

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 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 now prunes unneeded speculation. 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 (Trinder et al. 1998), 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 (Loidl et al. 1999; Trinder et al. 2002). Strategies have been used for some 15 years in a number of parallel variants of Haskell (Harris et al. 2005; Loogen et al. 2005; Trinder et al. 1998).

This paper presents a complete redesign of the strategy abstraction. 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:

- 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 (McBride and Paterson 2008), 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 3). 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

[Copyright notice will appear here once ‘preprint’ option is removed.]

used compositionally. These drawbacks are resolved by the new framework (Section 5.1).

- Motivated by wanting to have different versions of `par` to control locality in large architectures, the new formulation allows for abstracting over the coordination combinator used (Section 4.4).

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

- There is some extra complexity in the implementation 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 (`'using' s`) is always an identity function for any Strategy `s`. 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` package¹, and the version we describe in this paper will be released as version 2.3. 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, `par` and `pseq`, with the following types²:

```
par  :: a -> b -> b
pseq :: a -> b -> b
```

The `par` combinator introduces a potential for parallel evaluation. When `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 `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 `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 \perp , 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 `par a b` is

¹<http://hackage.haskell.org/package/parallel>

²The original presentation used `seq` rather than `pseq` (Trinder et al. 1998); however, Haskell later adopted a `seq` operator but without the order-of-evaluation property required for parallel execution (Marlow et al. 2009). Hence, to avoid confusion with Haskell’s `seq`, we now use `pseq` for expressing sequential ordering of evaluation.

always `b`, regardless of the value of `a`, \perp or otherwise. In practice, this isn’t a problem for typical Haskell implementations, as a lazy computation can already have value \perp .

It is not enough to provide `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 `par` will be evaluated, or what will be evaluated before or after it (or indeed in parallel with it). This is the reason for `pseq`: a call `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. The denotational semantics of `pseq` are

$$\begin{aligned} \text{pseq } a \ b &= \perp, & \text{if } a &= \perp \\ &= b, & \text{otherwise} \end{aligned}$$

and the operational semantics are that `a` must be evaluated to weak head normal form before `b` is evaluated (Baker-Finch et al. 2000).

An example to illustrate the usage of `par` and `pseq` follows, using the traditional Fibonacci function. More examples can be found in the literature (Jones Jr. et al. 2009; Trinder et al. 1998).

```
fib :: Int -> Int
fib n
  | n <= 1    = 1
  | otherwise = let
                    x = fib (n-1)
                    y = fib (n-2)
                in
                    x 'par' (y 'pseq' x + y + 1)
```

The Fibonacci computation is shaped like a binary tree. At each node of the computation we combine `par` and `pseq` to evaluate one branch in parallel with the other branch. The pattern here is a common one: `in x 'par' (y '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 `pseq` is used to control *evaluation order*.

The parallelism here is independent of the number of processors; every time `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 `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 `par` and `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 -> ()` for some `a`:

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

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

Some basic strategies can be defined as follows.

```

r0 :: Strategy a
r0 x = ()

rwhnf :: Strategy a
rwhnf x = x 'pseq' ()

rnf :: NFData a => Strategy a
-- rnf is a method in the class NFData

```

`r0` is a strategy that evaluates nothing of its argument, `rwhnf` evaluates its argument to weak-head normal form, and `rnf` evaluates its argument completely. The definition of `rnf` 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`).

Strategies are applied with the `using` combinator:

```

using :: a -> Strategy a -> a
using x s = s x 'pseq' x

```

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

```

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

```

The function `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, `parList rwhnf` causes each spark to evaluate its list element as far as the top-level constructor, whereas `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 `parList` function can also be used to illustrate the modular nature of strategies; for example,

```

parMap strat f xs = map f xs 'using' parList strat

```

The `parMap` function takes a strategy `strat`, a function `f`, and a list `xs` as arguments and maps the function `f` over the list in parallel, applying `strat` to every element. Note how the construction of the result with `map`, on the left of `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 (Loidl et al. 1999).

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 (Marlow et al. 2009).

To understand the problem we need to consider how `par` is implemented. When the Haskell program evaluates the expression `par a b`, the runtime system saves a pointer to the heap node representing `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 `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 (Mohr et al. 1990). More details on the implementation of spark pools can be found in Marlow et al. (2009); 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 `ROOT` and `WEAK` respectively, following the terminology of Marlow et al. (2009):

1. **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. **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 `WEAK`, and examine how it works with the definition of `parList` in the previous section. The sparks created by `parList` are all expressions of the form `(strat x)` for some strategy `strat` applied to some list element `x`. Now, every such expression is uniquely allocated for the sole purpose of being passed to `par`; the spark pool will contain references to many expressions of the form `(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 `WEAK` policy then every spark created by `parList` will be discarded by the garbage collector, and we lose all the parallelism.

Moreover, there is no definition of `parList` that can avoid this problem. The only value that the `parList` strategy can return is `()`, so the only way that `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 `parList` that works:

```

parListWHNF :: Strategy [a]
parListWHNF [] = ()
parListWHNF (x:xs) = x 'par' parListWHNF strat xs

```

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, `ROOT`, where we treat the spark pool as a source of roots? Considering the `parList` example again, the spark pool would still contain references to expressions of the form `(strat x)` in the heap, but this time all the expressions will be retained by the garbage collector, and no parallelism is lost. However, another problem arises: what happens when there are not enough parallel processors to evaluate all the sparks? The spark pool retains references to all the `(strat x)` expressions, perhaps long after each `x` is no longer

required by the program and would otherwise be reclaimed by the garbage collector.

In an attempt to retain potential parallelism, the storage manager is retaining memory that should have been released: this is a space leak, and can and does have dramatic performance implications³. Even an innocuous `parList` or `parMap` can turn a program that ran in constant space into one that requires linear heap. The adverse effects tend to manifest when running parallel programs on a single processor, because there are no spare processors to evaluate the sparks and hence allow them to be removed from the spark pool. However, effects are felt even when multiple processors are available: the garbage sparks occupy space in the spark pool that could be used for real parallelism, and processors waste time evaluating garbage sparks which erodes the overall speedup achieved.

It is tempting to think that perhaps we can solve the space leak by only retaining sparks that share some data with the main program. This is difficult to achieve, however, and in any case it is not clear that it would be a robust solution to the problem: how *much* data should be shared before we consider the spark to be alive?

3.1 Fizzled sparks

It is possible that a spark in the spark pool can refer to a computation that has already been evaluated by the program. Perhaps there were not enough processors to evaluate the spark in parallel, and another thread ended up evaluating the computation during the normal course of computing its results.

When a spark in the spark pool refers to a *value*, rather than an unevaluated computation, we say the spark has *fizzled*: this potential for parallel execution has expired (Marlow et al. 2009). The runtime system can, and should, remove fizzled sparks from the spark pool so that the storage manager can release the memory they refer to, to avoid the mutator wasting time evaluating useless sparks, and to make more room for real potential parallelism in the spark pool.

This is all well and good, but note that in the original strategies formulation, *most sparks will never fizzle* because they are expressions of the form `(strat x)` that are unshared and hence can never be evaluated by the main program. In contrast, the sparks generated by the simpler non-compositional operation `parListWHNF` above can fizzle, because in that case `par` is applied directly to a part of the data structure, rather than to a new unshared expression, and presumably the main program will proceed by evaluating the same data structure itself.

3.2 Speculative parallelism

Sparking ought to support *speculative* parallelism, by which we mean sparking an expression whose value is not known for certain to be eventually required by the computation as a whole. Ideally, speculative parallelism should be automatically *pruned* by the system when it can be proven to be never needed.

Speculative parallelism can be created using `par`; the question is whether speculative sparks are ever discarded. Under the `ROOT` policy, a speculative spark that is never evaluated will become a space leak, whereas under the `WEAK` policy unreachable speculative sparks will be discarded and their heap reclaimed. In short, only the `WEAK` policy supports speculation.

³in fact, one unhappy user of GHC even reported this behaviour as a bug (ticket 2185).

3.3 Summary

The following table summarises the interaction between the choice of GC policy (`ROOT` or `WEAK`), original strategies (Section 2) or new strategies (Section 4), and speculative versus non-speculative parallelism.

Strategies	Parallelism	ROOT	WEAK
Original	non-speculative	space leaks	lost parallelism
Original	speculative	space leaks	lost parallelism
New	non-speculative	OK	OK
New	speculative	space leaks	OK

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 fizzles; 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 'pseq' Done x

rpar :: Strategy a
rpar x = x 'par' Done x

rdeepseq :: MData a => Strategy a
rdeepseq x = rnf x 'pseq' Done x
```

The new basic strategies `r0`, `rseq` and `rdeepseq` are analogues to the original strategies `r0`, `rwhnf` and `rnf` respectively (in fact, `rdeepseq` uses the original `rnf`).

<pre> type Strategy a = a -> () using :: a -> Strategy a -> a x 'using' s = s x 'pseq' x r0 :: Strategy a r0 x = () rwhnf :: Strategy a rwhnf x = x 'pseq' () rnf :: NFData a => Strategy a -- rnf is a method in the class NFData seqList :: Strategy a -> Strategy [a] seqList s [] = () seqList s (x:xs) = s x 'pseq' (seqList s xs) parList :: Strategy a -> Strategy [a] parList s [] = () parList s (x:xs) = s x 'par' (parList s xs) </pre>	<pre> data Eval a = Done a instance Monad Eval where return x = Done x Done x >>= k = k x runEval :: Eval a -> a runEval (Done x) = x type Strategy a = a -> Eval a using :: a -> Strategy a -> a x 'using' s = runEval (s x) dot :: Strategy a -> Strategy a -> Strategy a s2 'dot' s1 = s2 . runEval . s1 r0 :: Strategy a r0 x = return x rseq :: Strategy a rseq x = x 'pseq' return x rdeepseq :: NFData a => Strategy a rdeepseq x = rnf x 'pseq' return x rpar :: Strategy a rpar x = x 'par' return x evalList :: Strategy a -> Strategy [a] evalList s [] = return [] evalList s (x:xs) = do x' <- s x; xs' <- evalList s xs; return (x':xs') parList :: Strategy a -> Strategy [a] parList s = evalList (rpar 'dot' s) </pre>
--	---

Figure 1. Like-for-like comparison of original strategies (left column) versus new strategies (right column).

4.1 The Evaluation-order Monad

We can declare `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:

```

instance Monad Eval where
    return x = Done x
    Done x >>= k = k x

```

The strict identity monad⁴ gives us a convenient and flexible notation for expressing evaluation order, i.e. the ordering between applications of `rseq` and `rpar`, which is exactly what we need for expressing basic parallel evaluation. For example, the following fragment of `fib`:

```

let
    x = fib (n-1)
    y = fib (n-2)
in
    x 'par' (y 'pseq' x + y + 1)

```

can be rewritten as

⁴this is in fact isomorphic to the `Lift` monad in the `MonadLib` package, <http://hackage.haskell.org/packages/archive/monadLib/3.6.1/doc/html/src/MonadLib.html>

```

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

```

which clearly expresses the ordering between `rpar` and `rseq`, using a notation that Haskell programmers will find familiar.

Programmers using the new strategies API no longer need to use `par` and `pseq` to construct new strategies, instead they use the `Eval` monad with `rpar` and `rseq`. The `Eval` monad raises the level of abstraction for `pseq` and `par`; it makes fragments of evaluation-order first class, and lets us compose them together. We should think of the `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 `return` in place of `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` (McBride and Paterson 2008):

```
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 (s 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 `s` 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 (McBride and Paterson 2008). 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 s = 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 Sections 4.1:

```
runEval $ do
  x <- rpar (fib (n-1))
  y <- rseq (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:

```
x + y + 1 'using' strat
where
  x = fib (n-1)
  y = fib (n-2)
  strat v = runEval $ do rpar x; rseq 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 embedded in the result returned, otherwise the spark is likely to 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:

```
forceList = evalList rseq
```

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-recursive⁵.

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:

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

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.

```
ins :: SeqStrategy a -> Strategy a
ins ss x = ss x 'pseq' return x
```

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

⁵one would typically not use `parList` on long lists as too many sparks would be created, instead `parBuffer` tends to be more practical.

```
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 combinators may use the ordinary Haskell `seq` operator instead of `pseq`, granting the compiler more freedom to optimise the order of evaluation. In contrast, 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`.

```
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`):

```
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`.

4.8 Fusion

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`:

```
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 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

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 (Trinder et al. 1998):

```
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 $i + n$ 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 parameterised 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 `Eval` 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 :: Int -> Strategy a -> Strategy [a]
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)
```

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 (Loidl et al. 2001). 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 `evalClusterBy` 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
evalClusterBy 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`.

```
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 `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 `evalCluster` function, just like `evalClusterBy`, intentionally hides the cluster type. In order to expose the cluster type constructor `c` in the type of an `evalCluster` strategy, we make use of GADTs and introduce the following wrapper, enabling the type system to infer `c` when `evalCluster` is used.

```
data ClusterStrategy c a where
  ClustStrat :: forall c a . Cluster c a =>
              Strategy a -> ClusterStrategy c a

evalCluster :: forall a c . Cluster c a =>
              Int -> ClusterStrategy 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 `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 `parMapCluster` by instantiating its first argument to e.g. `ClusterStrategy [] [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 (Cole 1988).

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 `threshold` that is used to limit the amount of parallelism, by using a sequential strategy for arguments below the threshold.

```
divConq :: (a -> b)           -- compute the result
-> a                          -- the value
-> (a -> Bool)                 -- 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 r1; return x
          where r | threshold arg = rseq
                | otherwise      = rpar
```

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

5.4 Improving Safety

The original strategy type `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 `a -> Eval a` type should be identity functions, i.e. only evaluate their argument but never change its value, and we term this property *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 restrict 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 `newtype` constructor `S`, and to provide a destructor (`$$$`) that unwraps a strategy returning a function and is used like function application `$`. The `SafeStrategy` module is adapted to use `S` and `$$$` 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 `SafeStrategies` module, and programmers are free to trade expressive power for type safety by choosing whether to import these or to leave the type `SafeStrategy` abstract. If neither `S` nor `$$$` are imported then programmers only gain access to a restricted Strategy interface. Strategies can be built only with the predefined identity-preserving strategy combinators, and applied only with `using`. If `$$$` is imported without `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 `Eval` monad directly and apply safe strategies with `$$$`. If both `S` and `$$$` are imported then the full strategies interface is available.

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 3, 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

Program	Sequential Runtime (seconds)	Δ Time (%)		
		Original Strategies	New Strategies	Paradigm
<i>LinSolv</i>	23.40	+0.90	+1.97	Nested Data par
<i>Sphere</i>	21.11	+4.78	+3.32	Nested Data par
<i>MiniMax</i>	36.98	+0.87	+3.22	D&C
<i>Coins</i>	42.49	+1.11	+2.12	D&C
<i>Queens</i>	25.51	+1.37	+6.12	D&C
<i>Genetic</i>	33.46	+2.96	+3.97	D&C Data par
<i>MatMult</i>	35.48	-1.35	-2.06	Data par
<i>Hidden</i>	41.49	+8.41	+2.70	Data par
<i>Maze</i>	40.93	-2.22	-3.59	Nested Data par
<i>TransClos</i>	83.13	+0.75	+1.68	Data par
Geom. Mean		+1.72	+1.91	

Table 1. Sequential Runtime Overheads

Linux Fedora 7 using a 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.

Our benchmarks are 10 parallel applications from a range of application areas; some have previously been studied (Loidl et al. 1999) and others are taken from the GHC nofib suite and parallelised (Aswad et al. 2009). 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.

Genetic aligns RNA sequences from related organisms, using divide-and-conquer parallelism and (nested) data parallelism. *MiniMax* 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 from 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.

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.

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.

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, giving us further confidence that the two formulations are expressing the same parallelism. Small differences in the number of generated sparks arise because GHC has a non-deterministic execution model in which a particular expression may be evaluated multiple times at runtime (Harris et al. 2005).

In the cases where the new strategies exhibit poorer performance, the reduction in speedup is still very small: from 5.67 to 5.48 in the worst case for *MiniMax*. This reflects the low overhead associated with the new strategies, quantified in the previous sub-section.

Interestingly, the performance of the new strategies in the *Queens* and *Sphere* programs is better than in the original strategies. Examining the heap consumption reveals that with the new strategies the heap residency is significantly reduced: -24.11% for *Queens* and -24.80% for *Sphere*. This results in a lower total garbage collection time, which contributes to about half of the reduction in runtime. The reduction in residency is accounted for by the improved space behaviour of the new strategies: the space retained by superfluous sparks is being reclaimed.

Granularity improvement: 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.

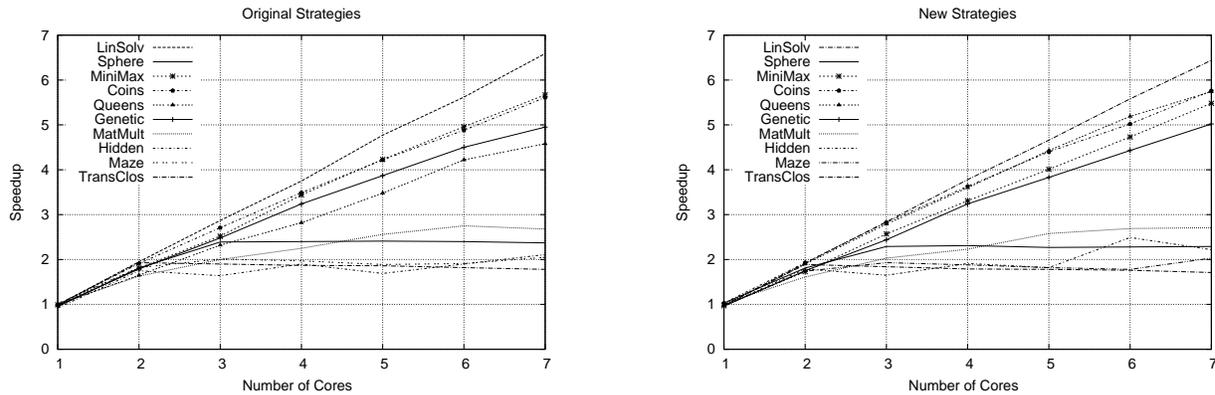


Figure 2. Speedups of the Application Kernels with Original and New Strategies

	Speedup		Generated Sparks		Converted Sparks		Allocated Heap		Maximum Residency	
	Orig.	New	Orig.	New	Orig.	New	Orig. (MB)	New $\Delta\%$	Orig. (KB)	New $\Delta\%$
<i>LinSolv</i>	6.59	6.44	7562	7562	7562	7562	6050.10	+0.15	7104.70	+3.87
<i>Sphere</i>	5.09	5.75	160	160	160	159	8632.30	-1.11	142303.30	-24.80
<i>MiniMax</i>	5.67	5.48	1464	1464	1464	163	30476.85	-0.01	98.05	-7.17
<i>Coins</i>	5.61	5.53	145925	146853	2702	1060	79833.20	+1.59	302.10	+20.36
<i>Queens</i>	4.58	5.49	1589	1563	1589	636	14903.30	-17.52	19134.50	-24.11
<i>Genetic</i>	4.95	4.94	659	672	659	172	12180.20	-6.73	493.90	+35.37
<i>MatMult</i>	2.68	2.71	30	30	30	29	13725.50	-2.93	31950.90	+0.00
<i>Hidden</i>	2.11	2.54	972	972	972	972	42908.40	-0.22	14325.00	-4.11
<i>Maze</i>	2.05	2.01	2723	2835	2525	481	194122.00	+7.74	71.20	-33.15
<i>TransClos</i>	1.80	1.72	1035	1035	1035	986	80039.40	-0.06	86.50	+11.91
Geom. Mean	3.72	3.85								

Table 2. Speedups, Number of Sparks and Heap Consumption on 7 Cores

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 memory⁶ 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.

Speculation: To assess the effectiveness of the WEAK and ROOT garbage collection policies, described in Section 3, 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:

```
sum $ map snd $ drop ((m1-m0) 'quot' 2) $
  [(n, length (primes n)) | n <- [m0..m1] ]
  'using' parList rdeepseq
```

⁶<http://hackage.haskell.org/trac/ghc/ticket/2185>

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.

7. Related Work

Most parallel functional languages combine high level coordination sublanguages with their high level computation language (Hammond and Michaelson 1999). A range of high level coordination models have been used (Trinder et al. 2002), and this section re-

lates the semi-explicit approach adopted by evaluation strategies to other approaches.

Skeleton based coordination, as in (Loogen et al. 2005; Michaelson et al. 2005), 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 (Cole 1988). 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 (Blelloch 1996; Chakravarty et al. 2007), 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 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 (Harris and Singh 2007) or parallel iteration as in (Grelck and Scholz 2003). Few entirely implicit approaches, other than parallel iteration have delivered acceptable performance (Nikhil and Arvind 2001). Evaluation strategies provide more general parallel coordination than loop parallelism.

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).

References

- M. Aswad, P.W. Trinder, A.D. Al Zain, G.J. Michaelson, and J. Berthold. Low Pain vs No Pain Multi-core Haskell. In *TFP09 — Symposium on Trends in Functional Programming*, Komarno, Slovakia, June 2009.
- C. Baker-Finch, D.J. King, J.G. Hall, and P.W. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *ICFP’00 — International Conference on Functional Programming*, pages 162–173, Montreal, Canada, September 2000. ACM Press.
- G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
- M.M.T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP’07 — Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.
- M.C. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988.
- C. Grelck and S.B. Scholz. SaC – from High-Level Programming with Arrays to Efficient Parallel Execution. *Parallel Processing Letters*, 13(3):401–412, 2003.
- K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
- T. Harris and S. Singh. Feedback Directed Implicit Parallelism. *SIGPLAN Not.*, 42(9):251–264, 2007. ISSN 0362-1340.
- T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell ’05 — ACM SIGPLAN Workshop on Haskell*, pages 49–61. ACM Press, September 2005. ISBN 1-59593-071-X.
- D. Jones Jr., S. Marlow, and S. Singh. Parallel Performance Tuning for Haskell. In *Haskell’09 — ACM SIGPLAN Symposium on Haskell*. ACM, 2009.
- H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency – Practice and Experience*, 11:701–752, 1999.
- H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4), 2001.
- R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari. Parallel functional programming in Eden. *J. of Functional Programming*, 15(3):431–475, 2005. ISSN 0956-7968.
- S. Marlow, S. Peyton Jones, and S. Singh. Runtime Support for Multicore Haskell. In *ICFP’09 — ACM SIGPLAN Intl. Conf. on Functional Programming*, August 2009.
- C. McBride and R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
- G. Michaelson, S. Horiguchi, N. Scaife, and P. Bristow. A Parallel SML Compiler based on Algorithmic Skeletons. *J. of Functional Programming*, 15(4):615–650, 2005.
- E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP’90 — Conference on Lisp and Functional Programming*, pages 185–197, Nice, France, June 27–29, 1990.
- R. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, May 2001. ISBN 1-55860-644-0.
- P.W. Trinder, K. Hammond, H-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.
- P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskell. *J. of Functional Programming*, 12(4–5):469–510, July 2002. Special Issue on Haskell.