J. Functional Programming **8** (1): 23–60, January 1998. Printed in the United Kingdom
© 1998 Cambridge University Press

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 parallelism, 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. Some of these applications are described in detail here. The largest application we have studied to date, Lolita, is a 40,000 line natural language engineering system. Initial results show that for these programs we can achieve acceptable parallel performance, for relatively little programming effort.

Capsule Review

This paper advocates that control of parallelism and evaluation order should be largely separated from an underlying program. The control depends heavily on the underlying program, but the underlying program can be examined largely without regard to the control. The results are presented in the context of a parallel version of Haskell, GpH, extended to include two primitive operations for controlling parallelism and evaluation order, `par` and `seq` (where `seq` is included as standard in Haskell 1.3). The authors have extensive experience using GpH to write real parallel programs.

Based on their experience, the authors advocate using *evaluation strategies* to control program behaviour. Surprisingly, the introduction of strategies requires no changes to the underlying language. A strategy is a function of type `a -> ()`. An infix operator `using` is defined so `` exp `using` strategy = strategy exp `seq` exp ``. That is, the strategy is

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

applied to an expression to force a certain evaluation pattern. The result is discarded and the original expression is returned.

The authors present a number of examples drawn from real problems and illustrating a variety of different forms of parallelism to demonstrate the practical value of their approach.

1 Writing parallel programs

While it is hard to write good sequential programs, it can be considerably harder to write good parallel ones. At the University of 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 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 the 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 dynamic-behaviour code.

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 values defined in the former being 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. Changing the algorithm may entail specifying new dynamic behaviour; conversely, it is easy to modify the strategy without changing the algorithm.

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 40,000 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

GpH is available free with the Glasgow Haskell compiler and is supported by GUM, a robust, portable runtime system (Trinder *et al.*, 1996). GUM is message-based, and portability is facilitated by using the PVM communications harness that is available on many multi-processors. As a result, GUM is available for both shared-memory (Sun SPARCserver multi-processors) and distributed-memory (networks of workstations, and CM5) architectures. The high message-latency of distributed machines is ameliorated by sending messages asynchronously, and by sending large packets of related data in each message. GUM delivers wall-clock speedups relative to the best sequential compiler technology for real programs (Trinder *et al.*, 1996). Most of the example programs below are run on shared-memory architectures.

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`, and is not strict in its first argument, i.e. $\perp$ `'par' e` has the value of `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*, and a thread may subsequently be created to evaluate it if a processor becomes idle. There is no global priority ordering between sparks on different processors, although the sparks on a single processor are scheduled in first-in first-out (FIFO) order. 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 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 is taken in the compiler to protect them. A more detailed description of the implementation of `par` and `seq` is given 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.

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

If `n` is greater than 1, then `pfib (n-1)` is sparked, and the thread continues to evaluate `pfib (n-2)`. Fig. 1 shows a process diagram of the execution of `pfib 15`. Each node in the diagram is a function application, and each arc carries the data value, in this case an integer, used to communicate between the invocations. Brackets can safely be omitted because `seq` has a higher precedence than `par`.

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

```
quicksortN :: (Ord a) => [a] -> [a]
quicksortN []      = []
quicksortN [x]     = [x]
```

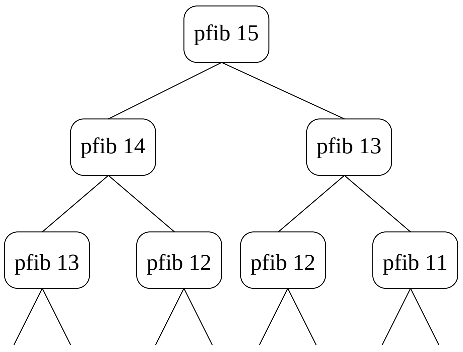

Fig. 1. pfib Divide-and-conquer process diagram.

```
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 in 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, such as `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

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-*

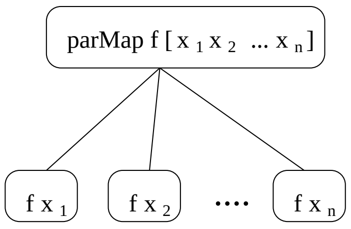

Fig. 2. parMap process diagram.

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 Fig. 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 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 several aspects of 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.
- *Thread granularity*: it is important to spark only those expressions where the cost of evaluation greatly exceeds the thread creation overheads.
- *Locality*: part of the cost of evaluating a thread is the time required to communicate its result and the data it requires, and in consequence it may only be worth creating a thread if its data is local.

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*, i.e. the evaluation-degree may usefully be more strict than the lazy function. Section 5.5 contains a case study program where a strategy is used to introduce speculative parallelism.

3 Strategies separate algorithm from dynamic behaviour

3.1 Evaluation strategies

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. This section describes *evaluation strategies*, our solution to this dilemma. The driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour*.

An *evaluation strategy* is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

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

3.2 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. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element 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` ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains *no* redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or datatypes, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

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

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

```
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
```

3.3 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. Elements of a strategy are combined by sequential or parallel composition (`seq` or `par`). Many useful strategies are higher-order, for example, `seqList` below 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 datatype, indeed in later versions of Haskell (1.3 and later (Peterson *et al.*, 1996)) constructor classes can be defined that work on arbitrary datatypes. The strategic examples in this paper are presented in Haskell 1.2 for pragmatic reasons: they are extracted from programs run on our efficient parallel implementation of Haskell 1.2 (Trinder *et al.*, 1996).

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

3.4 Data-oriented parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism 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)
```

Data-oriented strategies are applied by the `using` function which applies the strategy to the data structure `x` before returning it. 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 because it acts as a separator between algorithmic and behavioural code.

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

A strategic version of the parallel map encountered in Section 2.2 can be written as follows. Note how the algorithmic code `map f xs` is cleanly separated from the strategy. 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), and this relationship is discussed further in section 6.2.

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

3.5 Control-oriented parallelism

Control-oriented parallelism is typically expressed by a sequence of strategy applications composed with `par` and `seq` that specifies which subexpressions of a function are to be evaluated in parallel, and in what order. The sequence is loosely termed a strategy, and is invoked by either the `demanding` or the `sparking` function. The Haskell `flip` function simply reorders a binary function's parameters.

```
demanding, sparking :: a -> () -> a

demanding = flip seq
sparking  = flip par
```

The control-oriented parallelism of `pfib` can be expressed as follows using `demanding`. Sections 4.4 and 4.2 contain examples using `sparking`

```
pfib n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) ‘demanding‘ strategy
    where
      n1 = pfib (n-1)
      n2 = pfib (n-2)
      strategy = rnf n1 ‘par‘ rnf n2
```

If we wish to avoid explicitly naming the result of a function, it is sometimes convenient to apply a control-oriented strategy with `using`. Quicksort is one example, and 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
```

3.6 Additional dynamic behaviour

Strategies can control other aspects of dynamic behaviour, thereby avoiding cluttering the algorithmic code with them. A simple example is a thresholding mechanism that controls thread granularity. In `pfib` for example, granularity is improved for many machines if threads are not created when the argument is small. More sophisticated applications of thresholding are discussed in sections 5.3 and 6.2.

```
pfibT n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfibT (n-1)
      n2 = pfibT (n-2)
      strategy = if n > 10
                   then rnf n1 `par` rnf n2
                   else ()
```

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. 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. Non-determinism is not available in Haskell, though languages such as Sisal (McGraw, 1985) provide it for precisely such a purpose, and Burton and Jackson have shown how to encapsulate the nondeterminacy in an abstract data type with deterministic semantics (Burton, 1991), and discussed a parallel implementation (Jackson and Burton, 1993)

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`. A classic example is drawn from the manufacturing application domain, and is based on 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
P6	P7	8
P2	P5	3

A naïve function `explode` lists the components of a single part, `main`. For example, the result of exploding `P1` in the relation above is `[P2, P4, P5, P3]`. The core of the query is the function `explosions` which explodes a sequence of parts.

```
type PartId = Int
type BillofMaterial = [(PartId, PartId, Int)]

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

explosions :: PartId -> PartId -> BillofMaterial -> [[PartId]]
explosions lo hi bom =
  map (explode bom) [lo..hi]
```

The `explosions` function is inherently data parallel because the explosion of one part is not dependent on the explosion of any other. On the target Sun SPARCserver architecture, an appropriate thread granularity is to compute each explosion in parallel, but without parallelism within an explosion. This dynamic behaviour is specified by adding the following evaluation strategy which operates on the resulting list of lists. The `seqList rwhnf` forces all of the explosion to be computed by each thread. Subsequent sections include data-oriented strategies defined over many types including pairs, triples and square matrices.

```
explosions lo hi bom =
  map (explode bom) [lo..hi] ‘using‘ parList (seqList rwhnf)
```

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.

The computation performed in the solve-stage of the computation is essentially a determinant computation which can be specified as follows:

- Given: a matrix $(A_{i,j})_{1 \leq i,j \leq n}$
- 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 `demanding` parList rnf l_par
where
  l_par = map determine1 [jLo..jHi]
  determine1 j = (if pivot > 0 then
                    sign*pivot*det' `demanding` strategyD
                  else
                    0) `sparking` rnf 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 =
                       parSqMatrix (parList rwhnf) mat' `seq`
                       (rnf det' `par` ())
```

In this example almost all available parallelism is exploited, and 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 (A) of size n is computed in terms of the determinants of n matrices (A') of size $n - 1$.

The first strategy, `parList rnf l_par` 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, `` `sparking` rnf sign`` specifies that the sign of the determinant should be calculated in parallel with the conditional. 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. The `strategyD` is invoked with `demanding` to ensure that it is evaluated, if `sparking` had been used, the final `` `par` ()`` could be omitted, but the strategy might never be executed. Note that some data-oriented strategies such as `parList` and `parSqMatrix` are used within the overall control-oriented structure.

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 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 a bounded or n-place buffer. There are many other possible ways to express producer/consumer parallelism, for example, to improve granularity the producer could compute the next n-element "chunk" of the list rather than just a single value.

The dynamic behaviour of an n-place list buffer is as follows. Initially, the first n

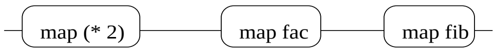

Fig. 3. Pipeline process diagram.

list elements are eagerly constructed, and then, whenever the head of the buffer-list is demanded, the *n*th element is sparked. In effect the producer speculatively assumes that the next *n* elements in the list will be used in the computation. This assumption introduces parallelism because, if there is a free processor, a thread can produce the *n*th element, while the consumer is consuming the first.

Applying the following `parBuffer n s` function to any list converts it into an `n`-place buffer that applies strategy `s` to each list element. Initially `start` sparks the first `n` elements and returns a (shared) list from the `n`th element onwards. The value of the `return` function is identity on it's first argument, and it's dynamic behaviour is to spark the `nth` element (the head of the `start` list) whenever the head of the list is demanded.

```
parBuffer :: Int -> Strategy a -> [a] -> [a]
parBuffer n s xs =
  return xs (start n xs)
  where
    return (x:xs) (y:ys) = (x:return xs ys) ‘sparking‘ s y
    return xs     []     = xs

    start n []     = []
    start 0 ys     = ys
    start n (y:ys) = start (n-1) ys ‘sparking‘ s y
```

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. This kind of parallelism is easily expressed in a non-strict language by function composition. The non-strict semantics ensures that no barrier synchronisation is required between the different stages.

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 s inp []     = inp
pipeline s inp (f:fs) =
  pipeline s out fs ‘sparking‘ s 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. Fig. 3 gives the process diagram for the example above.

Some 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 ‘demanding‘ 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 = (parPair rwhnf (parList rwhnf)) inp                ‘seq‘
             (parPair rwhnf (parList (parPair rwhnf rwhnf))) r1  ‘seq‘
             rnf r2                                              ‘par‘
             rnf r3                                              ‘par‘
             rnf r4                                              ‘par‘
             rnf r5                                              ‘par‘
             rnf r6                                              ‘par‘
             (parTriple rwhnf (parList rwhnf) rwhnf) r7          ‘seq‘
               ()
```

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 introduce two strategic combinators to express sequential and parallel function application. Explicit function application is written $, and `f $ x = f x`. The new combinators take an additional strategic parameter that specifies the strategy to be applied to the argument, and hence textually separate the algorithmic and behavioural code.

The definition of the new combinators is as follows:

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

($|) f s x  = f x ‘demanding‘ s x
($||) f s x = f x ‘sparking‘  s x
```

We have also defined similar combinators for strategic function composition, which can be viewed as a basic pipeline combinator. Pipelines can now be expressed more concisely, for example the pipeline above becomes:

```
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

Using evaluation strategies, we have written a number of medium-scale parallel programs, and are currently parallelising a large-scale program, Lolita (60,000 lines). This section discusses the use of strategies in three programs, one divide-and-conquer, one pipelined and the third data-oriented. The methodology we are developing out of our experiences is also described. We have built on experience gained with several realistic parallel programs in the FLARE project (Runciman, 1995). The FLARE project was the first attempt to write large parallel programs in Haskell, and predated the development of strategies.

Until recently parallel programming was 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, and understanding and explicitly controlling it is hard. A major motivation for investigating our predominantly-implicit approach is to address irregular parallelism. Two recent parallel programming models, Bulk Synchronous Processing (BSP) (McColl, 1996) and Single-Program Multiple-Data (SPMD) (Smirni *et al*., 1995), have gone some way towards addressing this problem by providing a framework in which some irregularity can be supported in an otherwise regular program.

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 an `NFData` instance for datatypes would greatly reduce the amount of code to be modified and avoid this problem of reproducing libraries. As an interim measure we have developed a tool that, *inter alia*, automatically derives an `NFData` instance for any datatype.

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. **Idealised Simulation.** Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a 'proving' step: if the program isn't parallel on an idealised machine it won't be on a real machine. We now use GranSim, but have previously used `hbcpp`. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a workstation.
6. **Realistic Simulation.** 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 artifacts of the communication system or other dynamic characteristics. In its overall structure our methodology is similar to others used for large-scale parallel functional programming (Hartel *et al.*, 1995).

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

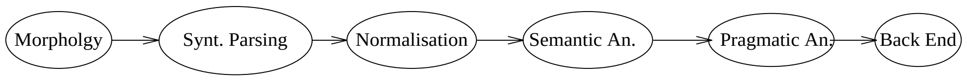

Fig. 4. Overall pipeline structure of Lolita.

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 (compositional analysis of meaning);
- Pragmatic analysis (using contextual information from previous sentences).

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 may 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 Sun SPARCserver. A pipeline approach is a promising way to achieve this relatively small degree of parallelism (Figure 4). 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 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 of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures.

In addition to the pipeline parallelism, we introduce parallelism in the syntactic parsing stage. The parallelism in this module has the overall structure of a parallel tree traversal. In order to improve the granularity in this stage we apply a threshholding strategy (similar to the one at the end of section 3.1) to a system parameter, which reflects the depth in the tree. In fact the same polymorphic threshholding strategy is applied to two lists of different types.

Another source of parallelism can be used to improve the quality of the analysis by applying the semantic and pragmatic analyses in a data-parallel fashion on different possible parse trees for the same sentence. Because of the complexity of these analyses the sequential system always picks the first parse tree, which may cause the analysis to fail, although it would succeed for a different parse tree. We have included code to exploit this kind of parallelism but not yet tested its influence on the quality of the result.

Fig. 5 shows the parallel structure arising when all of the sources of parallelism described above are used. Note that the analyses also produce information that is put into a ‘global context’ containing information about the semantics of the text. This creates an additional dependence between different instances of the analysis (indicated as vertical arcs). Lazy evaluation ensures that this does not completely sequentialise the analyses, however.

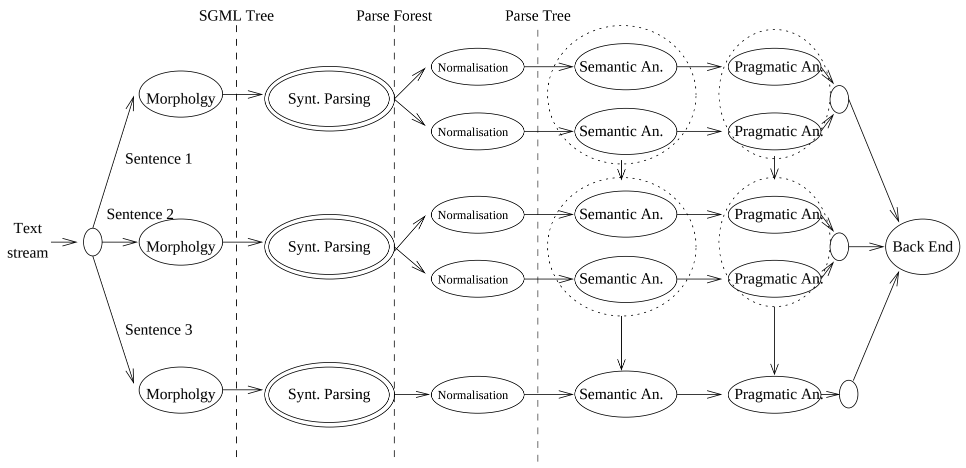

Fig. 5. Detailed structure of Lolita.

The code of the top level function `wholeTextAnalysis` in Fig. 6 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;
2. the `evalScores` strategy which defines data parallelism in the analysis stages over possible parse trees; and
3. the use of strategic 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 (specified in `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. Since Lolita makes heavy use of laziness it is very important that the strategies are not too strict. Otherwise, redundant computations are performed, which yield no further improvements in runtime.

It should be emphasised that specifying the strategies that describe this parallel behaviour entailed understanding and modifying only one of about three hundred modules in Lolita and three of the thirty six functions in that module. So far, the only module we have parallelised is the syntactic parsing stage. If it proves necessary to expose more parallelism we could parallelise other sub-algorithms, which also contain significant sources of parallelism. In fact, the most tedious part of the code changes was adding instances of `NFData` for intermediate data structures, which are spread over several dozen modules. However, this process has been partially automated, as described in section 5.1

```
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
   evalAnalysis = stateMap_TimeOut analyseSemPrag pt global
   evalScores =   parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))

-- 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. 6. The top level function of the Lolita application.

We are currently tuning the performance of Lolita on the Sun SPARCserver. A realistic simulation showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. Since Lolita was originally written without any consideration for parallel execution and contains a sequential front end (written in C) of about 10–15%, we are pleased with this amount of parallelism. In particular the gain for a set of rather small changes is quite remarkable.

The wall-clock speedups obtained to date are disappointing. With two processors and small inputs we obtain an average parallelism of 1.4. With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance. The reason for this is that GUM, which is designed to support distributed-memory architectures uniformly, loads a copy of the entire code, and a separate local heap, onto each processor. Lolita is a very large program, incorporating large static data segments (totaling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation. Clearly, a great deal of this data could be shared, an opportunity we are exploring. We hope to obtain better results soon using a machine with twice the memory capacity. We are also making the C parsing functions re-entrant which will allow the analysis to be performed in a data-parallel fashion over a set of input sentences.

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 Fig. 7.

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.

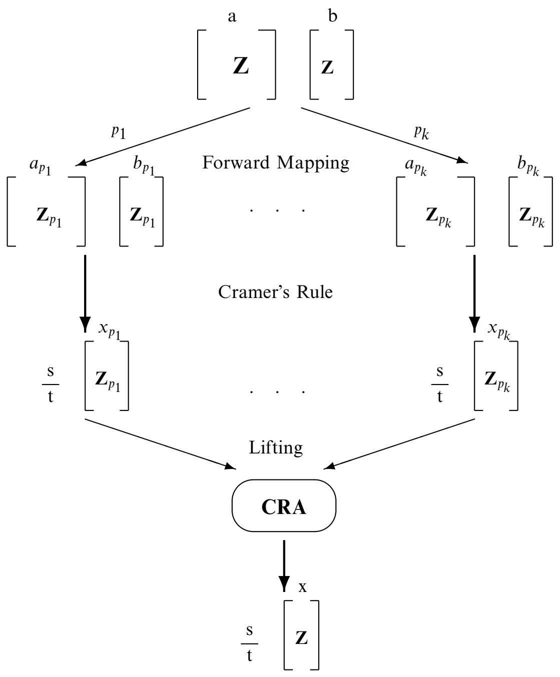

Fig. 7. Structure of the LinSolv algorithm.

5.5 Accident blackspots

The UK Centre for Transport Studies requires to analyse police accident records to discover 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 algorithm used is as follows. For each of the matching criteria an index is constructed over the set of accidents. The indices are used to construct an indexed, binary same-site relation that pairs accidents occurring at the same location. The partition is obtained by repeatedly choosing an accident and finding all of the accidents reachable from it in the same-site relation (Trinder *et al.*, 1996).

Fine-grained version The first parallel version uses fine-grained parallelism, and has four stages in the top-level pipeline: reading and parsing the file of accidents; constructing the criteria indices over the set of accidents; constructing the indexed same-site relation; and forming the partition. Little parallelism is gained from this top-level pipeline (a speedup of 1.2) because partitioning depends on the same-site index, and constructing the same-site relation depends on the criteria indices and the first value cannot be read from an index (or tree) until all of the index has been constructed.

The individual pipeline stages are parallelised using a variety of techniques. The file reading and parsing stage is made data parallel by partitioning the data and reading from *n* files.

```
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 = concat (map parse8Tuple cts `using` parList rnf)
  in ...
```

Control parallelism is used to construct the three criteria indices.

```
mkAccidentIxs :: [Accident] -> AccidentIxs
mkAccidentIxs accs = (jIx,neIx,rpIx) `demanding` strategy
  where
    jIx  = ...
    neIx = ...
    rpIx = ...
    strategy = rnf jIx  `par`
               rnf neIx `par`
               rnf rpIx `par` ()
```

The pipeline stages constructing the same-site relation and the partition both use benign speculative parallelism. For partitioning, the equivalence classes of *n*, 20 say, accidents are computed 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 7,500 accidents. 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)
                 `demanding` strategy
    otherwise -> ...
    where
      aList = take n (setToList accs)
      matchList = [mkSet (reachable [a] ixRel) | a <- aList]
      rest     = minusManySet accs matchList
      strategy = parList rnf matchList
```

On four processors the fine-grained program achieves an average parallelism of 3.5 in an idealised simulation. Unfortunately, average parallelism falls to 2.3 for the simulated target machine because thread granularity is small and data locality is poor.

Coarse-grained version The second version of the program partitions the data geographically into a number of tiles using the grid references. Each tile has an overlap with its neighbours to capture multiple-accident sites that span the borders. Each area is partitioned in parallel and duplicated border sites are eliminated. There are currently just four tiles: top left, top right, bottom left and bottom right; and the strategy is trivial:

```
strategy =
  rnf tlPartition ‘par‘
  rnf trPartition ‘par‘
  rnf blPartition ‘par‘
  rnf brPartition
```

The advantages of this simple, coarse-grained approach are excellent thread granularity and data locality. On four processors an average parallelism of 3.7 is achieved for both idealised and realistic simulations. The program is at an early stage of tuning on a shared-memory Sun SPARCserver with 4 Sparc 10 processors, and is already delivering wall-clock speedups of 2.2 over the sequential version compiled with full optimisation. The sequential Haskell version is already an order of magnitude faster than the interpreted PFL version constructed at the Centre for Transport Studies (Trinder *et al.*, 1996). Evaluation strategies facilitated experiments with many different types of parallelism in this application.

6 Related work

Many different mechanisms have been proposed to specify the 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; Flanagan and Nikhil, 1996), 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.

Table 1. *The relationship of evaluation strategies and transformers*

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`

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 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 or to allow speculation; to use a strategy that is known to be safe, though stricter than the analyser can detect; or to use a strategy that is less strict than the analyser can determine, in order to improve granularity. Finally, it is not always straightforward to determine how much strictness an analyser might detect, and small program changes may have dramatic effects on strictness information.

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 some strategies. 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-2. Certain evaluation strategies can therefore be seen as control parallel implementations of data parallel constructs, targeted more at distributed-memory or shared-memory machines than at massively parallel architectures.

Unusually for a data parallel language, NESL supports *nested* parallelism. This allows more complex and more irregular computation patterns to be expressed than with traditional data-parallel languages such as C* (Rose, *et al.*, 1987). For example, to compute the product of a sequence using a divide-and-conquer style algorithm, the following code could be used:

```
function product(a) =
  if (#a == 1) then a[0]
  else let r = {product(v) : v in bottop(a)};
       in r[0] * r[1]
```

The RHS of `r` is a sequence comprising two calls to `product`. The `bottop` function is used to split the argument sequence, `a`, into two equal sized components.

This could then be used in an outer sequence if required, for example,

```
function products(m,n) =
   {product(s) : s in {[p:n] : p in [m:n-1]}}
```

NESL has been implemented on a variety of machines including the CM-2, the Cray Y-MP and the Encore Multimax.

Dataflow Many recent dataflow languages are functional, e.g. Id (Arvind *et al.*, 1989); one of the most recent, pH (Nikhil *et al.*, 1993), is in fact a variant of Haskell. These languages usually introduce parallelism implicitly, for example by using an evaluation scheme such as *lenient evaluation* (Traub, 1991) which generates massive amounts of fine-grained parallelism. Unfortunately, these threads are often too small to be utilised efficiently by conventional thread technology. The solutions are to use hardware support for parallelism as with Monsoon (Papadopoulos, 1990) or *T (Nikhil *et al.*, 1992), or to use compiler optimisations to create larger threads statically (Traub *et al.*, 1992). In contrast, used with suitable performance analyses or measurement tools, evaluation strategies provide a readily available handle that can help to control thread size.

Sisal (McGraw, 1985) provides high-level loop-based constructs in a first-order dataflow language. These constructs support implicit control parallelism over arrays. The Sisal 90 language (Feo *et al.*, 1995) adds higher-order functions, polymorphism and user-defined reductions.

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 3.3 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 as follows. All of these strategic skeletons are much higher-level than the skeletons used in practice which have a careful implementation giving good data distribution, communication and synchronisation.

```
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 `demanding` 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)
```

Many strategic functions take the opposite approach to skeletons: a skeleton parameterises the control function over the algorithm, i.e., it takes sequential sub-programs as arguments. However, a strategic function may instead specify the algorithm and parameterise the control information, i.e. take a strategy as a parameter. Several of the functions we have already described take a strategy as a parameter, including `parBuffer`.

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 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 being written in the computation language, and composed in the coordination language.

The best known coordination languages are PCN (Foster and Taylor, 1994) and Linda (Gelernter and Carriero, 1992), both of which adopt a much more explicit approach than evaluation strategies. Since both languages support fully general programming structures and unrestricted communication, it is, of course, possible to introduce deadlock with either of these systems, unlike evaluation strategies.

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.* (1995) integrate the coordination language approach with the skeleton approach, providing a system for composing skeletons, SCL. 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 *farm*s, *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^3L language (Danelutto *et al.*, 1991), which provides a set of skeletons for common classes of algorithm.

Control abstraction Crowl and Leblanc (Crowl and Leblanc, 1994) have developed an approach with similarities with evaluation strategies. The approach is based on 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, control abstractions take a more control-oriented approach, relying on a meta-language to capture the essential notions of closure and control dependency that are directly encoded in our GPH-based system. In this 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 encourages us to believe 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, 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 Clean annotations. 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 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 language construct to attach schedules to expressions, 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 ⊥. There are also constructs to deal with task placement and dynamic load information which have no equivalent strategic formulation.

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 direct 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 schedules could be used to encode strategies, thus regaining the type information.

There can also be a loss of referential transparency when using schedules, since expressions involving *sched* may sometimes evaluate to ⊥, and other times to a non-⊥ value. This can happen both through careless use of demand and wait as in the deadlock-inducing example above, and conceivably if dynamic load information is used to demand an otherwise unneeded value. If the program terminates (yields a non-⊥ 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 are similar to GPH `par`s.

At a slightly higher level, Jones and Hudak have worked on commutative Monads (Jones and Hudak, 1993), that 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 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. Preliminary results indicate that acceptable parallelism is attained with relatively little programming effort.

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 is the fundamental mechanism that supports the separation of algorithm from dynamic behaviour: essentially 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.

Abuse of strategies As with any powerful language construct, 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. Adding a strategy to a function can also greatly increase space consumption, e.g. where the original function incrementally constructs and consumes a data structure, a strategic version may construct all of the data structure before any of it is consumed. Finally, strategies sometimes require additional runtime traversals of a data structure. In pathological cases care must be taken to avoid multiple traversals, e.g. when a small part of a large data structure has been changed, or with accumulating parameters. Many unnecessary traversals could be avoided with a runtime mechanism that tags closures to indicate their evaluation degree.

Additional applications This paper has focussed on the use of evaluation strategies for parallel programming, but we have also found them useful in other contexts. Strategies have been used for example in Lolita to force the evaluation of data structures that are transferred from Haskell to C (see section 5.3). Furthermore, they can cause a reduction in heap usage in cases where the strictness analysis is overly conservative and the normal evaluation would hang on to data that can be safely evaluated.

The separation of behavioural and algorithmic code provided by strategies suggests that they can be used to model the context in which a certain function is used. For example, in a sequential profiling setting strategies can define the evaluation degree of a function. The performance of the function can then be measured in the given context. Also, when tuning parallel performance, a driver strategy can define the pattern of parallelism generated in a certain context. This facilitates the testing of parts of the program in isolation.

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. To date we have only demonstrated modest wall-clock speedups for real programs, although this is partially due to the limited machine resources available to us. Several of the parallel functional implementations outlined in Section 6 achieve rather larger speedups. We would like to port the GUM runtime system underlying GPH to a larger machine, with a view to obtaining larger speedups. Another plausible target for GPH programs in the near future are modestly parallel workstations, with 8 processors for example. Interestingly it has required remarkably little effort to gain acceptable parallelism even for large, irregular programs like Lolita.

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 ordered 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.

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 directly-parallel code originally written. Compared with the strategic version presented earlier, the directly parallel version is both lower-level and more obscure.

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)
```

References

Arvind, Nikhil, R. S. and Pingali, K. K. (1989) I-Structures – Data structures for parallel computing. *ACM TOPLAS* **11**(4): 598–632.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Spielstein, J. and Zagha, M. (1993) Implementation of a Portable Nested Data-Parallel Language. *Proc. 4th ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, pp. 102–111. San Diego, CA.

Blelloch, G. E. (1996) Programming Parallel Algorithms. *Comm. ACM*, **39**(3): 85–97.

Botorog, G. M. and Kuchen, H. (1996) Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Computation. *Proc. 5th. IEEE Int. Symposium on High Performance Distributed Computing*, pp. 253–262. Syracuse, NY.

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

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

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

Burton F. W. (1991) Encapsulating Nondeterminacy in an Abstract Data Type with Deterministic Semantics. *J. Functional Programming.*, **1**(1): 3–20.

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

Crowl, L. A. and Leblanc, T. J. (1994) Parallel programming with control abstraction. *ACM TOPLAS*, **16**(3): 524–576.

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

Darlington, J., Guo, Y., To, H. W. and Yang, J. (1995) Parallel skeletons for structured composition. *Proc. 5th ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, pp. 19–28. Santa Barbara, CA.

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

Feo, J., Miller, P., Skedziewlewski, S., Denton, S. and Soloman, C. (1995) Sisal 90. *Proc. HPFC '95 – High Performance Functional Computing*, pp. 35–47. Denver, CO.

Finne, S. O. and Burn, G. L. (1993) Assessing the evaluation transformer model of reduction on the Spineless G-Machine. *Proc. FPCA '93*, pp. 331–340. Copenhagen, Denmark.

Gelernter, D. and Carriero, N. (1992) Coordination languages and their significance. *Comm. ACM*, **32**(2): 97–107.

Flanagan, C. and Nikhil, R. S. (1996) pHluid: The design of a parallel functional language implementation. *Proc. ICFP '96*, pp. 169–179. Philadelphia, Penn.

Foster, I. and Taylor, S. (1994) A compiler approach to scalable concurrent-program design. *ACM TOPLAS*, **16**(3): 577–604.

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

Halstead, R. (1985) MultiLisp: A language for concurrent symbolic computation. *ACM TOPLAS*, **7**(4): 501–538.

Hammond, K., Loidl, H.-W. and Partridge, A. S. (1995) Visualising granularity in parallel programs: A graphical winnowing system for Haskell. *Proc. HPFC'95 – High Performance Functional Computing*, pp. 208–221. Denver, CO.

Hartel, P., Hofman, R., Langendoen, K., Muller, H., Vree, W. and Hertzberger, L. O. (1995) A toolkit for parallel functional programming. *Concurrency – Practice and Experience.*

High Performance Fortran Forum (1993) *High Performance Fortran Language Specification.*

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

Hudak, P. (1986) Para-functional programming. *IEEE Computer*, **19**(8): 60–71.

Hudak, P. (1988) Exploring para-functional programming: Separating the what from the how. *IEEE Software*, **5**(1): 54–61.

Hudak, P. (1991) Para-functional programming in Haskell. In *Parallel Functional Languages and Computing*, pp. 159–196. ACM Press/Addison-Wesley.

Hudak, P. and Anderson, S. (1987) Pomset interpretations of parallel functional languages. *Proc. FPCA '87: Lecture Notes in Computer Science 274*, pp. 234–256. Springer-Verlag.

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

Jackson, W. K. and Burton F. W. (1993) Improving intervals. *J. Functional Programming.*, **3**(2): 153–169.

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

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

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

Kranz, D., Halstead, R. and Mohr, E. (1989) Mul-T: A high-performance parallel Lisp. *Proc. PLDI '89*, pp. 81–90. Portland, OR.

Lauer, M. (1982) Computing by homomorphic images. In B. Buchberger, G. E. Collins, R. Loos and R. Albrecht (eds.), *Computer Algebra – Symbolic and Algebraic Computation*, pp. 139–168. Springer-Verlag.

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

McColl, W. F. (1996) Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, **12**(4): 265–272.

McGraw, J. R., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J. R. W., Kirkham, C., Boyce, W. and Thomas, R. (1985) *SISAL: Streams and Iteration in a Single Assignment Language*. Language Reference Manual Version 1.2, Manual M-146, Rec. 1, Lawrence Livermore National Laboratory.

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

Mirani, R. and Hudak, P. (1995) First-class schedules and virtual maps. *Proc. FPCA '95*, pp. 78–85. La Jolla, CA.

Mohr, E., Kranz, D. A. and Halstead, R. H. (1991) Lazy task creation – a technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel and Distributed Systems*, **2**(3): 264–280.

Morgan, R. G., Smith, M. H. and Short, S. (1994) Translation by meaning and style in Lolita. *Int. BCS Conf. – Machine Translation Ten Years On*, Cranfield University.

Nikhil, R. S., Arvind and Hicks, J. (1993) pH language proposal. DEC Cambridge Research Lab Technical Report.

Nikhil, R. S., Papadopolous, G. M. and Arvind (1992) *T: a multithreaded massively parallel architecture. *Proc. ISCA '92*, pp. 156–167.

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

Rogers, A., Carlisle, M. C., Reppy, J. H. and Hendren, L. J. (1995) Supporting dynamic data structures on distributed-memory machines. *ACM TOPLAS*, **17**(2): 233–263.

Papadopoulos, G. M. and Culler, D. E. (1990) Monsoon: An explicit token store architecture. In *Proc. ISCA '90*, Seattle, WA.

Peterson J. C. and Hammond, K. (eds.) (1996) *Report on the Non-Strict Functional Language, Haskell, Version 1.3*, (1996). `http://haskell.org/report/index.html`

Rabhi, F. A. (1993) Exploiting parallelism in functional languages: a 'Paradigm-oriented' approach". In Dew, P. and Lake, T. (eds.), *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press.

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

Rose, J. and Steele, G. L. Jr. (1987) C*: an extended C language for data parallel programming. *Technical Report PL87-5*, Thinking Machines Corp.

Runciman, C. and Wakeling, D. (1995) *Applications of Functional Programming* UCL Press.

Sansom, P. M. and Peyton Jones, S. L. (1995) Time and space profiling for non-strict, higher-order functional languages. *Proc. POPL '95*, pp. 355–366.

Smirni, E., Merlo, A., Tessera, D., Haring, G. and Kotsis, G. (1995) Modeling speedup of SPMD applications on the Intel Paragon: a case study. In B. Hertzberger and G. Serazzi (eds.), *Proc. High Performance Computing and Networking (HPCN '95)*, pp. 94–101. Milan, Italy.

Subhlok, J., Stichnooch, J. M., O'Hallaron, D. R. and Gross, T. (1993) Exploiting task and data parallelism on a multicomputer. *Proc. 4th ACM Conf. on Principles & Practice of Parallel Programming (PPoPP)*, pp. 13–22. San Diego, CA.

Traub, K. R. (1991) *Implementation of Non-Strict Functional Programming Languages*, Research Monographs in Parallel and Distributed Computing, Pitman/MIT Press.

Traub, K. R., Culler, D. E. and Schauser, K. E. (1992) *Global analysis for partitioning non-strict programs into sequential threads*. In *LFP '92*, pp. 324–334. San Francisco, CA.

Trinder, P. W., Hammond, K., Mattson, J. S. Jr., Partridge, A. S. and Peyton Jones, S. L. (1996) GUM: a portable parallel implementation of Haskell. *Proc. PLDI '96*, pp. 79–88. Philadelphia, Penn.

Trinder, P. W., Hammond, K., Loidl, H.-W., Peyton Jones, S. L. and Wu, J. (1996) A case study of data-intensive programs in parallel Haskell. *Proc. Glasgow Functional Programming Workshop*. Ullapool, Scotland.

Wadler, P. L. and Hughes, R. J. M. (1987) Projections for strictness analysis. *Proc. FPCA '87*.