Call-pattern Specialisation for Haskell Programs

Simon Peyton Jones
Microsoft Research, UK
simonpj@microsoft.com

Abstract

User-defined data types, pattern-matching, and recursion are ubiquitous features of Haskell programs. Sometimes a function is called with arguments that are statically known to be in constructor form, so that the work of pattern-matching is wasted. Even worse, the argument is sometimes freshly-allocated, only to be immediately decomposed by the function.

In this paper we describe a simple, modular transformation that specialises recursive functions according to their argument "shapes". We describe our implementation of this transformation in the Glasgow Haskell Compiler, and give measurements that demonstrate substantial performance improvements: a worthwhile 10% on average, with a factor of 10 in particular cases.

Categories and Subject Descriptors D.1.0 [*Programming techniques*]: Functional programming; D.3.3 [*Programming languages*]: Processors

General Terms Algorithms, Languages, Performance.

Keywords Haskell, compilers, optimisation, specialisation.

1. Introduction

Consider the following Haskell function definition:

```
last :: [a] -> a
last []       = error "last"
last (x : []) = x
last (x : xs) = last xs
```

The final equation is the heavily-used path. Annoyingly, though, this equation first establishes that `xs` is a cons, in order to exclude the second equation — and then calls `last` recursively on `xs`. The first thing `last` will do is to analyse `xs` to see whether it is a cons or nil, even though that fact is already known.

A programmer who worries about this might rewrite `last` as follows, so that there is no redundant pattern-matching:

```
last [] = error "last"
last (x:xs) = last' x xs
  where
    last' x []     = x
    last' x (y:ys) = last' y ys
```

ICFP'07, October 1–3, 2007, Freiburg, Germany.
Copyright © 2007 ACM 978-1-59593-815-2/07/0010...$5.00
Reprinted from ICFP'07,, [Unknown Proceedings], October 1–3, 2007, Freiburg, Germany., pp. 327–337.

Here `last'` is a specialised version of the original `last`, specialised for the case when the argument is a cons. The idea of specialising a function for particular argument "shapes" is a very general one, and is the subject of this paper. In particular, our contributions are these:

- We describe a simple, general program transformation that specialises functions based on the "shapes" of their arguments, or *call patterns* (Section 3). Since these shapes are constructor trees, we call it the `SpecConstr` transformation.

- The basic idea is simple enough, and easy to prove correct. However, to be effective, it must specialise the right functions in the right way, something that is governed by a set of heuristics (Section 3.3). In the light of our experience of using `SpecConstr` in practice, we have developed a series of non-obvious refinements to the basic heuristics (Section 4).

- We have implemented `SpecConstr` in GHC, a state-of-the art optimising compiler for Haskell. The implementation is very modular, consisting simply of a Core-to-Core transformation, and does not interact with any other part of the compiler. ("Core" is GHC's main intermediate language.)

- We give measurements of the effectiveness of `SpecConstr`, both for the full `nofib` suite, and for a few kernel array-fusion benchmarks (Section 5). The results are encouraging: the `nofib` suite shows a 10% improvement in run time, and the array-fusion benchmarks run twice as fast or (sometimes much) better.

Good compilers have a lot of bullets in their gun; each particular bullet may only be really effective on a few targets, but few programs evade all the bullets. `SpecConstr` is an excellent bullet. It is cheap to implement (less than 500 lines of code in a compiler of 100,000 lines), and the programs that it hits are well and truly knocked for six.

2. Motivation

We begin by giving some further examples that motivate the need for call-pattern specialisation of recursive functions.

2.1 Avoiding allocation

The `last` example shows that specialising a recursive function can avoid redundant *pattern matching*, but it can also avoid redundant *allocation*. Consider this standard function:

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

GHC translates Haskell into a small intermediate language called *Core*, which is what the optimiser works on. Here is the translation of drop into Core[1], after a bit of inlining:

```
drop = \n. \xs.
  case n of {
        I# un ->

  case un of {
      0 -> xs ;
      _ ->

  case xs of {
        []     -> [] ;
        (y:ys) -> drop (I# (un -# 1)) ys
  }}}
```

The first case takes apart n, which has type Int. In GHC the Int type is not primitive; it is declared like this:

```
data Int = I# Int#
```

This is an ordinary algebraic data type declaration, saying that Int has a single constructor I#, which has a single field of type Int# (Peyton Jones and Launchbury 1991). The "#" has a mnemonic significance only; the constructor I# is just an ordinary constructor with a funny-looking name. The type Int# is a truly primitive type, however, built into GHC; it is the type of 32-bit finite-precision integer values. So the second case expression does a perfectly ordinary pattern-match on n to bind un (short for "unboxed n"), of type Int# to the value of n. Haskell is a lazy language, so n may be a thunk in which case the case expression will evaluate it.

The second case expression tests whether or not un is zero, while the third scrutinises xs to see whether or not it is of form (y:ys); if so, there is a recursive call to drop, passing (n-1) as argument. This argument (which must have type Int) is built by decrementing un, using the built-in operator -# :: Int# -> Int# -> Int#, and constructing an Int value using the data constructor I#.

Notice that in every iteration except the last, *the newly-constructed* Int *is immediately de-constructed in the recursive call*. In short, there is a redundant heap allocation of the Int value in every iteration, leading to increased memory traffic and garbage-collector load. Especially for tight loops, eliminating allocation is highly desirable.

In the case of drop, we want to specialise the function for the case when the first argument has shape (I# <something>). The specialised function looks like this:

```
drop' :: Int# -> [a] -> [a]
drop' = \un.\xs. case un of {
                   0 -> xs ;
                   _ ->
                 case xs of {
                   []     -> [] ;
                   (y:ys) -> drop' (un -# 1) ys }}
```

Now there is no allocation in the loop — and removing allocation from the inner loop of a program can be a very big win indeed.

2.2 Stream fusion

These examples are suggestive, but our recent interest in SpecConstr was provoked by the work on stream fusion by Coutts et al. (2007a,b). Their goal is to eliminate intermediate data structures, such as lists or arrays. For example, consider the following function:

[1] NB: this display omits all type information, which Core includes.

```
sum_append :: [Int] -> [Int] -> Int
sum_append xs ys = sum (xs ++ ys)
```

We would like to compute the result without constructing the intermediate list xs++ys. The details of their work can be found elsewhere in this proceedings, but the key point is this: a fold operation (sum in this case) is performed by a *stream* fold, looking something like this:

```
data Stream a
  = forall s. Stream (s -> Step a s) s

data Step a s = Done | Yield a s

sumStream :: Stream Int -> Int
{-# INLINE sumStream #-}
sumStream (Stream next s)
  = go 0 s
  where
    go z s = case (next s) of
                Done       -> z
                Yield x s' -> go (z+x) s'
```

The intention is that sumStream will be inlined at its call sites, which will instantiate its body with a (perhaps rather complicated) function next and a (perhaps also complicated) initial state s, thereby producing a specialised, but still recursive, version of go.

In the case of sum_append, here it is the code that arises after inlining sum and (++), and simplifying a little (this example is taken from Coutts et al. (2007a)):

```
sum_append xs ys
 = go 0 (Left xs)
 where
    go z (Left xs)
       = case xs of
           []      -> go z (Right ys)
           x : xs' -> go (z+x) (Left xs')
    go z (Right ys)
       = case ys of
           []      -> z
           y : ys' -> go (z+y) (Right ys')
```

Notice the recursive calls to go with explicit Left and Right constructors, and the pattern matching on that same parameter. If we specialised go for these two cases we would get this:

```
sum_append xs ys
 = go_left 0 xs
 where
   go_left z xs  = case xs of
                     []      -> go_right z ys
                     x : xs' -> go_left (z+x) xs'
   go_right z ys = case ys of
                     []      -> z
                     y : ys' -> go_right (z+y) ys'
```

Now the program stands revealed: go_left adds the elements of xs into an accumulating parameter, and then switches to go_right, which does the same for ys. Stream fusion entwines these two loops together into one, driven by a state that distinguishes them. SpecConstr unravels the loop nest, improving performance by avoiding the allocation of Left and Right constructors. Indeed, without SpecConstr the performance is no better than the original list-ful program.

(The alert reader may notice that the good performance of sum_append relies on the strictness analyser spotting that go_left

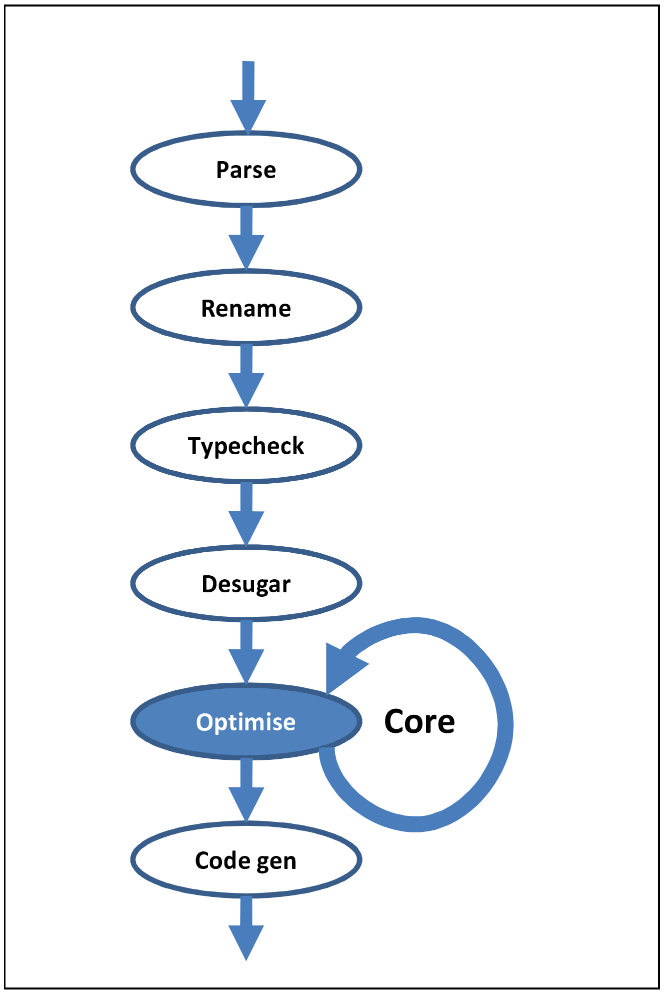

Figure 1: GHC compilation pipeline

and `go_right` are strict in `z`, else the accumulating parameter will build up a chain of thunks. That is indeed true, but it is *also* true of the original `sumStream` function. Furthermore, we run the strictness analyser *before* `SpecConstr`, so the latter transforms the program only after strictness analysis has already done its work. Hence, one does not need to worry that `SpecConstr` might obfuscate the program, thereby defeating strictness analysis.)

3. Implementing `SpecConstr`

It is easy enough to write down the desired result of the transformation, but we also need a general algorithm that implements it. In this case, we can leverage GHC's existing infrastructure to make the algorithm rather simple. Before discussing the `SpecConstr` implementation, we therefore digress briefly to describe this infrastructure.

GHC's compilation pipeline looks like Figure 1. The program is parsed, renamed, typechecked, and desugared into the Core language. Core is a small, explicitly-typed lambda-calculus language in the style of System F. The Core program is processed by a succession of Core-to-Core optimisations, one of which is `SpecConstr`, after which it is fed to the code generator.

A particularly important Core-to-Core pass is the *Simplifier*, which implements a large set of simple, local optimising transformations, such as constant folding, beta reduction, inlining, and so on (Peyton Jones and Santos 1998). A common pattern is that a sophisticated optimisation, such as strictness analysis or `SpecConstr`, does the heavy lifting, but produces a result program that is littered with local inefficiencies, of precisely the sort that the Simplifer can clean up. The assumption that the Simplifier will run later makes each optimisation much simpler to implement.

Returning now to `SpecConstr`, the implementation proceeds in three steps:

Step 1: *Identify the call patterns* for which we want to specialise the function.

Step 2: *Specialise*: create a specialised version of the function for each distinct call pattern.

Step 3: *Propagate:* replace calls to the original function with calls to the appropriate specialised version.

A *call pattern* for a particular function is a pair

$$\overline{v} \rhd \overline{p}$$

where $\overline{v}$ is a list of variables, which we call the *pattern variables*, and $\overline{p}$ is a list of argument expressions. In the case of `drop`, the recursive call `(drop (I# (un -# 1)) ys)` gives rise to the call pattern

$$[v, ys] \rhd [\texttt{I\#}\ v,\ ys]$$

A call pattern describes the argument templates for which we want to generate a specialised variant of the function. In the case of `drop`, the call pattern specifies a specialised variant for calls of the form

$$\texttt{drop (I\#}\ \langle\textit{something}\rangle\texttt{)}\ \langle\textit{something else}\rangle$$

The pattern variables v, ys stand for the "$\langle$*something*$\rangle$" holes in this template. The order of pattern variables $\overline{vs}$ in a call pattern is unimportant, and a call pattern is insensitive to consistent α-renaming of its pattern variables.

In the case of `drop`, the number of pattern variables happens to be the same as the number of arguments, but that is not in general the case. To illustrate, here are some further examples of call patterns:

$[x] \rhd [\texttt{True}, x]$	First argument is `True`, second is anything
$[x, xs] \rhd [(x\texttt{:}xs)]$	Sole argument is a cons `(:)`

We now give the details of Steps 1-3 identified above, in order of increasing difficulty, starting with Step 3.

3.1 Step 3: Propagation

The third step of the algorithm, propagation, replaces calls to the original function with calls to a specialised version of the function, whenever such a version has been created by the earlier steps. The propagation step is particularly easy to implement. The Simplifier already provides a general mechanism called *extensible rewrite rules*, that allows an optimisation pass (or indeed the programmer) to create a rewrite rule that is subsequently applied by the Simplifier as it traverses the program (Peyton Jones et al. 2001). Knowing that the Simplifier will run subsequently, all Step (3) need do is create one rewrite rule for each call pattern. For example, in the case of `drop`, from the call pattern $[v, ys] \rhd [\texttt{I\#}\ v,\ ys]$ Step (3) adds the rule:

```
{-# RULES
  "drop-spec" forall v xs.
                  drop (I# v) xs = drop' v xs
#-}
```

(More precisely, this is the concrete syntax that a programmer would use to express a rewrite rule, but Step (3) directly creates the rule in its internal form.) The effect of this rule is that whenever the Simplifier sees a call matching the left-hand side of the rule, it replaces the call with the right-hand side. The rule applies when compiling the module `Data.List` where `drop` is defined, but it *also* survives across separate compilation boundaries, so that any module that imports `Data.List` will also exploit the rule.

Given a function f, a call pattern $[v_1, \ldots, v_n] \triangleright [p_1, \ldots, p_m]$, and the corresponding specialised function f', the rule is trivial to generate:

$$\texttt{forall}\ v_1 \ldots v_n.\ f\ p_1 \ldots p_m = f'\ v_1 \ldots v_n$$

Incidentally, since Core is an explicitly-typed, polymorphic language, the pattern variables vs may, and often do, include type variables. For example, in its internal form the explicitly-typed rule for `drop` looks like this:

$$\begin{array}{l}\texttt{forall}\ (a{:}*)\ (v{:}\texttt{Int\#})\ (xs{:}\,\texttt{[}a\texttt{]}). \\ \quad \texttt{drop}\ a\ (\texttt{I\#}\ v)\ xs\ =\ \texttt{drop'}\ a\ v\ xs\end{array}$$

The existing rewrite-rule mechanism therefore completely takes care of propagation.

3.2 Step 2: Specialisation

The Specialise step looks rather harder, because specialisation can have a radical effect. Whole chunks of code can disappear. For example, there is one fewer `case` expression in `drop'` compared with `drop`, and the allocation has disappeared; and in the `last` example, the call to `error` does not appear in `last'` at all. Fortunately, the Simplifier makes it easy. Given a function definition $f = \lambda x_1 \ldots x_m.e$, and call pattern $vs \triangleright ps$ for f, we can construct the specialised version f' thus:

$$f' = \lambda v_1 \ldots \lambda v_n.\ e[p_1/x_1, \ldots, p_m/x_m]$$

where the notation $e[p/x]$ means the result of substituting p for x in e. In our now-familiar `drop` example, with call pattern $[v, ys] \triangleright$ $[\texttt{I\#}\ v,\ ys]$, we get

```
drop' = \v.\ys. <body>[I# v/n, ys/xs]
```

where `<body>` is the Core code given near the start of Section 2.1. Of course, this code is *bigger* than the original definition, since we are substituting terms p_i for variables v_i. But the whole point is that we do this precisely when (we believe that) the Simplifier will subsequently be able to simplify the substituted body.

In the case of `drop`, for example, `(case n of ...)` in the original `drop` becomes `(case (I# v) of ...)` in the substituted body, so the case expression can be eliminated, leaving

```
drop' :: Int# -> [a] -> [a]
drop' = \v.\xs. case v of {
                  0 -> xs ;
                  _ ->
                case xs of {
                  []     -> [] ;
                  (y:ys) -> drop (I# (v -# 1)) ys }}
```

At this stage `drop'` calls `drop`. However, the Simplifier can apply the rewrite rule `"drop-spec"` that we constructed in Step (3), and that "ties the knot" to give the self-recursive code for `drop'` given at the end of Section 2.1.

In short, Step (2) is extremely simple: just make a fresh copy of the right-hand side of the function, instantiated with the call patterns. The Simplifier will do the rest.

3.3 Step 1: Identifying call patterns

Now we turn to the first step, that of identifying the call patterns for which we want to specialise each function. Here is the set of heuristics that we used initially: we treat a call $(f\ e_1 \ldots e_n)$ as a specialisable call if all of the following conditions hold:

H1 The function f is bound by a definition of the form

$$f = \lambda x_1 \ldots x_a.e \qquad (a > 0)$$

That is, the lambdas are explicit, and the function has arity a.

H2 The right hand side e is "sufficiently small". In our implementation this size threshold is controlled by a flag.

H3 The function f is recursive, and the specialisable call appears in its right-hand side.

H4 All f's arguments are supplied in the call; that is $n \geq a$.

H5 At least one of the arguments e_i is a constructor application.

H6 That argument is `case`-analysed somewhere in the body of f.

We can only specialise functions whose definitions are statically visible (H1). For example, if f is lambda-bound, then even if we find a call in which f is applied to structured arguments, we cannot specialise f's definition. The further requirement that f's definition has explicit lambdas allows us to establish whether or not (H4-6) hold. For example, the definition

```
f = head ys
```

is not specialisable.

We specialise only *recursive* functions (H3), because they represent loops. A non-recursive function is often specialised by inlining. A large function will not be inlined, however, so it might be worth considering specialising non-recursive functions called from within loops; but we have not yet done so.

For such recursive functions, we specialise only calls found in the body of the function itself (H3 again). Calls from outside the function start the loop; calls in the body are part of the loop. Moreover, to keep things simple we require that the call is saturated; that is, if the function definition looks like $f = \lambda x_1 \ldots x_a.e$ then the call has at least a arguments (H4).

The essence of call-pattern specialisation is that one or more arguments of the call is a constructor application (H5). However, there is no point in specialising the function for a call pattern unless the specialised version can take advantage of the knowledge about the argument's shape, and that is what (H6) is about. Take a unary function $f = \lambda x.e$, for example. The body of the specialised function is $e[p/x]$, where e is the body of the original function, and p is a constructor application. This is only going to be an improvement if some, or preferably all, of the occurrences of x in e are the scrutinee of a `case` expression, because then the `case` expression can be eliminated, and the constructor application need never be constructed. For example consider:

```
f x y = (case x of { Just v -> v; Nothing -> 0 })
        : (f (Just y) (y+1))
```

For (H5) we note that the recursive call has the constructor application `(Just y)` as its argument; while for (H6) we note that `f` decomposes its first argument with a `case` expression. Notice, however, that the deconstruction of `x` need not lexically enclose the recursive call; indeed it need not be certain to be evaluated at all. Specialisation still eliminates the `case`, regardless. In our current example we get (after some simplification):

```
{-# RULE "fs1" forall y w. f (Just y) w = f’ y w #-}
f’ v y = v : f’ y (y+1)
```

The call-pattern-identification algorithm (Step 1) therefore works in two phases, as follows:

Step 1.1: It traverses the program, gathering two sorts of information:

(a) *Call instances*; that is, the function together with its actual arguments (i.e. not yet turned into call *patterns*).

(b) *Argument usage*; that is, information about which arguments are scrutinised by `case` expressions.

Step 1.2: At the definition site for a function f it combines (a) the call-instance information for f with (b) the argument-usage information that describes how f uses its arguments, to make the *call patterns* for which f will be specialised.

For example, consider a recursive function $f = \lambda x_1 x_2.e$. Suppose that in e we find a call $(f\ (\texttt{Just}\ v)\ (p,q))$, where both arguments are constructor applications. Suppose further that the argument-usage information from e indicates that only x_1 is scrutinised by a `case` expression. Then it would be fruitless to specialise on x_2, so we generate the call pattern $[v,w] \triangleright [\texttt{Just}\ v, w]$, where w is a fresh pattern variable.

Pattern variables are also used in call patterns in place of parts of the call that do not take the form of constructor applications. For example, consider the call $(f\ (\texttt{Just}\ x)\ (g\ x))$. It would be perfectly correct to specialise f for the call pattern

$$[g,x] \triangleright [\texttt{Just}\ x, g\ x]$$

But it would be foolish to do so, because the specialised version of f cannot usefully exploit the knowledge that its second argument is a function application. Instead, when turning a call instance into a call pattern, GHC abstracts each sub-expression of the call that is not a constructor application by a pattern variable. In our example, the derived call pattern would be $[x,w] \triangleright [\texttt{Just}\ x, w]$.

Lastly, in step 1.2, GHC eliminates duplicate call patterns, modulo α-conversion of course, since nothing is gained by making two identical specialisations of the same function.

3.4 Summary

This concludes the overview of the `SpecConstr` transformation. The transformation is implemented in GHC, which is itself written in Haskell. As a way to make the earlier discussion more concrete, here is the type signature of the main `SpecConstr` function in GHC’s implementation:

```
specExpr :: ScEnv -> CoreExpr
         -> (ScUsage, CoreExpr)

data ScEnv
   = SCE { sc_size      :: Int,
           sc_how_bound :: Map Id HowBound }

data HowBound = SpecFun | SpecArg

type ScUsage = (Calls, ArgUsage)

type Calls = Map Id [Call]
type Call  = [CoreExpr]

type ArgUsage = Set Id
```

The function `specExpr` takes a `CoreExpr` and an environment that gives information about the context of the expression. It returns a transformed expression, along with usage information (of type `ScUsage`) that describes how the expression uses its free variables.

The environment `ScEnv` has two fields:

- `sc_size`, the (fixed) size threshold for specialisation (H2).
- `sc_how_bound`, a finite mapping that identifies specialisable functions (`SpecFun`), and their arguments (`SpecArg`). This mapping is extended in the obvious way when the transformation moves inside the body of a specialisable function (H1). `Id` is GHC’s data type for identifiers.

The usage information, `ScUsage` has two components: call instances (`Calls`) and argument usage (`ArgUsage`). The former is simply a finite mapping from a specialisable function (a `SpecFun`) to a list of its calls, each represented by a list of arguments. The latter is a set of the identifiers (identified as `SpecArgs`) that are scrutinised by a `case` expression.

4. Refining the basic scheme

The alert reader will have noticed that *the particular choice of call patterns does not affect correctness*. We can specialise for too many call patterns (so that the specialisations go unused) or too few (so that worthwhile optimisation opportunities are missed), but in either case the program will still run correctly. Our goal is to select call patterns for which useful optimisation opportunities will arise.

So only Steps 2 and 3 affect correctness, and it is easy to prove that they are in fact correct. Step 2 simply adds a new function definition, which has no effect on the meaning of the program, so the only question is whether the rewrite rule created in Step 3 is correct. For a function $f = \lambda x_1 \ldots x_m.e$, and call pattern $\overline{v} \triangleright \overline{p}$, the rule looks like:

$$\texttt{forall}\ v_1 \ldots v_n.\ f\ p_1 \ldots p_m = f'\ v_1 \ldots v_n$$

Does the equality claimed by the rule hold? The left-hand side of the rule is, by β-reduction, equal to $e[\overline{p}/\overline{x}]$. The definition of f' is $f' = \lambda v_1 \ldots v_m.e[\overline{p}/\overline{x}]$, so the right-hand side of the rule is also equal to $e[\overline{p}/\overline{x}]$, and we are done.

Matters are much less cut-and-dried when it comes to identifying call patterns (Step 1). We have spent considerable time tuning the choice of call patterns in the light of experience, and these refinements are the subject of the rest of this section. Remember: they are all optional!

4.1 Variables that have known structure

Consider this function:

```
f1 n x = ...(case x of (p,q) -> f1 p y)...
       where
         y = (n,True)
```

Although the argument `(n,True)` does not appear *literally* in the call, it is obvious that we would like to record a call pattern $[p,n] \triangleright [p, (n, \texttt{True})]$, in which the second argument of `f` is a pair whose second component is `True`.

A very similar situation arises when the recursive call occurs in a branch of a `case` expression, thus:

```
f2 n x = case x of { (p,q) -> ...(f2 m x)... }
```

Again, although the second argument of the recursive call is not literally a constructor application, `x` is known to be a pair `(p,q)` at the moment of the call. So it is desirable to record the call pattern $[m,p,q] \triangleright [m, (p,q)]$.

A third situation is a call like this:

```
...f3 (let x = h y in (x,x))...
```

Again, `f3` is not literally applied to a pair, but despite the intervening `let` it is clear that `f3` could usefully be specialised for the call pattern $[p,q] \triangleright (p,q)$.

Incidentally, the reader might wonder why the Simplifier does not instead eliminate the `let` in the first place. It cannot substitute `(h y)` for `x`, because that would duplicate the call of `h`. Alternatively, it could float the `let` outwards, to give this:

```
...let x = h y in f3 (x,x)...
```

Indeed it will do so if `f3` is strict, but not otherwise, because if `f3` is lazy the transformed program risks allocates two objects (the thunk for `(h y)` and the pair `(x,x)`) instead of one (the thunk for `let x = h y in (x,x)`).

To summarise, here is the refinement:

R1: when collecting call patterns, `SpecConstr` should take account of

- Variables that are `let`-bound to a constructor application (example: `f1`).
- Variables that have been `case`-analysed by an enclosing `case` expression (example: `f2`).
- Arguments that are constructor applications disguised by enclosing `let`s (example: `f3`).

Exactly the same three refinements must also be made to the Simplifier's rule matcher. For example, the call pattern for `f2` will generate a rewrite rule looking like this:

```
{-# RULES
  "f2-spec" forall m p q. f2 m (p,q) = f2' m p q
#-}
```

The rule matcher embodied in the Simplifier must spot that the call in the right hand side of `f2` matches this rule. Similarly, the rule matcher should spot that the call

```
f2 m (let x = h y in (x,x))
```

is also an instance of rule `f2-spec`, and rewrite it to

```
let x = h y in f2' m x x
```

Notice that these refinements to the rule matcher are useful for *all* rules, not only for those generated by `SpecConstr`.

A reader who is familiar with Haskell may also notice that `f2` is strict in `x`, and so GHC's strictness analyser will make `f2` use call-by-value and, furthermore, will pass two components of the pair separately to the function, thereby achieving the same effect as call-pattern specialisation. But the `SpecConstr` transformation deals with two cases that leave the strictness analyser helpless. First, the function may not be strict:

```
f4 True  n x = n
f4 False n x = case x of
                 (p,q) -> ...(f4 c m x)...
```

Second, the argument may not be of a single-constructor type:

```
data Maybe a = Nothing | Just a

f5 :: Int -> Maybe Int -> Int
f5 n x = case x of
           Nothing -> n
           Just p  -> ...(f5 m x)...
```

Although `f5` is strict, GHC will still pass the argument `x` boxed. However, the `SpecConstr` transformation can spot that, at the recursive call, `x` is always of form `(Just p)`, and can make a specialised version of `f5` that passes `p` alone, eliminating the `case` expression altogether.

In concrete terms, the `SpecConstr` data structures sketched in Section 3.4 are modified as follows:

- The environment `ScEnv` is augmented with a field that describes the shape of any known variables:

  ```
  data ScEnv = SCE { ...;  sc_cons :: ShapeMap }

  type ShapeMap = Map Id (DataCon,[CoreExpr])
  ```

  That is, `ScEnv` is extended with new field `sc_cons` of type `ShapeMap`, which maps an identifier to its shape (if known). The type `DataCon` is GHC's data type representing a data constructor.

- The call-instance information must be augmented to capture the `ShapeMap` at the call site:

  ```
  data Call = Call ShapeMap [CoreExpr]
  ```

For example, in `f5` above, the `ShapeMap` would be augmented in the `Just` branch of the `case` with the mapping $[\mathtt{x} \mapsto \mathtt{Just\ p}]$. The `Call` record collected from the body of `f5`, will look like

$$\mathtt{Call}\ [\mathtt{x} \mapsto \mathtt{Just\ p}]\ [\mathtt{m}, \mathtt{x}]$$

4.2 Nested structure

Consider this function:

```
f x = ... (f (Just (x:xs))) ...
```

Here, the argument to `f` is a *nested* constructor application. It is obviously attractive to specialise for the nested call pattern $[x, xs] \triangleright [\mathtt{Just}\ (x{:}xs)]$. However, this apparently-simple refinement complicates (H6): we only want to specialise `f` for the nested call pattern if `f` not only `case`-analyses `x`, but *also* `case`-analyses the argument of the `Just`.

In general, then, we must record all the evaluation that `f` performs on its arguments, anywhere in its body. Here is a more complicated example:

```
data T = A (Either Bool Bool) | B (Int,Int)
data Either a b = Left a | Right b

g :: T -> Int
g (A (Left p))      = ...
g (A (Right True))  = ...
g (A (Right False)) = ...
g (B x)             = ...
```

Here are some call patterns for `g`, along with whether we would like to specialise `g` for that pattern:

$[p] \triangleright \mathtt{A}\ (\mathtt{Right}\ p)$	Yes
$[] \triangleright \mathtt{A}\ (\mathtt{Right\ True})$	Yes
$[] \triangleright \mathtt{A}\ (\mathtt{Left\ True})$	No: `g` does not decompose the `Bool`
$[x] \triangleright \mathtt{B}\ x$	Yes
$[x, y] \triangleright \mathtt{B}\ (x, y)$	No: `g` does not decompose the pair

So we must extract usage information from g's body that says:

- g's argument is case-analysed
- If it is of form A x, then x is case-analysed:
  - If x has form (Left p) then p is not analysed further.
  - If x has form (Right q) then q is case-analysed.
- If it is of form B x, then x is not further analysed.

In this example, all the pattern matching is done at the "top" of g, but that need not be the case; we have seen earlier examples in which the case expressions are nested inside the function, and/or passed as arguments to other function calls in the body.

R2: when collecting argument-usage information, SpecConstr should record *nested* pattern matches within the expression.

In concrete terms, rather than simply accumulating a *set* of the variables that are case-analysed, SpecConstr must accumulate, for each variable, evaluation information about each possible data constructor to which that variable might evaluate. This is a case where writing the code is easier than describing it informally:

```
type ArgUsage = Map Id ArgOcc

data ArgOcc = ArgOcc (Map DataCon [Maybe ArgOcc])
```

For each variable x we accumulate a finite map that gives information of x's occurrences. If x is in the domain of the ArgUsage map, then x is scrutinised; otherwise it is not. If it *is* in the ArgUsage map, its ArgOcc is a finite map that summarises, for each data constructor in x's type, how the pattern-bound arguments of that data constructor are used, using Nothing to indicate that the argument is not scrutinised. In our example g above, the ArgOcc for g's argument would look like this[2]:

$$\left[\begin{array}{l} \texttt{A} \mapsto \left[\begin{array}{ll} \texttt{Left} & \mapsto [\texttt{Nothing}] \\ \texttt{Right} & \mapsto \left[\begin{array}{l} \texttt{True} \mapsto [\,] \\ \texttt{False} \mapsto [\,] \end{array}\right] \end{array}\right] \\ \texttt{B} \mapsto [\texttt{Nothing}] \end{array}\right]$$

It is easy to define functions that take the union of two ArgOcc values, so that the ArgUsage from different sub-expressions can be merged as we move back up the tree.

4.3 Specialisation fixpoints

Consider this recursive function:

```
f :: Either Int Int -> (Int,Int) -> Int
f (Left n)  (p,q) = f (Right n) (q,p)
f (Right n) x     = if n==0 then fst x
                    else f (Left (n-1)) x
```

The function is somewhat contrived, in the interests of brevity, and we have again taken the liberty of using pattern-matching definitions instead of the case expressions that GHC really uses. From the right hand side of f we get two call patterns:

$$[n,p,q] \triangleright [\texttt{Right}\ n,\ (q,p)]$$
$$[m,x] \triangleright [\texttt{Left}\ m,\ x]$$

Specialising f for these patterns gives the functions:

```
-- Specialise f (Right n) (q,p)
f1 n q p = if n==0 then fst (q,p)
           else f (Left (n-1)) (q,p)

-- Specialise f (Left n) x
f2 n (p,q) = f (Right n) (q,p)
```

[2] We have left out a couple of Just constructors to reduce clutter

(We have dropped some dead code here, although in practice that would not be done until the simplifier runs in Step 3.) A cursory examination shows that the specialised function f1 has a *new* call pattern for f, namely $[m,p,q] \triangleright [\texttt{Left}\ m,\ (q,p)]$. On reflection, it is unsurprising that specialising a function may give rise to yet-more-specialised call patterns from its right-hand side. If we specialise f for this pattern too, we get:

```
-- Specialise f (Left m) (q,p)
f3 m q p = f (Right m) (p,q)
```

The call pattern from the right-hand side of this specialisation is $[m,q,p] \triangleright [\texttt{Right}\ m, (p,q)]$; and this one is the same as the call pattern for f1. So we have reached a fixed point.

After the simplifier runs, and propagates the specialised versions to their call sites, we get this resulting program:

```
f (Left n)  (p,q) = f1 n q p
f (Right n) x     = if n==0 then fst x
                    else f2 (n-1) x

f1 n q p = if n==0 then fst (q,p)
           else f3 (n-1) q p

f2 n (p,q) = f1 n q p

f3 m q p = f1 m p q
```

There is no construction and deconstruction of Left and Right inside the loop; and once the pair is evaluated for the first time it is never evaluated again.

The refinement we need is this:

R3: when specialising a function definition, collect new call patterns from its specialised right-hand side.

This refinement is easy to implement: we must simply call specExpr on each specialised copy of the function. In practice, instead of a two-pass algorithm — substitute and then specialise — GHC extends ScEnv with one more field, a substitution. Then specExpr substitutes and specialises at the same time, which turns out to be quite convenient.

Is there any guarantee that this iterative process will reach a fixed point? Yes, there is. Remember that we never specialise a function for call patterns that are "deeper" than the ArgUsage information, and this information is invariant across all the specialisations.

4.4 Mutual recursion and non-recursive functions

So far we have only considered *self*-recursive functions (H3). But mutual recursion is quite common, and gives rise to no real difficulty. Consider a recursive group

$$\texttt{rec}\ \{\ f_1{=}e_1;\ \ldots;\ldots f_n{=}e_n\ \}$$

We simply look for call instances for f_1, not only in e_1 but also in $e_2 \ldots e_n$. Then we derive call patterns from those calls, and specialise f_1 for those call patterns, just as before.

R4: when gathering call instances for a recursive function f, look in *all* definitions of the letrec in which f is defined.

This optimisation turned out to be important in practice, because GHC sometimes splits a self-recursive function into two mutually-recursive parts. Here is an example:

```
foo :: Maybe Int -> Int
foo Nothing  = 0
foo (Just 0) = foo Nothing
foo (Just n) = foo (Just (n-1))
```

This does not look mutually recursive, but GHC lifts the constant sub-expression (`foo Nothing`) out of the loop (Peyton Jones et al. 1996), to give this mutually-recursive pair:

```
lvl = foo Nothing
foo Nothing  = 0
foo (Just 0) = lvl
foo (Just n) = foo (Just (n-1))
```

In this case, nothing is gained by this transformation, since (`foo Nothing`) is evaluated only once in the loop, but GHC is not clever enough to spot this.

5. Results

We implemented `SpecConstr`, including all the refinements described above, in the Glasgow Haskell Compiler (as at March 2007). Our implementation comprises some 475 lines of Haskell. It is a straightforward Core-to-Core pass, which readily slots into GHC's compilation pipeline.

We measured its effectiveness in two ways. First, we ran the entire `nofib` benchmark suite with the `SpecConstr` transformation switched off, and then again with it switched on. All other optimisations were enabled (`-O2`). The results are shown in Figure 2. The minimum, maximum and geometric means are taken over all 91 programs in the suite, but the table only shows programs whose allocation changed by more than 2% or whose runtime changed by more than 5%. We place little credence in small changes in runtime, because they are hard to reproduce, whereas the allocation figures are repeatable. A runtime change of "-" therefore means that the runtime was too short to report a meaningful change. We compiled the entire set of Haskell libraries in the same way as the benchmark program itself (i.e. with or without running `SpecConstr`, respectively).

These figures show a consistent increase in binary size of a few percent, which is not surprising since we are duplicating code. Some programs, such as `queens` and `mandel2` show a dramatic decrease in allocation. Most others show much smaller changes (remember that many are suppressed altogether from the figure), but alas a handful show a noticeable increase. We investigated one of these, `fibheaps` in detail, and found that the increase was due to reboxing, which we discuss in Section 6.1. Nevertheless, run time almost invariably decreases, with a geometric mean of 10%.

The second way in which we evaluated `SpecConstr` was by applying it to some small array-fusion examples, taken from work in the Data Parallel Haskell project (Chakravarty et al. 2007). We took a set of five example pipelines of array operations that should fuse to make a single loop, and tried them with and without `SpecConstr`. The results are dramatic, and are shown in Figure 3. Performance is at least doubled, and in one case is multiplied by 10. Here is one of the pipelines:

```
pipe1 :: UArr Int -> UArr Int -> UArr Int
pipe1 xs ys = mapU (+1) (xs +:+ ys)
```

It could hardly by simpler: add the two arrays `xs` and `ys` element-wise, and then increment each element of the resulting array. For good performance it is essential that we eliminate the intermediate array. A `UArr Int` is an array of unboxed integers, so the fused loop will run along `xs` and `ys` adding corresponding elements, incrementing the result, and writing it into the result array. In

Program	Binary size % increase	Allocation % increase	Run time % increase
anna	+5.5%	+0.5%	-9.0%
ansi	+3.2%	-10.1%	-26.4%
atom	+3.1%	-0.1%	-12.8%
bernouilli	+3.3%	-2.0%	-15.1%
boyer	+3.2%	+0.0%	-16.4%
boyer2	+3.2%	+3.7%	-
calendar	+3.2%	+0.6%	-17.9%
cichelli	+3.3%	-2.2%	-2.6%
circsim	+3.0%	-0.7%	-5.1%
clausify	+4.7%	-6.8%	-5.5%
comp_lab_zift	+3.8%	+0.2%	-16.4%
compress	+3.3%	+0.0%	-19.0%
compress2	+8.7%	-2.2%	-13.7%
constraints	+3.1%	-0.6%	-5.8%
exp3_8	+3.1%	+0.0%	-18.6%
fft	+2.9%	-1.5%	-5.2%
fibheaps	+2.8%	+2.0%	-2.5%
fulsom	+2.8%	-0.3%	-10.8%
gamteb	+2.9%	-8.9%	-
gcd	+3.3%	-10.6%	-10.2%
gen_regexps	+3.2%	+0.0%	-6.4%
genfft	+3.0%	-0.5%	-17.0%
gg	+3.4%	+11.0%	-
ida	+3.5%	-0.5%	-27.9%
integer	+3.2%	-2.3%	-11.2%
knights	+3.2%	+0.0%	-8.5%
lcss	+3.3%	-0.0%	-9.2%
life	+3.2%	+0.0%	-14.3%
lift	+3.3%	+2.7%	0.00
listcompr	+3.0%	+0.1%	-20.4%
listcopy	+3.0%	+0.2%	-22.1%
mandel2	+3.5%	-67.3%	-
multiplier	+3.2%	+0.0%	-12.6%
parstof	+2.8%	-2.3%	0.01
pic	+3.0%	-5.6%	0.01
power	+3.6%	-4.6%	-25.0%
primes	+3.1%	+0.0%	-8.7%
primetest	+3.7%	-4.8%	-0.4%
puzzle	+3.2%	+0.0%	-11.2%
queens	+3.1%	-79.5%	-38.5%
rewrite	+3.4%	-0.0%	-6.1%
rsa	+3.8%	-8.5%	-
scs	+3.0%	-3.6%	-9.7%
simple	+2.9%	-0.0%	-10.5%
solid	+2.8%	+0.4%	-7.2%
sphere	+3.0%	-2.0%	-8.9%
symalg	+5.2%	-2.9%	0.02
transform	+3.8%	+0.3%	-13.0%
typecheck	+2.9%	+0.4%	-6.2%
wang	+3.0%	+0.3%	-20.3%
wheel-sieve2	+3.1%	+0.0%	-9.5%
...and another 40 programs...			
Min	+2.8%	-79.5%	-38.5%
Max	+8.7%	+11.0%	+3.7%
Geometric Mean	+3.4%	-3.7%	-10.5%

Figure 2: Effects of SpecConstr on nofib programs

Program	Runtime (ms) without	with	Runtime % increase
pipe1	506	205	-60%
pipe2	159	77	-51%
pipe3	284	114	-60%
pipe4	545	70	-87%
pipe5	6,761	720	-89%

Figure 3: Effects of SpecConstr on array-fusion pipelines

the fused loop, the increment is practically free, but there is a tremendous loss of performance if instead we allocate and fill an intermediate array. Array fusion is carried out using the same stream paradigm as that described in Section 2.2, but successful fusion is much more important.

(Indeed that is also the weakness of the fusion approach: failure to fuse can turn a good program into a bad one, yet that failure might be due to an obscure and hard-to-predict interaction of optimisation heuristics. It remains to be seen whether array fusion can be made reliable and robust enough that this problem does not show up in practice, but that is a challenge for another day.)

6. Further work

In Section 4 we described a series of refinements to the basic scheme. However we have also encountered shortcomings in our heuristics that are not so easy to fix, and we collect that experience here.

6.1 Reboxing

So far we have presented the advantages of specialisation. But here is a function for which matters are more ambiguous:

```
f :: (Int,Int) -> [Int]
f x = h x : case x of (p,q) -> f (p+1,q)
```

The recursive call to `f` is a constructor, and the argument `x` is indeed decomposed inside the function, so our heuristics say we should specialise it. Here is the specialised function:

```
f1 :: Int -> Int -> [Int]
f1 p q = h (p,q) : f1 (p+1) q
```

We see good news: `f1` does not allocate the pair `(p+1,q)` at the recursive call to `f`, as `f` did. But we see countervailing bad news: `f1` allocates the pair `(p,q)` at the call to `h`, which `f` does not. So, matters are no worse than before, and we have eliminated the `case`, but they are not as much better as we might have expected.

The problem is that the argument `x` is used in `f` *both* as a case scrutinee, *and* as a regular argument to an unrelated function `h`. Hence, if we pass the argument in its constituted parts, `p` and `q`, we may need to re-box them before passing it to `h`. We call this the “reboxing problem”. In the example above, allocation was no worse, and the code was shorter, so specialisation is probably still a good idea. Alas, there are other examples where allocation is *increased* by `SpecConstr` due to reboxing, and so specialisation would be a Bad Thing — and it is not easy to predict exactly when the problem will occur. One possibility is to modify (H6) thus:

H6’ Specialise on an argument x only if x is *only* scrutinised by a `case`, and is not passed to an ordinary function, or returned as part of the result.

Sadly, this is too conservative. Here is a slightly contrived example:

```
f :: Maybe Int -> Int -> Int
f x n = case x of
          Nothing -> 0
          Just m -> if n==0 then f x (n-1)
                    else m
```

Here `x` occurs passed as a regular argument — to `f` itself! So (H6’) would disable specialisation on the grounds that we do not want to risk reboxing `x`. But if we *do* specialise `f` for the first argument being `Just m`, the specialised function will not, in fact, rebox the argument. A simple-minded analysis risks throwing the baby out with the bathwater.

Our current implementation of `SpecConstr` simply ignores the reboxing problem. Close inspection reveals that reboxing is the reason that some programs allocate more heap with `SpecConstr` enabled, as we saw in Section 5. We need a more sophisticated analysis of argument usage, so that we can get (most of) the wins without risking losses from reboxing.

6.2 Function specialisation

Here is an example that arose in a real program.

```
the_fun :: [a] -> Step a [a]
the_fun x = ...non-recursive...

getA :: [[a]] -> [a]
getA [] = []
getA (x:xs) = getB the_fun x xs

getB :: ([a] -> Step a [a]) -> [a] -> [[a]] -> [a]
getB f x xs = case (f x) of
                Done       -> getA xs
                Yield y ys -> y : getB f ys xs
```

Here, `the_fun` is a non-recursive function, and only `getA` was called elsewhere in the program. It would obviously be profitable to specialise `getB` for the case when its first argument is `the_fun`, so that `the_fun` could be inlined in the specialised copy. This paper has focused on specialising functions for particular *constructor* arguments, but here we want instead to specialise on a particular *functional* argument. Doing so is of more than theoretical interest: the fragment above arose when compiling the expression

```
concatMap (map (+ 1)) xs
```

using the stream-fusion techniques of Coutts et al. (2007a).

Specialising for function arguments is more slippery than for constructor arguments. In the example above the argument was a simple variable but what if it was instead a lambda term? Should we generate a RULE like this?

$$f\ (\lambda x.\ldots)\ =\ e$$

The trouble is that lambda abstractions are much more fragile than constructor applications, in the sense that simple transformations may make two abstractions *look* different although they have the same value. So while we could generate such a rule, there is a good chance that it will never match anything!

An alternative approach might be to specialise only on function-valued *variables*, and perhaps also partial applications thereof. In the example above, that amounts to treating `the_fun` very similarly to a nullary constructor. We have not yet followed this idea through to its conclusion, but it looks promising.

6.3 Join points

Our heuristic (H3) also only considered *recursive* functions. However a recursive function may have a *non-recursive* function defined inside it. For example:

```
f x y = let g p = ...(f C p)...
        in case x of
            A p -> g True
            B q -> g False
            C   -> if y then ... else ...
```

Here, g is a local, non-recursive function defined inside the recursive function f. In fact, g is what GHC calls a "join point" because it is a single (albeit parameterised) piece of code that describes what to do in both the A and B branches of the case. Here we assume that g is large enough that GHC does not inline a copy at each call site.

Looking at f, the only call pattern for f has arguments $[\mathtt{C}, \mathtt{p}]$. However, if we specialised g, we would get call patterns for f with arguments $[\mathtt{C}, \mathtt{True}]$ and $[\mathtt{C}, \mathtt{False}]$, which is much better. There are two reasons that g is not specialised: first, it is non-recursive; and second, even if we were to specialise non-recursive functions, g does not scrutinise its argument p.

We do not yet have a good heuristic for specialising f. Although the situation may look contrived, it is not uncommon in practice. GHC generates many local join points as a result of the crucial case-swapping transformation, described in detail in Peyton Jones and Santos (1998).

7. Related work

Automatic function specialisation is a well-known idea. However, specialising recursive functions based on statically-known information is largely the domain of the partial evaluation community (Jones et al. 1993). Early partial evaluators would only specialise a function if one of its arguments was completely static (i.e. known to the compiler), but later work generalised this to *partially-static values* (Mogensen 1988). Even then, the "partially-static" nature of the structure usually appears to mean that part of the structure is completely known and part is unknown, whereas our focus is on structures whose *shape* is known. The partial-evaluation work that seems closest to ours is Bechet's Limix partial evaluator (Bechet 1994). In particular, he handles sum types as well as products, and he identifies the reboxing problem (Section 6.1). Indeed, he develops an analysis designed to identify functions where reboxing is not a problem.

Although there are some similarities, partial evaluation has quite a different flavour than the work described here. The SpecConstr transformation has modest goals and modest cost, whereas partial evaluation is a whole rich research area in its own right. Nevertheless, SpecConstr can certainly be regarded as a rather specialised partial evaluator.

Much more closely related is the work of Thiemann (1994) and its precursor (Thiemann 1993). Thiemann's goal is precisely the same as ours, although his context is that of a strict language. He presents a relatively sophisticated abstract interpretation to determine both argument usage and call patterns. Our work is less technically complex, but perhaps more practical; Thiemann's paper never led to a full-scale implementation.

8. Conclusion

The SpecConstr transformation is simple to describe, economical to implement, and devastatingly effective for certain programs. We are not ready to declare that it should be applied to every program, because it causes an increase in code size of 3-4%, even when it does not improve performance and, on occasion, can decrease performance. Nevertheless, an average runtime improvement of 10%, against the baseline of an already well-optimised compiler, is an excellent result. We have also identified avenues for further work that may improve it further.

Furthermore, and perhaps most important, we have advanced the enterprise of domain-specific optimisation technology. The idea is this: that ordinary programmers (i.e. not compiler writers) should be able to build libraries that, through the medium of programmer-specified rewrite rules, effectively extend the compiler with domain-specific knowledge (Peyton Jones et al. 2001). The streams library is an example of just such a library, and SpecConstr eliminates a road-block on its usefulness.

Acknowledgements

We owe a particular debt of thanks to Roman Leshchinskiy for supplying a long succession of small, well-characterised test cases for SpecConstr, and for his array-fusion benchmarks. My thanks also goes to Brandon Moore, Peter Thiemann, and the ICFP reviewers for their helpful feedback.

References

Denis Bechet. Limix: a partial evaluator for partially static structures. Technical report, INRIA, 1994.

Manuel Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Data Parallel Haskell: a status report. In *ACM Sigplan Workshop on Declarative Aspects of Multicore Programming*, Nice, January 2007.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, Freiburg, Germany, October 2007a. ACM.

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In *Practical Aspects of Declarative Languages (PADL'07)*, pages 50–64. Springer-Verlag, January 2007b.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

Torben Mogensen. Partially static structures in a self-applicable partial evaluator. In *Partial Evaluation and Mixed Computation*, 1988.

Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *2001 Haskell Workshop*. ACM SIGPLAN, September 2001.

SL Peyton Jones and J Launchbury. Unboxed values as first class citizens. In RJM Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston, 1991. Springer.

SL Peyton Jones and A Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

SL Peyton Jones, WD Partain, and A Santos. Let-floating: moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. ACM Press, Philadelphia, May 1996.

Peter Thiemann. Higher-order redundancy elimination. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pages 73–83, Orlando, Florida, June 1994. ACM.

Peter Thiemann. Avoiding repeated tests in pattern matching. In Gilberto Filé, editor, *3rd International Workshop on Static Analysis*, number 724 in Lecture Notes in Computer Science, pages 141–152, Padova, Italia, September 1993. Springer Verlag. ISBN 3-540-57264-3.