Seq no more: Better Strategies for Parallel Haskell

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Patrick Maier
Heriot-Watt University
P.Maier@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University
H.W.Loidl@hw.ac.uk

Mustafa K Aswad
Heriot-Watt University
mka19@hw.ac.uk

Phil Trinder
Heriot-Watt University
P.W.Trinder@hw.ac.uk

Abstract

We present a complete redesign of Evaluation Strategies, a key abstraction for specifying pure, deterministic parallelism in Haskell. Our new formulation preserves the compositionality and modularity benefits of the original, while providing significant new HWL: usability benefits. First, we introduce an evaluation-order monad to provide clearer, more generic, and more efficient specification of parallel evaluation. Secondly, the new formulation resolves a subtle space management issue with the original strategies, allowing parallelism (sparks) to be preserved while reclaiming heap associated with superfluous parallelism. Related to this, the new formulation provides far better support for speculative parallelism as the garbage collector no longer needlessly speculates. Finally, the new formulation provides improved compositionality: we can directly express parallelism embedded within lazy data structures, producing more compositional strategies, and our basic strategies are parametric in the coordination combinator, facilitating a richer set of parallelism combinators. We give measurements over a range of benchmarks demonstrating that the runtime overheads of the new formulation relative to the original are low, and the new strategies even yield slightly better speedups on average than the original strategies.

Categories and Subject Descriptors D.1.1 [Applicative (Functional) Programming]: D.1.3 [Concurrent Programming]

General Terms Performance and Measurement

Keywords Parallel functional programming, strategies

1. Introduction

Evaluation strategies (?), or “strategies” for short, are a key abstraction for adding pure, deterministic, parallelism to Haskell programs. Using strategies, parallel specifications can be built up in a compositional way, and the parallelism can be specified independently of the main computation. Despite the apparent conflict between lazy evaluation and the eagerness implied by parallelism, evaluation strategies show that non-strictness and parallelism can co-exist in a coherent programming model, and non-strictness even has some advantages for parallel languages (??). Strategies have been used for some 15 years in a number of parallel variants of Haskell (??). This paper presents a complete redesign of the strategy abstraction.

HWL: achieving a generalisation of custom-made abstractions for parallelism to a general-purpose evaluation monad Our reformulation preserves the key compositionality and modularity benefits of the original strategies (Section 4), together with their low time and space overheads (Section 6), while providing the following additional benefits:

• HWL: claim SeqList as a DeepSeq replacement?

• HWL: more efficient spec. or easier to use spec?

• Clearer, more generic and more efficient specification of parallel evaluation. Describing a parallel algorithm requires specifying an order of evaluation, something which the Haskell language deliberately, and rightly, leaves unspecified. In the new strategies we introduce an evaluation-order monad, allowing the ordering of a set of evaluations to be specified in a perspicuous and compositional way (Section 4). Moreover, by using Applicative Functors and the Traversable class (?), we can define generic regular strategies over data structures (Section 4.5). Our framework also supports fusion, which allows the intermediate lists introduced by modular strategies to be eliminated by the compiler (Section 4.8).

• The new strategies resolve a subtle space management issue where the original strategies retain heap unnecessarily (Section 6). The crux of the space management challenge is to preserve parallelism (sparks), while being able to reclaim the heap associated with superfluous parallelism. Our measurements demonstrate improved space behaviour for existing parallel programs simply by switching to the new strategies (Section 6). Furthermore, the new strategies support speculative parallelism with unnecessary speculative tasks being pruned automatically by the garbage collector, something which was not possible with original strategies.

• There is a class of important parallel coordination abstractions that cannot be expressed as original strategies, but can be expressed in the new formulation. The feature that this class of abstractions has in common is that they all embed parallelism within lazy components of a data structure, a technique that is essential for parallelising stream-processing pipelines. In the
original strategies we could write these functions, but they were not instances of the strategy abstraction and so could not be used compositionally. These drawbacks are resolved by the new framework (Section 5.4).

• Motivated by wanting to have different versions of \( \text{par} \) to control locality in large architectures, the new formulation allows for abstracting over the coordination combinator used (Section 5.4).

Sadly, however, we must all pay for our lunch, and the new formulation raises three issues.

• There is some extra complexity in the implementation HWL: or: construction of new of strategies. However, many casual users of the library are insulated from the changes: using and composing strategies works exactly as before, modulo some renaming. Only users who need to define their own strategies will have to become familiar with the new idioms, and there should now be fewer such users given that we provide generic strategies over any Traversable data type.

• The original strategies provided a strong identity safety property, namely that \( (\text{using} \ a) \) is always an identity function for any Strategy \( a \). The new strategies cannot provide the same guarantee, although the library strategies are identities, and the combinators preserve the property. Safety can be regained at the expense of expressiveness by making the strategy type abstract, giving the programmer a choice of expressiveness/safety levels (Section 5.4).

• To express control parallelism an original strategy may freely spark expressions. The corresponding new strategy must carefully preserve any sparked expressions (Section 4.6).

The new strategies are incorporated in the Haskell parallel package\(^1\) and the version we describe in this paper will be released as version 2.3.HWL: or: 3.0. All the code for our benchmarks is available online (Section 6), and the results were obtained with a recent development GHC snapshot (6.13 as of 20.5.2010).

2. Original Strategies

Pure parallelism in Haskell is achieved using only two primitives, \( \text{par} \) and \( \text{pseq} \), with the following type\(^2\):

\[
\text{par} :: a \to b \to b \\
\text{pseq} :: a \to b \to b
\]

The \( \text{par} \) combinator introduces a potential for parallel evaluation. When \( \text{par} \) is applied to two arguments, it returns the value of its second argument, while its first argument is possibly evaluated in parallel. We say "possibly", because as far as the program is concerned, the result of \( \text{par} \ a \ b \) is always \( b \); it makes no difference to the meaning of the program whether \( a \) is evaluated in parallel or not. We should think of \( \text{par} \) as an annotation; it merely hints to the Haskell implementation that it might be beneficial to evaluate the first argument in parallel.

What if the computation evaluated in parallel has the value \( \bot \), or an error? Surely then it makes a difference to the meaning of the program whether it is evaluated, or not? In fact it does not – the system is required to ensure that the semantics of \( \text{par} \ a \ b \) is always \( b \), regardless of the value of \( a \). \( \bot \) or otherwise. In practice, this isn’t a problem for typical Haskell implementations, as a lazy computation can already have value \( \bot \).

It is not enough to provide \( \text{par} \) alone, because generally when suggesting that something is to be evaluated in parallel, it is useful to be able to say what it is to be evaluated in parallel with. Haskell neither specifies nor requires a particular order of evaluation, so normally the programmer has no control over this aspect of their program’s execution. We have no control over when a particular call to \( \text{par} \) will be evaluated, or what will be evaluated before or after it (or indeed in parallel with it). This is the reason for \( \text{pseq} \): a call \( \text{pseq} \ a \ b \) introduces an order-of-evaluation requirement that \( a \) is evaluated to weak-head normal form before evaluating \( b \) and returning its value. HWL: maybe defer WHNF mentioning to highlight “order" here The denotational semantics of \( \text{pseq} \) are

\[
\text{pseq} \ a \ b = \bot, \quad \text{if } a = \bot = b, \quad \text{otherwise}
\]

and the operational semantics are that \( a \) must be evaluated to weak head normal form before \( b \) is evaluated (?).

An example to illustrate the usage of \( \text{par} \) and \( \text{pseq} \) follows, using the traditional Fibonacci function. More examples can be found in the literature (?).

\[
\text{fib} :: \text{Int} \to \text{Int} \\
\text{fib} \ n = \begin{cases} 1 & \text{if } n \leq 1 \\ \text{let} \ x = \text{fib} (n-1) \\ y = \text{fib} (n-2) \\ x \ '\text{par}' \ (y \ '\text{pseq}' \ x + y + 1) & \text{otherwise} \end{cases}
\]

The Fibonacci computation is shaped like a binary tree. At each node of the computation we combine \( \text{par} \) and \( \text{pseq} \) to evaluate one branch in parallel with the other branch. The pattern here is a common one: in \( x \ '\text{par}' \ (y \ '\text{pseq}' \ e) \), typically \( e \) involves both \( x \) and \( y \). The effect of this pattern is to cause \( x \) to be evaluated in parallel with \( y \). When the evaluation of \( y \) is complete, computation proceeds by evaluating \( e \). Here the \( \text{pseq} \) is used to control evaluation order.

The parallelism here is independent of the number of processors; every time \( \text{par} \) is evaluated it creates a new opportunity for some work to be evaluated in parallel (a spark), but the implementation is free to ignore these opportunities. Indeed typical usage of \( \text{par} \) creates many more sparks than there are processors available to execute them, and the surplus sparks are simply discarded by the runtime system.

2.1 Strategies

The basic programming model described above provides the raw material for expressing parallelism in Haskell. Building on this, strategies were invented as an abstraction layer over \( \text{par} \) and \( \text{pseq} \) to allow larger-scale parallel algorithms to be expressed.

Strategies are a remarkably simple idea. In the original formulation, a strategy is a function of type \( a \to () \) for some \( a \):

\[
\text{type } \text{Strategy } a = a \to ()
\]

Thus, a Strategy may evaluate its argument either in full or in part, and it may only return () (or diverge). Crucially, using \( \text{par} \) and
pseq, a strategy may specify a recipe for evaluating its argument in parallel.

Some basic strategies can be defined as follows.

\[
\begin{align*}
    \text{r0} & : : \text{Strategy a} \\
    \text{r0} x & = () \\
    \text{rwhnf} & : : \text{Strategy a} \\
    \text{rwhnf} x & = x \text{ `pseq' } () \\
    \text{rfn} & : : \text{NFData a } \Rightarrow \text{Strategy a} \\
    \text{-- rnf is a method in the class NFData} \\
    \text{r0} & \text{ is a strategy that evaluates nothing of its argument, rwhnf evaluates its argument to weak-head normal form, and rfn evaluates its argument completely. The definition of rfn depends on the structure of its argument, so it is defined using a type class NFData, which has to be instantiated separately for each data type (the strategies library provides instances for common types such as Booleans, Integers, lists and tuples).}
\end{align*}
\]

Strategies are applied with the \texttt{using} combinator:

\[
\begin{align*}
    \text{using} & : : \text{a } \Rightarrow \text{Strategy a } \Rightarrow \text{a} \\
    \text{using} x s & = s x \text{ `pseq' } x
\end{align*}
\]

So far we haven’t presented any strategies containing actual parallelism. A simple one is \texttt{parList}, which applies a strategy to each element of a list in parallel:

\[
\begin{align*}
    \text{parList} & : : \text{Strategy a } \Rightarrow \text{Strategy [a]} \\
    \text{parList strat} [\text{x:xs}] & = (\text{strat x `par' \parList strat xs})
\end{align*}
\]

The function \texttt{parList} illustrates the compositional nature of the strategies abstraction: it takes as an argument a strategy to apply to each list element, and returns a strategy for the whole list. The strategy argument is typically used to specify the evaluation degree, that is, how much each list element should be evaluated. For instance, \texttt{parList rwhnf} causes each spark to evaluate its list element as far as the top-level constructor, whereas \texttt{parList rnf} evaluates the elements completely. Various evaluation degrees between these two extremes are possible, such as evaluating the spine of a list (we give examples later in Section 4.7).

The \texttt{parList} function can also be used to illustrate the modular nature of strategies; for example,

\[
\begin{align*}
    \text{parMap strat f xs} & = \text{map f xs `using' parList strat}
\end{align*}
\]

The \texttt{parMap} function takes a strategy \texttt{strat}, a function \texttt{f}, and a list \texttt{xs} as arguments and maps the function \texttt{f} over the list in parallel, applying \texttt{strat} to every element. Note how the construction of the result with \texttt{map}, on the left of \texttt{using}, is separate from the specification of the parallelism, on the right. This is a small-scale example, but the idea also scales to much more elaborate settings (3).

The key to the modularity is lazy evaluation. The argument to a strategy can be a complex data structure with lazy components, or even a lazily-created data structure, and this allows the algorithm that creates the data structure to be separated from the strategy that specifies how to evaluate it. It’s not a panacea: not all algorithms lend themselves to being decomposed in this way, and the intermediate lazy data structure has costs of its own. Nevertheless, in many cases the modularity benefits outweigh the costs, and sometimes the intermediate data structure can be automatically eliminated by the compiler (Section 4.8).

3. Space Management: Preserving Parallelism, not Garbage

In this section we describe the main problem in the original strategies formulation that prompted the redesign described in this paper. The problem we are about to describe only came to light recently (2).

To understand the problem we need to consider how \texttt{par} is implemented. When the Haskell program evaluates the expression \texttt{par a b}, the runtime system saves a pointer to the heap node representing \texttt{a} in a data structure that we call a spark pool. For our purposes, the spark pool is simply a set of pointers to heap objects representing computations that have been sparked by \texttt{par}. The runtime system from time to time removes objects from the pool in order to evaluate them using idle processors, so-called lazy task creation (3). More details on the implementation of spark pools can be found in (2); the particular implementation details are not important here.

How should the storage management system, in particular the garbage collector, treat the spark pool? There are two main alternatives, which we call \texttt{ROOT} and \texttt{WEAK} respectively, following the terminology of (2):

\textbf{PWT:} There is a 3rd alternative: the GC could use \texttt{ROOT} initially, and if it fails to recover sufficient space it can then discard some sparks. Kevin Hammond says a version of GRIP did this.

1. \textbf{ROOT}: entries in the spark pool should be considered implicitly live. That is, the spark pool is a source of roots for the garbage collector.

2. \textbf{WEAK}: an entry in the spark pool is only alive if the object to which it points is independently reachable. That is, the spark pool contains weak pointers in the usual terminology.

In fact, both of these policies lead to problems with original strategies. First, let us consider \texttt{WEAK}, and examine how it works with the definition of \texttt{parList} in the previous section. The sparks created by \texttt{parList} are all expressions of the form \texttt{(strat x)} for some strategy \texttt{strat} applied to some list element \texttt{x}. Now, every such expression is uniquely allocated for the sole purpose of being passed to \texttt{par}; the spark pool will contain references to many expressions of the form \texttt{(strat x)}, and in every case, the reference from the spark pool is the only reference to that expression in the heap. So, by definition, if we adopt the \texttt{WEAK} policy then every spark created by \texttt{parList} will be discarded by the garbage collector, and we lose all the parallelism.

Moreover, there is no definition of \texttt{parList} that can avoid this problem. The only value that the \texttt{parList} strategy can return is \texttt{()}, so the only way that \texttt{parList} can create a reachable spark is by sparking part of the structure it was originally given, such as the list elements. For example, we can define a non-compositional variant of \texttt{parList} that works:

\[
\begin{align*}
    \text{parListWHNF} & : : \text{Strategy [a]} \\
    \text{parListWHNF} [] & = () \\
    \text{parListWHNF} (x:xs) & = x \text{ `par' \parListWHNF strat xs}
\end{align*}
\]

But unfortunately we lose the compositional nature of strategies that was so appealing about the original formulation.

So what about the alternative garbage collection policy, \texttt{ROOT}, where we treat the spark pool as a source of roots? Considering the \texttt{parList} example again, the spark pool would still contain references to expressions of the form \texttt{(strat x)} in the heap, but this time all the expressions will be retained by the garbage

\[
\begin{align*}
    \text{parListWHNF :: Strategy [a]} \\
    \text{parListWHNF} [] & = () \\
    \text{parListWHNF} (x:xs) & = x \text{ `par' \parListWHNF strat xs}
\end{align*}
\]
3.3 Summary

PM: Also, should we have this summarising table already here? It declares the impact of the new strategies on GC policy before these new strategies have been introduced.

SM: I did worry about the forward ref, yes. I’m open to suggestions.

The following table summarises the interaction between the choice of GC policy (ROOT or WEAK), original strategies (Section 2), or new strategies (Section 3), and speculative versus non-speculative parallelism.

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Parallelism</th>
<th>ROOT</th>
<th>WEAK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>non-speculative</td>
<td>space leaks</td>
<td>lost parallelism</td>
</tr>
<tr>
<td>New</td>
<td>speculative</td>
<td>space leaks</td>
<td>OK</td>
</tr>
</tbody>
</table>

4. A New Formulation of Strategies

The difficulties with managing the space behaviour of sparks described in Section 3 are rooted in the choice of the type for strategy functions: if a strategy function returns the unit type (), then there is no way for it to spark new expressions and to return them to the caller, thus ensuring that the sparked expressions remain reachable by the garbage collector.

The key idea in our reformulation is that a strategy returns a new version of its argument, in which the sparked computations have been embedded. For example, when sparking a new parallel task of the form (strat x), rather than discarding this expression, the strategy will now build a new version of the original data structure with (strat x) in place of x. The caller will consume the new data structure and discard the old, so that the parallel task (strat x) remains reachable as long as the consumer requires it. Furthermore, if the consumer evaluates (strat x) before it is evaluated by a parallel thread, then the spark fizzes; superfluous parallelism is discarded by the garbage collector, which is exactly what we need.

Perhaps our strategies should be identity functions. However, the simplest identity function, a -> a, is not a suitable candidate. Functions of this type are necessarily strict, so we cannot express r0, the strategy that performs no evaluation of its argument, as a function of this type. To accommodate r0, the codomain has to be lifted. We use a trivial lifting, Eval, and provide a way to unlift, runEval.

type Strategy a = a -> Eval a
data Eval a = Done a
runEval :: Eval a -> a
runEval (Done a) = a

The rationale for the names will become clear shortly. Now we can define some basic strategy combinators using the new type:

r0 :: Strategy a
r0 x = Done x
rseq :: Strategy a
rseq x = x ‘rseq’ Done x
rpar :: Strategy a
rpar x = x ‘rpar’ Done x
rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x ‘rseq’ Done x
The new basic strategies \( r_0 \), \( r_{seq} \) and \( r_{deepseq} \) are analogues to the original strategies \( r_0 \), \( r_{whnf} \) and \( r_{nf} \) respectively (in fact, \( r_{deepseq} \) uses the original \( r_{nf} \)).

4.1 The Evaluation-order Monad

We can declare \( \text{Eval} \) to be a monad. There are two choices here: either it is the standard identity monad, or it is a strict identity monad. The latter turns out to be a much more useful choice:

```haskell
instance Monad Eval where
  return x = Done x
  Done x >>= k = k x
runEval :: Eval a -> a
runEval (Done x) = x
```

The strict identity monad gives us a convenient and flexible notation for expressing evaluation order, i.e. the ordering between applications of \( r_{seq} \) and \( r_{par} \), which is exactly what we need for expressing basic parallel evaluation. For example, the following fragment of \( \text{fib} \):

```haskell
let
  x = fib (n-1)
  y = fib (n-2)
in
```

can be rewritten as

```haskell
runEval $ do
  x <- rpar (fib (n-1))
  y <- rseq (fib (n-2))
  return (x + y + 1)
```

which clearly expresses the ordering between \( r_{par} \) and \( r_{seq} \), using a notation that Haskell programmers will find familiar.

Programmers using the new strategies API no longer need to use \( \text{par} \) and \( \text{pseq} \) to construct new strategies, instead they use the \( \text{Eval} \) monad with \( r_{par} \) and \( r_{seq} \). The \( \text{Eval} \) monad raises the level of abstraction for \( \text{pseq} \) and \( \text{par} \); it makes fragments of evaluation-order first class, and lets us compose them together. We should think of the \( \text{Eval} \) monad as an Embedded Domain-Specific Language (EDSL) for expressing evaluation order, embedding a little evaluation-order-constrained language inside Haskell, which does not have a strongly-defined evaluation order.

Figure 1 summarises the differences between the API for the original strategies and the new strategies. Note that we have redefined a few combinators using the monadic style consistently; using \( \text{return} \) in place of \( \text{Done} \), for example.
4.2 Eval, applicatively

An evaluation order is often something we want to impose on an existing expression. Since Eval is a monad, it is also an Applicative Functor (?:)

instance Functor Eval where
  fmap f x = x >>= return . f

instance Applicative Eval where
  pure x = return x
  (<*>) = ap

This means that we can use applicative notation for threading “evaluation order” through an expression. Here’s a simple example: in one of our benchmarks (Coins), a result value is defined as

res = append left right

and we wanted to spark left in parallel with right. We could use the monadic syntax as we did for the fib example above, but sometimes even the monadic syntax is too heavy, and obscures the structure of the original code. The Applicative operators pure, <$> and <*> let us rewrite the expression to include the parallelism, without losing its structure:

res = runEval $ append <$> rpar left <*> rseq right

One might object that this is not a modular specification of parallelism, and that would be a fair criticism. However, note that apart from the introduction of rpar and rseq, the translation to applicative style is mechanical, so this is a minimal and yet precise way to add a little parallelism to an existing expression. We discuss how to recover modularity in cases like this in Section 4.6.

Applicative notation fixes the ordering to be depth-first, so in cases where depth-first is not appropriate the monadic syntax has to be used.

4.3 Using Strategies

As with the original strategies, a strategy application operator is provided:

using :: a -> Strategy a -> a
x 'using' s = runEval (a x)

The using function is defined to have the lowest precedence and associate to the left, that is e ‘using’ s1 ‘using’ s2 stands for (e ‘using’ s1) ‘using’ s2. This stacking of strategies being similar to the stacking of function applications, there is a strategy composition dot such that

(e ‘using’ s1) ‘using’ s2 = e ‘using’ (s2 ‘dot’ s1)

Just like function composition, dot has highest precedence and associates to the right, so the parentheses can be dropped from the above equation.

4.4 Compositional Strategies over Data

We build strategies over data types by first constructing a basic strategy for the data type, parameterised over strategies for the components of the type. The basic strategy traverses the data type in the Eval monad, applies the argument strategies to the components in depth-first order, and builds a new instance of the type.

As an example, consider the Strategy combinator evalList, which walks over a list and applies the argument strategy s to every element:

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x
  xs' <- evalList s xs
  return (x':xs')

The evalList combinator generalises both parList and seqList of original Strategies, and more besides. For example, parList is obtained by composing the element strategy a with rpar:

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar ‘dot’ s)

Original strategies had a seqList function, whereas we do not provide a seqList in the new strategies. In fact, evalList is equivalent to seqList, but it is not immediately obvious why this should be so – seqList is defined in terms of pseq, but there are no pseqs to be found in the definition of evalList. The purpose of seqList is to apply the strategy s to each element of the list in order from left to right, and it achieves this ordering by using pseq at each step. In evalList, we achieve the same ordering, but by using the Eval Monad instead: the Eval Monad explicitly sequences the application of the strategy s to each list element in order, so no pseqs are necessary.

We can specialise evalList in more ways. A number of new parallel primitives are envisioned, for instance, a bounded par that restricts locality, e.g. a spark with a low bound should be executed “nearby”. An advantage of the new strategies that all these primitives can be passed as parameters, saving code replication.

4.5 Generic Strategies

The Traversable class provides a convenient way to thread any Applicative computation through the components of a data structure in a depth-first manner, performing any effects on the way whilst building a new data structure (?:). This is exactly what we need for defining strategies over regular data structures such as lists and trees: i.e. a means of traversing the data structure using Eval, applying a strategy at the leaves, and building a new structure to return.

The method traverse has the following type:

traverse :: (Traversable t, Applicative f) => (a -> f b) -> t a -> f (t b)

This function is so generic it’s not immediately obvious how it can be applied in our setting. However, if we specialise a -> f b to Strategy a, then we get:

evalTraversable :: Traversable t
  => Strategy a
  -> Strategy (t a)
evalTraversable = traverse

this is a generic parameterised Strategy for any Traversable data type. It has evalList as an instance, and gives us strategies for types like Maybe and Array for free. Adding parallelism to the generic strategy is straightforward:

parTraversable :: Traversable t
  => Strategy a
  -> Strategy (t a)
parTraversable = evalTraversable (rpar ‘dot’ s)
4.6 Modularity

The key modularity property we have is that e ‘using’ s is observably equivalent to e, at least in so far as it is defined (the former may be less defined than the latter). The point of this guarantee is that someone who only wants to understand the algorithm can ignore the strategies, i.e. every ‘using’ s.

Of course, this property is only useful in cases where we can actually make use of using. Some of the examples we have already seen are not easily expressed with using; consider for example fib from Section 4.3.

```haskell
runEval $ do
  x <- rpar (fib (n-1))
  y <- rsseq (fib (n-2))
  return (x + y + 1)
```

This kind of parallelism is known as control or task parallelism, where the parallelism follows the control structure of the program. However, we cannot consider this a modular specification of parallelism, as it clearly interleaves the algorithm with the coordination.

We can write a modular version:

```haskell
x + y + 1 ‘using’ strat
where
  x = fib (n-1)
  y = fib (n-2)
  strat v = runEval $ do rpar x; rsseq y; return v
```

This strategy looks odd. We aren’t using the result of rpar, which should raise the red flags: normally the result of rpar should be discarded by the garbage collector, or become a space leak. However, it is acceptable to discard the result of rpar if the argument is a variable, and that variable is already shared by the result, as it is in this case.

This is a somewhat subtle rule-of-thumb, and the user may well prefer the original direct definition using runEval. Note that the same technique was possible with original strategies, although there we had no option to use the more direct runEval style.

The technique is applied to a more realistic example in section 5.3.

4.7 Sequential Strategies

An important class of strategies specify only evaluation degree, i.e. do evaluation only, and introduce no parallelism. Since they create no sparks, there is no need for these strategies to rebuild the data structure that they are passed. For example, if we were to define a strategy that evaluates a list sequentially as follows:

```haskell
forceList = evalList rsseq
```

then the result is a strategy that is not only needlessly inefficient, but worse, may overflow the stack on long lists because evalList is not tail-recursive.

Hence we treat the class of strategies that do evaluation only differently. The type SeqStrategy is a sequential strategy, and has the same definition as original strategies:

```haskell
type SeqStrategy a = a -> ()
```

4.8 Fusion

SM: I don’t think it belongs with advanced strategies, but maybe we should have a place to collect together “performance considerations” or something like that. HWL: Agree: stay and merge

Using strategies in a modular way often implies that an intermediate data structure is generated by the computation, filtered by the strategy, and finally consumed upstream. Consider once again parMap:

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

The list produced by map is consumed by parList, which generates another list to return to the caller of parMap. Furthermore, there is an extra traversal: both map and parList traverse the complete list.

Ideally we would like to have this intermediate structure and the extra traversal be eliminated by the compiler. Fortunately, using GHC it is almost trivial to arrange that this optimisation occurs: GHC provides user-defined transformation rules, which are used to

We make SeqStrategy a “subtype” of Strategy a by providing an explicit upcast ins, which evaluates a sequential strategy before returning the evaluated argument into the Eval monad.

```haskell
ins :: SeqStrategy a -> Strategy a
ins ss x = ss x ‘pseq’ return x
```

Like ordinary strategies, SeqStrategies can be combined, for example:

```haskell
seqList :: SeqStrategy a -> SeqStrategy [a]
seqList ss [] = ()
seqList ss (x:xs) = ss x ‘seq’ seqList ss xs
```

As the order of evaluation of substructures is irrelevant here, these combinator may use the ordinary Haskell seq operator instead of pseq, granting the compiler more freedom to optimise the order of evaluation. In constrast, the upcast ins must use pseq to force evaluation of the sequential strategy before returning.

Finally, seqFoldable is the sequential strategies’ counterpart to the generic strategy evalTraversable.

```haskell
seqFoldable :: Foldable t => SeqStrategy a
            -> SeqStrategy (t a)
seqFoldable ss = foldl’ (const ss) ()
```

seqFoldable strictly applies a strategy to all elements of a Foldable data structure. Given the simpler return type of sequential strategies, seqFoldable is defined already for Foldable data structures, which form a super class of the Traversable data structures.

Sequential strategies are widely used, and the example below transposes a list of matrices, each represented as a list of lists, in parallel without evaluating the matrix elements. The sequential strategy (seqList (seqList r0)) will evaluate just the shape of a matrix, while the parMap specifies the parallel transpose (S r0 here is the SeqStrategies equivalent of r0):

```haskell
parMap (ins (seqList (seqList S r0))) transpose matrices
```

The detailed control of evaluation degree provided by sequential strategies may also be useful for tuning sequential programs. In effect sequential strategies generalise existing abstractions like DeepSeq.
implement list fusion between many of the standard list-producing and consuming library functions. Our `parList` is defined in terms of `parTraverse`, which is defined in terms of `traverse`, and the list instance of `traverse` happens to be defined in terms of `foldr`. The intermediate list between `map` and `foldr` is automatically removed by GHC’s transformation rules, so in fact `parMap` compiles to an efficient single-traversal loop.

Unfortunately, the measurements we report in Section 6 are without the benefit of fusion as the standard `Data.Traverse` library requires an extra annotation (an `INLINE` pragma on `traverse`). However, we have verified that with the `INLINE` annotation in place, fusion does occur as expected.

5. Advanced Strategies

**HWL**: Where possible use examples from the benchmarks to motivate use

This section discusses how advanced features such as clustering, buffering and parallel patterns, can be expressed in the new strategies. Such features are essential for real parallel applications, and are used in the kernels measured in Section 6.3.

5.1 Embedded Strategies: Rolling Buffers

Some parallel abstractions that are important for parallel performance tuning rely on embedding parallelism inside a lazy data structure, such that opportunities for parallel evaluation are created “on demand” by the consumer of the data structure. The most commonly encountered example is a parallel buffer (?):

```
parBuffer :: Int -> Strategy a -> Strategy [a]
```

Informally the idea is that `parBuffer n s xs` yields a list in which evaluation of the `i`th element induces parallel evaluation of the `i`th element with the first `n` elements being evaluated in parallel immediately. The result list must therefore be lazy, at least beyond the first `n` elements.

In the original strategies, while `parBuffer` could be defined perfectly well, it could not be expressed as a `Strategy`, because it returns a new list containing parallelism embedded in the lazy components. That is, the type is

```
parBuffer :: Int -> Strategy a -> [a] -> [a]
```

This was an unfortunate wart, because it meant that `parBuffer` could not be used as the argument to a paramterised strategy function and thus compositionality was diminished.

Fortunately embedded parallelism can be directly expressed in the new strategy formulation, and so `parBuffer` and functions like it are instances of the `Strategy` type.

A fully compositional implementation of `parBuffer` can be found below. It implements a rolling buffer (with amortised constant overhead) by means of the highly optimised functional queue data structure provided by `Data.Sequence`. The rolling buffer functionality is provided by `roll`, which takes a functional queue (the buffer) and a list of elements yet to go into the buffer, and returns a list (via the `Eval1` monad). Whenever the result list is demanded, `roll` applies the strategy `s` to the first element `z` to go into the buffer and sticks the result at the end of the queue (by calling `q |> z`). Then it pulls the first element `y'` out of the queue (by matching `viewl ...` against `y' : q'`) and returns it as the head of the result list while embedding the recursive call into the tail of the result list.

```
evalBuffer n s xs =
  roll (fromList (ys 'using' evalList s)) zs
  where
    (ys,zs) = splitAt n xs
    roll q [] = return (toList q)
    roll q (z:zs) = do z' <- s z
                     let y':q' = viewl (q |> z')
                         return (y' : runEval (roll q' zs))
```

```
parBuffer :: Int -> Strategy a -> Strategy [a]
parBuffer n s = evalBuffer n (rpar 'dot' s)
```

**HWL**: names: list-, queue-based `parBuffer`?

5.2 Clustering

When tuning the performance of parallel programs it is often important to increase the size of parallel computation, i.e. to use a coarser granularity, in order to achieve a good ratio of computation versus coordination costs. Implementations often contain mechanisms to automatically use coarser granularity on loaded processors. The scenario of fizzling sparks discussed in Section 3.1 is such an example, because the work of a spark is performed by an already running computation. However further improvements can be obtained by explicitly controlling thread granularity, and in the context of the original strategies we developed a range of clustering techniques (?). This section adapts these techniques for the new strategies and extends them.

One obvious way to obtain a coarser granularity is to collect computations on related elements of a data structure in “clusters.” The `evalClusteredBy` function, defined below, takes two function arguments for splitting the data into clusters (`cluster`) and for merging it again (`decluster`). This function comes with the following proof obligation: `decluster . cluster == id`.

```
evalClusterBy :: (a -> b) -> (b -> a) -> Strategy b -> Strategy a
nevalClusterBy cluster decluster strat x
  = return (decluster (cluster x 'using' strat))
```

While such explicit clustering is very flexible, it is also intrusive in always having to specify functions that are only used for tuning the parallel performance. To simplify this interface, we define a class `Cluster` containing `cluster` and `decluster` functions, as well as a function `lift` that turns an operation over the original data structure into one over such a clustered data structure. By building on the `Traversable` class we get several operations for free. Indirectly through the `Functor` class, we can use the `fmap` function to lift an operation over the base type to one in the cluster type. Again indirectly through the `Foldable` class, we can use the `fold` function as the default definition for `decluster`. Finally, we can define a function `evalCluster`, which hides the application of clustering and declustering and which can be applied to any data structure that is an instance of `Traversable`.

**SM**: a minor comment, but we should restrict the context to just the essential superclasses, i.e. `Functor`, `Foldable`. **HWL**: Agreed; todo: change Clustering and apps

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster :: Int -> a -> c a
  decluster :: c a -> a
  lift :: (a -> b) -> c a -> c b

  lift = fmap -- c is a Functor, via Traversable
  decluster = fold -- c is Foldable, via Traversable

  -- we require: decluster . cluster n == id
```

As an example we provide an instance for lists. Notably, we only have to provide a definition for the \texttt{cluster} function in the class.

```
-- instance for lists with clustering
instance Cluster [a] [] where
    cluster = chunk
```

We want to specify the data structure, used for clustering, only through the type with the possibility of using different clusterings for different applications. However, the \texttt{evalCluster} function, just like \texttt{evalClusterBy}, intentionally hides the cluster type. In order to expose the cluster type constructor \texttt{c} in the type of an \texttt{evalCluster} strategy, we make use of GADTs and introduce the following wrapper, enabling the type system to infer \texttt{c} when \texttt{evalCluster} is used.

```
data ClustStrat :: for all c a . Cluster c a => Strategy a -> ClusterStrategy c a
```

```
evalCluster :: forall a c . Cluster c a => Int -> ClustStrat c a -> Strategy a
```

```
evalCluster n (ClustStrat strat) =
    evalClusterBy (cluster n) decluster stratC
where
    stratC = evalTraversable strat :: Strategy (c a)
```

With this infrastructure we can now define a \texttt{parMapCluster} function that hides the calls to the clustering functions, but is still generic in the cluster type. The latter is specified explicitly when calling \texttt{parMapCluster} by instantiating its first argument to e.g. \texttt{ClustStrat} \[]. \texttt{Int}. We call this implicit clustering.

```
parMapCluster :: forall c b a .
    ClusterStrategy c [b] -> Int ->
    (a -> b) -> [a] -> [b]
```

```
parMapCluster (ClustStrat strat) z f xs = map f xs
    'using' evalCluster z (ClustStrat (rpar 'dot' strat)
    :: ClusterStrategy c [b])
```

### 5.3 A Divide-and-conquer Pattern

One of the main strengths of strategies is the possibility of constructing abstractions over patterns of parallel computation. Thereby all code specifying the coordination of the program is confined to the pattern. Concrete applications can then instantiate the function parameters to get parallel execution for free. Such patterns are commonly known as algorithmic skeletons (?).

As an example we give the implementation of a divide-and-conquer pattern. It is parameterised by a function that specifies the operation to be applied on atomic arguments (f), a function to (potentially) divide the argument into two smaller values (divide), and a function to combine the results from the recursive calls (conquer). Additionally, we provide a function \texttt{threshold} that is used to limit the amount of parallelism, by using a sequential strategy for arguments below the threshold.

**SM: cleaned up a bit. do we really need to be strict when threshold is reached? HWL: I thought we agree to use the ... return (conquer l r) version above?**

```
divConq :: (a -> b) -- compute the result
    -> a -- the value
    -> (a -> Bob) -- par threshold reached?
    -> (b -> b -> b) -- combine results
    -> (a -> Maybe (a,a)) -- divide
    -> b
```

```
divConq f arg threshold conquer divide = go arg
where
go arg =
    case divide arg of
        Nothing -> f arg
        Just (l0,r0) -> conquer l1 r1 'using' strat
where
    l1 = go l0
    r1 = go r0
    strat x = do r l1; r rl; return x
    where r | threshold arg = rseq
                | otherwise = rpar
```

All coordination aspects of the function are encoded in the strategy \texttt{strat}, which describes how the two subcomputations \texttt{l1} and \texttt{l2} should be evaluated. The thresholding predicate \texttt{threshold} provided by the caller places a bound on the depth of parallelism, and this is used by \texttt{strat} to decide whether to spark both \texttt{l1} and \texttt{l2} or to evaluate them directly. The definition of \texttt{divConq} achieves separation between the specifications of algorithm and parallelism, the latter being confined entirely to the definition of \texttt{strat}.

### 5.4 Improving Safety

**HWL: Maybe call these SafeStrategies to make clear that this is not what's currently implemented**

**HWL: Could be shortened, IMHO**

The original strategy type \texttt{a ->} () embodies the key modularity goal of separating computation and coordination. As any original strategy can only ever return (), it can never change the result of a computation, up to divergence. Unfortunately, the new strategy type gives up this type safety. Strategies of the new \texttt{a -> Eval a} type should be identity functions, i.e. only evaluate their argument but never change its value, and we term this property \texttt{identity safety}. However the type system cannot enforce this behaviour and it is all too easy to accidentally write flawed strategies, for instance:

```
x:xs 'using' \_ \_ \_ \_ -> parList rdeepseq xs
```

The intention of the programmer is to evaluate the tail of the list in parallel when the list is demanded. The strategy will do that, but then returns only the tail of the list.

Type checked identity safety for programmers can be restored for programmers who are willing to restricted themselves to a set of pre-defined and trusted strategy combinators. The idea, not currently implemented, is to make the strategy type abstract by wrapping it with a \texttt{newtype} constructor \texttt{S}, and to provide a destructor ($\$\$) that unwarps a strategy returning a function and is used like function application $$. The \texttt{SafeStrategy} module is adapted to use \texttt{S} and $$\$\$ in the obvious way.

```
newtype SafeStrategy a = S (a -> Eval a)
($\$\$) :: SafeStrategy a -> a -> Eval a
(S s) $$\$\$$ x = s x
```

Both the strategy constructor and destructor are exported from the \texttt{SafeStrategies} module, and programmers are free to trade expressive power for type safety by choosing whether to import these or to leave the type \texttt{SafeStrategy} abstract. If neither \texttt{S} nor $$\$\$$ are imported then programmers only gain access to a restricted Strategy interface. Strategies can be built only with the predefined identity-preserving strategy combinators, and applied only with \texttt{using}. If $$\$\$$ is imported without \texttt{S} then programmers gain access to a richer, yet still restricted interface. They cannot write their own strategies, so the type checker would reject the flawed strategy above, but they can use the \texttt{Eval} monad directly and apply safe
strategies with $$$. If both $$ and $$ are imported then the full strategies interface is available.

**PM**: Is there a better name for the "identity safety" property?  
**HWL**: I like "identity safety"; suggest to introduced it earlier in the paper PWT: Added to introduction

6. Evaluation

This section discusses our measurements in detail, but first we summarise the key results:

- For all programs, the speedup and runtime results with original and new strategies are very similar, giving us confidence that they specify the same parallel coordination for a range of programs and parallel paradigms (Figure 2).
- The speedups achieved with the new strategies are slightly better compared to those with the original strategies: a mean of 3.85 versus 3.72 across all applications (Columns 3 & 2 of Table 2).
- The new strategies fix the space leak outlined in Section 6.4 and better support speculative parallelism (Section 6.4).
- The overheads of the new strategies are low: mean sequential run-time overhead is 1.91% (Table 1), and memory overheads are low for most programs (Columns 8 – 11 of Table 2).

6.1 Apparatus

Our measurements are made on an eight-core, 8GB RAM, HP XW6600 Workstation comprising two Intel Xeon 5410 quad-core processors, each running at 2.33GHz. The benchmarks run under Linux Fedora 7 using recent development GHC snapshot (6.13 as of 20.5.2010), and parallel packages 1.1.0.1 and 2.3.0.0, for original and new strategies respectively. The data points reported are the median of 3 executions, and we measure up to 7 cores as measurements on the 8th core are known to introduce some variability.

**HWL**: Shorten

Our benchmarks are 10 parallel applications from a range of application areas; some have previously been studied (?) and others are taken from the GHC nofib suite and parallelised (?). The programs are the computational kernels of realistic applications, cover a variety of parallel paradigms, and employ several important parallel programming techniques, such as thresholding to limit the amount of parallelism generated, and clustering to obtain coarser thread granularity.

**HWL**: TODO: take missing descriptions from (?)

*Genetic* aligns RNA sequences from related organisms, using divide-and-conquer parallelism and (nested) data parallelism. *MinMax* performs an alpha-beta search in a tree representing positions in a 2-player game. The program is divide-and-conquer style and laziness is exploited to prune unnecessary subtrees. *Queens*, solving the well-known n-queens problem, is implemented using divide-and-conquer parallelism with an explicit threshold. *LinSolv* finds an exact solution to a set of linear equations, employing the data parallel multiple homomorphic images approach often used in symbolic computation. *Hidden* performs hidden-line removal in 3D rendering and uses data parallelism via the *parList* strategy. *Maze* searches for a path in a 2D maze and uses speculative data parallelism. *Sphere* is a ray-tracer for the Haskell nofib suite, using nested data parallelism, implemented as *parMaps*. *TransClos* finds all elements that are reachable via a given relation from a given set of seed values, i.e. that are in the range of the transitive closure of the given relation. The algorithm uses a queue-based *parBuffer* over an infinite list. *Coins* computes the number of ways of paying the given value from a given set of coins, using a divide-and-conquer paradigm. *MatMult* performs matrix multiplication using data parallelism with explicit clustering.

**note**: All raw results are placed in /u1/pg/mka19/res24042010

### Table 1. Sequential Runtime Overheads HWL: recompute without spect-fib

<table>
<thead>
<tr>
<th>Program</th>
<th>Sequential Runtime (seconds)</th>
<th>Original Strategies</th>
<th>New Strategies</th>
<th>Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinSolv</td>
<td>23.40</td>
<td>+0.90</td>
<td>+1.97</td>
<td>Nested Data par</td>
</tr>
<tr>
<td>Sphere</td>
<td>21.11</td>
<td>+4.78</td>
<td>+3.32</td>
<td>Nested Data par</td>
</tr>
<tr>
<td>MinMax</td>
<td>36.98</td>
<td>-0.87</td>
<td>+3.22</td>
<td>D&amp;C</td>
</tr>
<tr>
<td>Coins</td>
<td>42.39</td>
<td>+1.11</td>
<td>+2.12</td>
<td>D&amp;C</td>
</tr>
<tr>
<td>Queens</td>
<td>25.51</td>
<td>+1.37</td>
<td>+6.12</td>
<td>D&amp;C</td>
</tr>
<tr>
<td>Genetic</td>
<td>33.46</td>
<td>+2.96</td>
<td>+3.97</td>
<td>D&amp;C Data par</td>
</tr>
<tr>
<td>MatMult</td>
<td>38.48</td>
<td>-1.35</td>
<td>-2.06</td>
<td>Data par</td>
</tr>
<tr>
<td>Hidden</td>
<td>41.49</td>
<td>+8.41</td>
<td>+2.70</td>
<td>Data par</td>
</tr>
<tr>
<td>Maze</td>
<td>40.93</td>
<td>-2.22</td>
<td>-3.59</td>
<td>Nested Data par</td>
</tr>
<tr>
<td>TransClos</td>
<td>83.13</td>
<td>+0.75</td>
<td>+1.68</td>
<td>Data par</td>
</tr>
</tbody>
</table>

**Geom. Mean** +1.72 +1.91

6.2 Sequential Overhead

Table 1 shows the sequential runtime as baseline, and the difference of the 1 processor runtime with both original and new strategies. For the new strategies, we encounter a runtime overhead of at most +6.12% for the divide-and-conquer style *Queens* implementation. Notably, the data parallel programs have a fairly low overhead, despite the additional traversal of a data structure to expose parallelism. Comparing the geometric mean of the runtime overheads imposed by both strategies versions we encounter only a slightly higher overhead for the new strategies: +1.91% compared to +1.72% with the original strategies. This justifies the new strategy approach of high-level generic abstractions.

**HWL**: discuss

6.3 Parallel Performance

**Speedups**: Figure 2 compares the absolute speedup curves (i.e. speedup relative to sequential runtime) for the applications with the original and new strategies. Both the runtime curves (not reported here) and speedup curves for the original and new strategies are very similar. The pattern is repeated in more detailed analysis, e.g. in Columns 2 and 3 of Table 2. We conclude that the original and new strategies specify the same parallel coordination for a variety of programs representing a range of parallel paradigms, and several tuning techniques.

The top six programs in Table 2 have been carefully tuned for parallelism, and hence are most relevant when assessing the performance of the new strategies. The mean speedups of these programs are 5.38 for the original and 5.59 for the new strategies. The remaining applications have potential for additional performance tuning, and yet none has a significantly lower speedup with the new strategies.

**HWL**: To separate algorithmic from system issues impacting scalability, we should also use the constructed examples here

**Performance**: Table 2 analyses in detail the speedups, number of sparks and memory consumption of all applications, running on 7 cores of an 8 core machine with the original strategies and the new strategies. The number of generated sparks was in all cases virtually identical between original strategies and new strategies,
The comparison of generated versus converted sparks in Table 2 demonstrates the runtime system’s effective handling of potential parallelism (sparks). Even when an excessive number of sparks is generated, for example in Coins, the runtime-system converts only a small fraction of these sparks. As with any divide-and-conquer program, a thread generated for a computation close to the root will itself evaluate potential child computations, causing their corresponding sparks to fizzle. Hence the granularity of the generated sparks is automatically made coarser, and reducing overheads, as can be seen from the speedups achieved. In general, the new strategies provide more opportunities for sparks to fizzle, as discussed in Section 3. This shows up in a lower number of converted sparks for all divide-and-conquer and nested data parallel programs. For single-level data parallelism as in Sphere, where sparks never share graph structures, there is little or no reduction in the number of converted sparks.

### 6.4 Memory Management

**Fixing the space leak:** The new strategies fix the space leak outlined in Section 3. For example, the parallel raytracer reported as a GHC bug for exhausting memory now terminates successfully. Although this improvement is crucial for execution on small numbers of cores, the heap measurements in Table 2 do not show a consistent reduction in residency for the new strategies on 7 cores. Interpreting the parallel memory consumption figures is complicated.
by a number of factors: the garbage sparks are often evaluated by other cores and hence do not create a space leak on a single core; changing the pattern of parallel execution changes residency; and residency is recorded by sampling and hence is very sensitive to small program changes.

SM: Where are the results for Spect-Pfib? SM: Was this with WEAK or ROOT? Only WEAK will prune sparks. SM: We haven’t said what converted sparks are.

Speculation: HWL: primes results are up-to-date but for a very short run: hence poor speedups

To assess the effectiveness of the WEAK and ROOT garbage collection policies, described in Section 5, for managing speculation we use a program that applies drop to a parallelised list, computing the number of primes up to a given value, thereby rendering the sparks on the dropped list elements speculative:

\[
\text{sum } \map \text{ and } \\text{drop } \((m_1-m_0) \cdot \text{quot}
2\) \text{ } \(([(n, \text{length } (\text{primes } n)) \mid n < [m_0..m_1]] \ \text{using} \ \text{parList rdeepseq})
\]

With the WEAK policy almost all sparks of the original strategies are discarded, as expected. With the new strategies 3195 out of 10001 are converted, 36% fewer than with the ROOT policy, although this value changes considerably between executions. Most importantly, the WEAK policy prunes 4998 sparks, almost all of the 5000 speculative sparks. In contrast, the ROOT policy prunes only 3202 sparks, all of them due to fizzling. For the small input used with this program, the new strategies achieve a speedup of 1.3 on 7 cores, whereas with the original strategies starvation leads to a slowdown. The memory residency with WEAK and ROOT policies is about the same because the generated parallelism is not very heap intensive.

Our application kernels do not use substantial speculation and as a result speedups with the WEAK policy are only slightly, but consistently, better than with the ROOT policy (MiniMax, Maze). Of course, the very inability of reclaiming speculative sparks with the ROOT policy discouraged any applications using them on a larger scale.

HWL: how much? HWL: Data on all programs?

7. Related Work

HWL: A lot of self-references; page numbers missing in bib file

Most parallel functional languages combine high level coordination sublanguages with their high level computation language (?). A range of high level coordination models have been used (?), and this section relates the semi-explicit approach adopted by evaluation strategies to other approaches.

Skeleton based coordination, as in (?), is popular in both imperative and functional languages, and exploits a small set of predefined skeletons. Each skeleton is a polymorphic higher-order function describing a common coordination pattern with an efficient parallel implementation (?). As polymorphic higher-order functions, evaluation strategies are similar to skeletons, but there are some key differences. Rather than a fixed set of skeletons, evaluation strategies are readily combined to form new strategies. Moreover, where skeletons are parameterised with computational arguments, a strategy is typically applied to a computation.

Data parallel coordination, as in (?), supports the parallel evaluation of every element in a collection. This is a good match with Haskell’s powerful constructs for bulk data types, in particular lists. Data parallelism is often more implicit than HWL: control parallelism with evaluation strategies: the programmer simply identifies

the collections to be evaluated in parallel. Strategies are more general in that they can express both control parallelism and data parallelism, although in terms of performance Data Parallel Haskell is designed to compile parallel programs down to highly optimised low-level loops over arrays, and hence should achieve significantly better absolute performance on data-parallel programs than would be possible using strategies.

Entirely implicit coordination aims to minimise programmer input, typically using either profiling as in (?) or parallel iteration as in (?). Few entirely implicit approaches, other than parallel iteration have delivered acceptable performance (?). Evaluation strategies provide more general parallel coordination than loop parallelism.

HWL: phps cite Arvind’s pH Book (?)

8. Conclusion

The original strategies were developed in 1996 for Haskell 1.2, i.e. before monads, and using a compiler with relatively tame optimisations. The context for the new strategies is radically different. Monads, supported by rich libraries and syntactic sugar like do notation, are now the preferred mechanism for sequencing computations, and are familiar to the rapidly growing Haskell user community. Applicative functors elegantly encode data structure traversals. Finally, the aggressive use of optimisations in mature Haskell implementations like GHC make bespoke efficiency specialisations unnecessary.

The new strategy formulation capitalises on improved Haskell idioms and implementations to provide a modular and compositional notation for specifying pure deterministic parallelism. While it has some minor drawbacks: being relatively complex, providing relatively weak type safety, and requiring care to express control parallelism, the advantages are many and substantial. It provides clear, generic, and efficient specification of parallelism with low runtime overheads. It resolves a subtle space management issue associated with parallelism, better supports speculation, and is able to directly express parallelism embedded within lazy data structures.

We plan to further enhance and formalise the identity safety of the new strategies, following the direction discussed in Section 5.4. Moreover the genericity of the new strategies could be improved by automatically deriving instances of the NFData class, as for the Traversable class.

Acknowledgments

Thanks to Greg Michaelson and Simon Peyton Jones for constructive feedback. This research is supported by the EPSRC HPC-GAP project (EP/G05553X), and the EU FP6 SCIEnce project (RII3-CT-2005-026133).