's Flagship machine (Glynn *et al.*, 1988; Kewley and Glynn, 1989). This covered behavioural aspects such as data and process placement, as well as simple partitioning and sequencing. As a compromise between simplicity and expressibility, however, we will describe the well-known set of annotations that have been provided for Concurrent Clean (Nöcker *et al.*, 1991).

The basic Concurrent Clean annotation is `e {P} f args`, which sparks a task to evaluate `f args` to WHNF on some remote processor and continues execution of `e` locally. Before the task is exported its arguments, `args`, are reduced to NF. The equivalent strategy is `rnf args `seq` (rwhnf (f args) `par` e)`.

The other Concurrent Clean annotations differ from the `{P}` annotation in either the degree of evaluation or the placement of the parallel task. Since GpH delegates task placement to the runtime system, there is no direct strategic equivalent to the annotations that perform explicit placement.

Other important annotations are:

- `e {I} f args` interleaves execution of the two tasks on the local processor.
- `e {P AT` *location*`} f args` executes the new task on the processor specified by *location*.
- `e {Par} f args` evaluates `f args` to NF rather than WHNF. The equivalent strategy is `rnf args `seq` (rnf (f args) `par` e)`.
- `e {Self} f args` is the interleaved version of `{Par}`.

As with evaluation strategies, Concurrent Clean annotations cleanly separate dynamic behaviour and algorithm. However, because there is no language for composing annotations, the more sophisticated behaviours that can be captured by composing strategies cannot be described using Concurrent Cleanannotations. This is, in fact, a general problem with the annotation approach.

Caliban Caliban (Kelly, 1989) provides a separation of algorithm and parallelism that is similar to that used for evaluation strategies. The `moreover` construct is used to describe the parallel control component of a program, using higher-order functions to structure the process network. Unlike evaluation strategies, the `moreover` clause inhabits a distinct value space from the algorithm – in fact one which comprises essentially only values that can be resolved at compile-time to form a static *wiring system*. Caliban does not support dynamic process networks, or control strategies. A clean separation between algorithm and control is achieved by naming processes. These processes are the only values which can be manipulated by the `moreover` clause. This corresponds to the use of closures to capture computations in the evaluation strategy model.

For example, the following function defines a pipeline. The □ syntax is used to create an anonymous process which simply applies the function it labels to some argument. `arc` indicates a wiring connection between two processes. `chain` creates a chain of wiring connections between elements of a list. The result of the pipeline function for a concrete list of functions and some argument is thus the composition of all the functions in turn to the initial value. Moreover, each function application is created as a separate process.

```
pipeline fs x = result
  where    result = (foldr (.) id fs) x
  moreover (chain arc (map (□) fs))
           /\ (arc □(last fs) x)
           /\ (arc □(head fs) result)
```

Para-Functional Programming Para-functional programming (Hudak, 1986; Hudak, 1988; Hudak, 1991) extends functional programming with explicit parallel scheduling control clauses, which can be used to express quite sophisticated placement and evaluation schemes. These control clauses effectively form a separate language for process control. For ease of comparison with evaluation strategies, we follow Hudak's syntax for para-functional programming in Haskell (Hudak, 1991).

Hudak distinguishes two kinds of control construct: schedules are used to express sequential or parallel behaviours; while mapped expressions are used to specify process placements. These two notions are expressed by the `sched` and `on` constructs, respectively, which are attached directly to expressions.

Schedules In order to use functional expressions in schedules, Hudak introduces labelled expressions: `l@e` labels expression `e` with label `l` (this syntax is entirely equivalent to a *let* expression.

There are three primitive schedules: `D`*lab* is the demand for the labelled expression *lab*; `^`*lab* represents the start of evaluation for *lab*; and *lab*`^` represents the end of evaluation for *lab*. Whereas a value may be demanded many times, it can only be evaluated once. Schedules can be combined using either sequential composition (`.`) or parallel composition (`|`). Since it is such a common case, the schedule *lab* can be used as a shorthand for `D`*lab*`.`*lab*`^`. Schedules execute in parallel with the expression to which they are attached.

So, for example,

```
(l@e0 m@e1 n@e2) sched l^ . (Dm|Dn)
```

requires `e0` to complete evaluation before either `m` or `n` are demanded.

Evaluating schedules in parallel is one major difference from the evaluation strategy approach, where all evaluation is done under control of the strategy. A second major difference is that schedules are not normal functional values, and hence are not under control of the type system.

Mapped Expressions The second kind of para-functional construct is used to specify static or dynamic process placement. The expression *exp* `on` *pid* specifies that *exp* is to be executed on the processor identified by an integer *pid*. There is a special value `self`, which indicates the processor id of the current processor, and libraries can be constructed to build up virtual topologies such as meshes, trees etc. For example,

```
sort (QT q1 q2 q3 q4) =
        merge (sort q1 on (left  self))
              (sort q2 on (right self))
              (sort q3 on (up    self))
              (sort q4 on (down  self))
```

would sort each sub-quadtree on a different neighbouring processor, and merge the results on the current processor. Because GPH deliberately doesn't address the issue of thread placement, there is no equivalent to mapped expressions in evaluation strategies.

First-Class Schedules First-Class schedules (Mirani and Hudak, 1995) combine para-functional programming with a monadic approach. Where para-functional schedules and mapped expressions are separate language constructs, first-class schedules are fully integrated into Haskell. This integration allows schedules to be manipulated as normal Haskell monadic values.

The primitive schedule constructs and combining forms are similar to those provided by para-functional programming. The schedule $d\ e$ demands the value of expression e, returning immediately, while $r\ e$ suspends the current schedule until e has been evaluated. Both these constructs have type `a -> OS Sched`. Similarly, both the sequential and parallel composition operations have type `OS Sched -> OS Sched -> OS Sched`. The monadic type `OS` is used to indicate that schedules may interact in a side-effecting way with the operating system. As we will see, this causes loss of referential transparency in only one respect.

Rather than using a schedule construct, Mirani and Hudak instead provide a function `sched`, whose type is `sched :: a -> OS Sched -> a`, and which is equivalent to our `using` function. The `sched` function takes an expression e and a schedule s, and executes the schedule. If the schedule terminates, then the value of e is returned, otherwise the value of the `sched` application is $\perp$.

In evaluation strategy terms, both the d and r schedules can be replaced by calls to *rwhnf* without affecting the semantics of those para-functional programs that terminate. Unlike evaluation strategies, however, with first-class schedules it is also possible to suspend on a value without ever evaluating it. Thus para-functional schedules can give rise to deadlock in situations which cannot be expressed with evaluation strategies. A trivial example might be:

```
f x y = (x,y) `sched` r x . d y | r y . d x
```

Compared with evaluation strategies, it is not possible to take as much advantage of the type system: all schedules have type `OS Sched` rather than being parameterised on the type of the value(s) they are scheduling. Clearly there is also a loss of referential transparency, since expressions involving *sched* may sometimes evaluate to $\perp$, and other times to a non-$\perp$ value. If the program terminates (yields a non-$\perp$ value), however, it will always yield the same value.

6.5 Fully-Explicit Approaches

More explicit approaches usually work at the lowest level of parallel control, providing sets of basic parallelism primitives that could then be exploited to build more complex structures such as evaluation strategies. The approach is typified by MultiLisp (Halstead, 1985) or Mul-T (Kranz *et al.*, 1989) which provide explicit *futures* as the basic parallel control mechanism. Futures aresimilar to GPH `par`s.

At an even more explicit level, languages such as CML (Reppy, 1991) also require communication and synchronisation to be specified. Again, these constructs can be used to build a higher-level, evaluation strategy approach (closures and laziness can be modelled using function application or conditionals), although to our knowledge, there has been no attempt yet to implement such an approach in this framework.

At a slightly higher level, Jones and Hudak have worked on commutative Monads (Jones and Hudak, 1993), which allow operations such as process creation (called `fork`) to be captured within a standard state-transforming monad. While this approach provides the essential building blocks which would be needed to support evaluation strategies, it has the disadvantage of raising all parallel operations to the monad level, thus preventing the clean separation of algorithm and behaviour that is observed with either evaluation strategies or first-class schedules.

7 Conclusion

7.1 Summary

This paper has introduced evaluation strategies, a new mechanism for controlling the parallel evaluation of non-strict functional languages. We have shown how lazy evaluation can be exploited to define evaluation strategies in a way that cleanly separates algorithmic and behavioural concerns. As we have demonstrated, the result is a very general, and expressive system: many common parallel programming paradigms can be captured. Finally, we have also outlined the use of strategies in three large parallel applications, noting how they facilitate the top-down parallelisation of existing code.

7.2 Discussion

Required Language Support In describing evaluation strategies, we have exploited several aspects of the Haskell language design. Some of these are essential, whereas others may perhaps be modelled using other mechanisms. For example, some support for higher-order functions is clearly needed: strategies are themselves higher-order functions, and may take functional arguments.

Lazy evaluation of some form is clearly essential since it allows us to postpone to the strategy the specification of which bindings, or data-structure components, are evaluated and in what order. Operationally, laziness avoids the recomputation of values referred to in both the algorithmic code and the strategy. Although we have not yet studied this in detail, the work on control abstraction by Crowl and Leblanc, plus other work referred to above, does suggest that enough of the characteristics of lazy evaluation could be captured in an imperative language to allow the use of evaluation strategies in a wider context than that we have considered.

In defining evaluation strategies, we have taken advantage of Haskell's type class overloading to define general evaluation-degree strategies, such as `rnf`. If general ad-hoc overloading is not available, then a number of standard alternative approaches could be taken, including:

- define a set of standard polymorphic evaluation-degree operations;
- require evaluation-degree operations to be monomorphic.

In either case, support can be provided as functions or language constructs. Neither approach is as desirable as that taken here, since they limit user flexibility in the first case, or require code duplication in the second.

Additional Control Issues Evaluation strategies have been used to specify some aspects of dynamic behaviour that are not described here. One such aspect is control of thread granularity. While it is not possible to exploit load information, for example, in a referentially transparent fashion, simple *thresholding* techniques can safely be employed. In quicksort, for example, if the sublists that are to be sorted are sufficiently small they can be evaluated sequentially rather than subdivided for parallel execution. Such tests are easily incorporated into an evaluation strategy, which consquently avoids cluttering the algorithmic code.

One parallel programming paradigm that we have not expressed here is branch-and-bound parallelism. This cannot be expressed functionally, however, without using semantic non-determinism of some kind. This is not available in Haskell, though languages such as Sisal (Feo *et al.*, 1995) do provide non-determinism for precisely such a purpose.

Abuse of Strategies Like most powerful language constructs, evaluation strategies can be abused. If a strategy has an evaluation degree greater than the strictness of the function it controls, it may change the termination properties of the program (note that unlike first-class schedules, however, this is still defined by the normal language semantics). Similarly it is easy to construct strategies with undesirable parallelism, e.g. a strategy that creates an unbounded number of threads. Finally, strategies sometimes require additional runtime traversals of a data structure. In pathological cases, e.g. when accumulating parameters are involved, care must be taken to avoid multiple traversals.

7.3 Future Work

The groups at Glasgow and Durham will continue to use evaluation strategies to write large parallel programs, and we hope to encourage others to use them too.

Initial performance measurements show that strategic code is as efficient as code with *ad hoc* parallelism and forcing functions, but more measurements are needed to confirm that this is true in general.

A framework for reasoning about strategic functions is under development. Proving that two strategic functions are equivalent entails not only proving that they compute the same value, but also that they have the same evaluation degree and parallelism/sequencing. The evaluation-degree of a strategic function can be determined adding laws for `par` and `seq` to existing strictness analysis machinery, e.g. Hughes and Wadler's projection-based analysis (Wadler and Hughes, 1987). As an operational aspect, parallelism/sequencing are harder to reason about. At present we have a set of laws, e.g. both `par` and `seq` are idempotent, but are uncertain of the best framework for proving them. One possible starting point is to use partially order multisets to provide a theoretical basis for defining evaluation order (Hudak and Anderson, 1987).

Some support for evaluation strategies could be incorporated into the language. If the compiler was able to automatically derive `rnf` from a type definition, the work involved in parallelising a large application would be dramatically reduced, and the replication of libraries could be avoided. Some form of tagging of closures in the runtime system could reduce the execution overhead of strategies: a data structure need not be traversed by a strategy if its evaluation degree is already at least as great as the strategies.

We would like to investigate strategies for strict parallel languages. Many strict functional languages provide a mechanism for postponing evaluation, e.g. `delay` and `force` functions. The question is whether cost of introducing explicit laziness outweighs the benefits gained by using strategies.

Our long term goal is to support more implicit parallelism. Strategies provide a useful step towards this goal. We are learning a great deal by explicitly controlling dynamic behaviour, and hope to learn sufficient to automatically generate strategies with good dynamic behaviour for a large class of programs. One promising approach is to use strictness analysis to indicate when it is safe to evaluate an expression in parallel, and granularity analysis to indicate when it is worthwhile. It may be possible to use a combined implicit/explicit approach, i.e. most of a program may be adequately parallelised by a compiler, but the programmer may have to parallelise a small number of crucial components.

References

Arvind, Nikhil, R.S., and Pingali, K.K., "I-Structures - Data Structures For Parallel Computing", *TOPLAS* **11(4)**, (1989), pp. 598–632.

Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Spielstein, J., and Zagha, M., "Implementation of a Portable Nested Data-Parallel Language", *Proc. Fourth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 19-22, (1993), pp. 102–111.

Blelloch, G.E., "Programming Parallel Algorithms", *CACM*, **39(3)** (1996), pp. 85–97.

Botorog, G.M., and Kuchen, H., "Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Computation", *Proc. 5th. IEEE Intl. Symposium on High Performance Distributed Computing*, Syracuse, NY, August 6-9, (1996), pp. 253–252.

Burn, G.L., *Abstract Interpretation and the Parallel Evaluation of Functional Languages*, PhD Thesis, Imperial College London, (1987).

Burn, G.L., "Implementing the Evaluation Transformer Model of Reduction on Parallel Machines", *J. Functional Prog.*, **1(3)**, (1991), pp. 329–366.

Burton, F.W., "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs", *ACM TOPLAS*, **6(2)**, April (1984), pp. 159–174.

M.I. Cole, *Algorithmic Skeletons*, Pitman/MIT Press (1988).

Crowl, L.A, and Leblanc, T.J., "Parallel Programming with Control Abstraction", *ACM TOPLAS*, **16(3)**, (1994), pp. 524–576.

Danelutto, M., Di Meglio, R., Orlando, S., Pelagatti, S., and Vanneschi, M., "The P^3L Language: An Introduction", Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Centre, December, (1991).

Darlington, J., Guo, Y., To, H.W., and Yang, J., "Parallel Skeletons for Structured Composition", *Proc. Fifth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 19-21, (1995), pp. 19–28.

Date, C.J., *An Introduction to Database Systems*, 4th Edition, Addison Wesley, (1976).

Feo, J., Miller, P., Skedziewlewski, S., Denton, S., and Soloman, C., "Sisal 90", *Proc. HPFC '95*, Denver, CO, April 9-11, (1995), pp. 35–47.

Finne, S.O., and Burn, G.L., "Assessing the Evaluation Transformer Model of Reduction on the Spineless G-Machine", *Proc. FPCA '93*, Copenhagen, (1993), pp. 331–340.

Gelernter, D., and Carriero, N., "Coordination Languages and Their Significance", *CACM*, **32(2)**, February, (1992), pp. 97–107.

Flanagan, C., and Nikhil, R.S., "pHluid: The Design of a Parallel Functional Language Implementation", *Proc. ICFP '96*, Philadelphia, Penn., May 24-26, (1996), pp. 169–179.

Foster, I., and Taylor, S., "A Compiler Approach to Scalable Concurrent-Program Design", *ACM TOPLAS*, **16(3)**, (1994), pp. 577–604.

Glynn, K., Kewley, J.M., Watson, P., and While, L., "Annotations for $Hope^+$", Technical Report IC/FPR/PROG/1.1.1/5, Imperial College, London, (1988).

Halstead, R., "MultiLisp: A Language for Concurrent Symbolic Computation", *ACM TOPLAS*, **7(4)**, (1985), pp. 501–538.

Hammond, K., Loidl, H.-W., and Partridge, A.S., "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell", *Proc. HPFC'95 — High Performance Functional Computing*, Denver, CO, April 9-11, (1995), pp. 208–221.

Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall (1985).

Hudak, P., "Para-Functional Programming", *IEEE Computer*, **19(8)**, (1986), pp. 60–71.

Hudak, P., "Exploring Para-Functional Programming: Separating the what from the how", *IEEE Software*, **5(1)**, (1988), pp. 54–61.

Hudak, P., "Para-Functional Programming in Haskell", In *Parallel Functional Languages and Computing*, ACM Press (New York) and Addison-Wesley (Reading, MA), (1991), pp. 159–196.

Hudak, P., and Anderson, S., "Pomset Interpretations of Parallel Functional Languages", *Proc. FPCA '87*, Springer-Verlag LNCS 274, September (1987), pp. 234–256.

Hughes, R.J.M., *The Design and Implementation of Programming Languages*, DPhil Thesis, Oxford University, (1983).

Jones M.P., and Hudak, P., "Implicit and Explicit Parallel Programming in Haskell", Research Report YALEU/DCS/RR-982, University of Yale, August 13, (1993).

Kelly, P.H.J., *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman/MIT Press, (1989).

Kewley, J.M., and Glynn, K., "Evaluation Annotations for Hope$^+$", *Glasgow Workshop on Functional Programming*, Fraserburgh, Scotland, Springer-Verlag WICS, (1989), pp. 329–337.

Kranz, D., Halstead, R., and Mohr, E., "Mul-T: A High-Performance Parallel Lisp", *Proc. PLDI '89*, Portland, OR, June, (1989), pp. 81–90.

M. Lauer, "Computing by Homomorphic Images", in *Computer Algebra — Symbolic and Algebraic Computation*, B. Buchberger, G.E. Collins, R. Loos, and R. Albrecht, (Eds.), Springer Verlag (1982), pp. 139–168.

Loidl, H.-W., Hammond, K., and Partridge A.S., "Solving Systems of Linear Equations Functionally: a Case Study in Parallelisation", Technical Report, Dept. of Computing Science, University of Glasgow, (1995).

Milner, A.J.R.G., *Communication and Concurrency*, Prentice Hall (1989).

Mirani, R., and Hudak, P., "First-Class Schedules and Virtual Maps", *Proc. FPCA '95*, La Jolla, CA, June, (1995), pp. 78–85.

Mohr, E., Kranz, D.A., and Halstead, R.H., "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems*, **2(3)**, July, (1991), pp. 264–280.

Morgan, R.G., Smith, M.H., and Short, S., "Translation by Meaning and Style in Lolita", *Intl. BCS Conf. — Machine Translation Ten Years On*, Cranfield University, November, (1994).

Nikhil, R.S., Arvind and Hicks, J., "pH language proposal", DEC Cambridge Research Lab Tech. Rep. (1993).

Nöcker, E.G.J.M.H., Smetsers, J.E.W., van Eekelen, M.C.J.D., and Plasmeijer, M.J., "Concurrent Clean", *Proc. PARLE '91*, Springer Verlag LNCS 505/506, (1991), pp. 202–220.

Peterson J.C., Hammond, K. (eds.), Augustsson, L., Boutel, B., Burton, F.W., Fasel, J., Gordon, A.D., Hughes, R.J.M., Hudak, P., Johnsson, T., Jones, M.P., Peyton Jones, S.L., Reid, A., and Wadler, P.L., *Report on the Non-Strict Functional Language, Haskell, Version 1.3*, (1996).

Rabhi, F.A. "Exploiting Parallelism in Functional Languages: a 'Paradigm-Oriented' Approach", in *Abstract Machine Models for Highly Parallel Computers*, Dew, P. and Lake, T. (eds.), Oxford University Press, (1993).

Reppy, J.H., "CML: a Higher-Order Concurrent Language", *Proc. PLDI '91*, Toronto, Canada, June 26-28, (1991), pp. 293–305.

Roe, P., *Parallel Programming using Functional Languages*, PhD thesis, Dept. of Computing Science, University of Glasgow, April, (1991).

Sansom, P.M., and Peyton Jones, S.L., "Time and Space Profiling for Non-Strict, Higher-Order Functional Languages", *Proc. POPL '95*, (1995), pp. 355–366.

Subhlok, J., Stichnooch, J.M., O'Hallaron, D.R., and Gross, T., "Exploiting Task and Data Parallelism on a Multicomputer", *Proc. Fourth ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, San Diego, CA, May 19-22, (1993), pp. 13–22.

Trinder, P.W., Hammond, K., Mattson, J.S. Jr., Partridge, A.S., and Peyton Jones, S.L., "GUM: a Portable Parallel Implementation of Haskell", *Proc. PLDI '96*, Philadelphia, Penn., May 22-24, (1996), pp. 79–88.

Trinder, P.W., Hammond, K., Loidl, H-W., Peyton Jones, S.L., and J. Wu, "A Case Study of Data-intensive Programs in Parallel Haskell" *Proc. Glasgow Functional Programming Workshop*, Ullapool, Scotland, (1996).

Wadler, P.L., and Hughes, R.J.M., "Projections for Strictness Analysis", *Proc. FPCA '87*, September, (1987).

A Determinant

This appendix contains two more versions of the determinant function from the linear equation solver described in Section 4.2. The version on the left is the original sequential version. That on the right is a slightly cleaned-up version of the one we originally wrote to parallelise this function. Compared with the strategic version presented earlier, the lower-level parallel version is much more obscure and difficult to understand.

Sequential Version

```
sum l_par where
  l_par = map determine1 [jLo..jHi]
  determine1 j =
   (if pivot > 0 then
     sign*pivot*det'
    else
     0)
   where
     sign = if (even (j-jLo))
            then 1 else -1
     pivot = (head mat) !! (j-1)
     mat' =
       SqMatrixC
            ((iLo,jLo),(iHi-1,jHi-1))
            (map (newLine j)
                 (tail mat))
     det' = determinant mat'
```

Direct Parallel Version

```
sum l_par where
  l_par =  do_it_from_to jLo
  do_it_from_to j
    | j>jHi =   []
    | otherwise = fx `par` (fx:rest)
    where
      sign = if (even (j-jLo))
             then 1 else -1
      mat' =
        SqMatrixC
           ((iLo,jLo),(iHi-1,jHi-1))
           (parMap (newLine j)
                   (tail mat))
      pivot = (head mat) !! (j-1)
      det' = mat' `seq`
             determinant mat'
      x = case pivot of
          0 -> 0
          _ -> sign*pivot*det'
      fx = sign `par`
           if pivot>0
           then det' `par` x else x
      rest = do_it_from_to (j+1)
```