Playing by the Rules: Rewriting as a practical optimisation technique in GHC

Simon Peyton Jones  
Microsoft Research Ltd  
simonpj@microsoft.com  

Andrew Tolmach  
Portland State University  
apt@cs.pdx.edu  

Tony Hoare  
Microsoft Research Ltd  
thoare@microsoft.com  

March 19, 2001

Abstract

We describe a facility for improving optimization of Haskell programs using rewrite rules. Library authors can use rules to express domain-specific optimizations that the compiler cannot discover for itself. The compiler can also generate rules internally to propagate information obtained from automated analyses. The rewrite mechanism is fully implemented in the released Glasgow Haskell Compiler.

Our system is very simple, but can be effective in optimizing real programs. We describe two practical applications involving short-cut deforestation, for lists and for tree trees, and document substantial performance improvements on a range of programs.

This paper has been submitted to ICFP 2001.

1 Introduction

Optimising compilers perform program transformations that improve the efficiency of the program. However, a compiler can only use relatively shallow reasoning to guarantee the correctness of its optimisations. In contrast, the programmer has much deeper information about the program and its intended behaviour. For example a programmer may know that \texttt{integerToInt} (\texttt{intToInteger} \(x\) = \(x\)) where \texttt{Integer} is the type of infinite-precision integers and \texttt{Int} is 32-bit integers, but the compiler has little chance of working this out for itself. While programmers are unlikely to write such expressions themselves, they can easily appear when aggressive inlining brings together code that was written separately.

In this paper we explore a very simple idea: 
encourage the programmer to specify properties of the program, and allow the compiler to use these properties to improve performance, by treating each property as a rewrite rule. In effect, we give the programmer the ability to extend the compiler with domain-specific optimisations, giving it specialised knowledge about the particular vocabulary of functions that are used heavily in a particular program. Our setting is that of the purely functional language Haskell, because the lack of side effects makes it possible to state many properties simply, without complex side conditions, and to exploit them using only local information.

We make the following contributions:

- We describe a concrete design, which is fully implemented in the released Glasgow Haskell Compiler, an optimising compiler for Haskell (Section 2; Section 6).
- We describe two practical applications of the technique, one to perform list fusion in the Haskell standard Prelude (Section 3) and other to perform tree fusion in an application-specific library (Section 7).
- We show that rewrite rules can also be generated automatically as a result of compiler analyses, and then constitute a useful way to exploit specialised versions of functions (Section 5).

The idea of allowing the programmer to specify domain-specific compiler extensions is not new (Section 8), but it has not yet been widely successful. Our principal selling point is simplicity. Rewrite rules are expressed declaratively using the syntax of Haskell itself, and not in a separate metalanguage. They use very simple pattern matching, have no side conditions, and are applied using a trivial strategy. Yet they are effective in real programs, assuming some cooperation from library writers.

Traditionally, programs convey the minimum information about algorithms and data representations that is required to compile and execute the program. But programmers have always been encouraged (often ineffectively) to annotate their programs with additional documentation specifying the intended purpose and properties of the program, independently of the implementation.

Such program properties, expressed as equations, have been used to explore efficient algorithms and as a design methodology that reduces the incidence of programming error (Bird and Moor, 1996). Another advantage may be reaped in testing and debugging of programs, where they can play the role of a test oracle (Claessen and Hughes, 2000). Perhaps the additional incentive of efficiency gains in compilation will really persuade programmers at last to write more informative and more accurate documentation?

2 The basic idea

Consider the familiar \texttt{map} function, that applies a function to each element of a list. Written in Haskell, \texttt{map} looks like this:

\[
\text{map } f \, [] = [] \\
\text{map } f \, (x:xs) = f \, x : \text{map } f \, xs
\]
Now suppose that the compiler encounters the following call of map:

\[ \text{map } f (\text{map } g \, \text{xs}) \]

We know that this expression is equivalent to

\[ \text{map } (f \circ g) \, \text{xs} \]

(where \( \circ \) is function composition), and we know that the latter expression is more efficient than the former because there is no intermediate list. But the compiler has no such knowledge.

One possible rejoinder is that the compiler should be smarter — but the programmer will always know things that the compiler cannot figure out. Another suggestion is this: allow the programmer to communicate such knowledge directly to the compiler. That is the direction we explore here.

The Glasgow Haskell Compiler (GHC) allows the programmer to add a rule to the program thus:

```
{-# RULES
"map/map" forall f g xs.
map f (map g xs) = map (f . g) xs #-
```

The “{-# ... #-}” brackets enclose a pragma, which is ignored by a non-optimising compiler. The RULES keyword identifies the pragma as defining a rewrite rule. The "map/map" part is an arbitrary string that names the rule; this name is used when reporting which rules fired during a compilation run in diagnostic mode. The body of the rule expresses the identity that

\[ \text{map } f (\text{map } g \, \text{xs}) = \text{map } (f \circ g) \, \text{xs} \]

while the forall part identifies which of the variables in the rule body are universally quantified (i.e., in this case), and which are constants bound elsewhere (map in this case).

One can regard the rules for a function as extra (redundant) equations defining the function, thus:

```
map f [] = []
map f (x:xs) = f x : map f xs
map f (map g xs) = map (f . g) xs
```

Unlike ordinary defining equations, of course, rules are not restricted to having constructors in the patterns on the left hand side.

Rewrite rules express identities that the programmer knows to be true, but GHC also assumes that they are oriented, so that the right hand side is preferable to the left. Throughout compilation, GHC tries to spot instances of the left hand side of a rule, and rewrite that call to the right hand side.

A RULES pragma can occur only at the top level of the program, and all the free variables of the rule, on both sides of the equation, must be in scope. However, a rule is not required to be in the same module as the function whose definition it extends. For example the "map/map" rule does not have to be given in the module that defined map. So rules can incrementally extend a function's definition. This is important, because a rule may describe the interaction of an imported function with one defined locally. Rules can also be given for class member functions, in which case they work on the corresponding function in each class instance.

### 2.1 Assumptions

The ability to add rewrite rules to a program is a pretty powerful weapon, and raises a host of issues. In particular:

- GHC makes no attempt to verify that the rule is indeed an identity, apart from ensuring that the left and right hand sides of the rule have the same type. The whole point is that the rule asserts something that GHC is not smart enough to work out for itself!
  
  Indeed, the rule might not even be "true" in a concrete sense! For example, consider an abstract data type for sets. It is sound to give a rule expressing the fact that \text{union} over sets is commutative. But suppose our implementation represents a set by an unordered list. Then the concrete representation of a "union" \( b \) may differ from \( b \text{ 'union' } a \), even though they represent the same sets.

  Having the rules explicitly codified does, however, raise the possibility of feeding the same program into a theorem prover, and having it prove the veracity of the rules, perhaps with some human assistance. We have not explored this avenue so far.

- GHC makes no attempt to ensure that the right hand side is more "efficient" than the left hand side. Again, this is a hard problem in general and, what is worse, it is one that is to some extent compiler-dependent. For the present, we rely on the (fallible) programmer.

- GHC makes no attempt to ensure that the set of rules is confluent, or even terminating. For example, the following rule will send GHC into an infinite loop if it encounters a call to \text{foo}.

```
{-# RULES
"commute" forall x y. foo x y = foo y x #-
```

There is a considerable literature on proving the confluence or termination of sets of rewrite rules; in particular, commutativity and associativity have received special study (Baader and Nipkow, 1999). However, for us matters are seriously complicated by the other automatic rewrites that the compiler performs (beta-reduction, inlining, case switching, let-floating, etc. (Peyton Jones and Santos, 1998)), so we are not able to take direct advantage of this work.

For an optimising compiler, confluence seems too strong, since that would implausibly suggest a canonical optimised form for a program. Termination is certainly important, but has not proved to be a problem in practice.

### 2.2 Restrictions

GHC also places a restriction on the form of a rule. The left hand side of a rule must take the form of a function application, thus:

\[ f \, e_1 \ldots e_n \]

where \( f \) is not quantified in the rule (i.e., \( f \) is not one of the forall'd variables), and the \( e_i \) are arbitrary expressions.

Here, for example, is a plausible rule that we cannot write:
The rule is illegal because the left hand side is not a function application. This restriction has two advantages. First, it underpins the idea introduced above, that a rewrite rule is simply an extra (redundant) equation defining a function. Second, it makes rules matching much more efficient, because the rules can be indexed by the function on the left hand side. At each call of \( f \), GHC need only check matches for rules for \( f \). If the left hand side of a rule could instead be an arbitrary expression, matching is likely to be much less efficient.

The function-application restriction does mean that rules cannot be used to replace many of GHC's built-in transformations. Inlining, let-floating, beta reduction, case swapping, case elimination, and so on are all too complex to explain using our restricted language of rules. There are, however, some compiler transformations — such as specialisation — for which rules do prove directly useful, as we discuss in Section 5.

### 2.3 Library writers and library clients

Reading these assumptions and restrictions, one might reasonably ask: are rewrite rules going to be of practical use? It is certainly easy to shoot oneself in the foot.

For this reason, we regard a set of rewrite rules as something much more like a domain-specific compiler extension than a general programming paradigm. We expect rewrite rules to be written mainly by the author of a library. Such authors often go to great lengths to craft efficient data structures and algorithms. Rewrite rules give them the ability to explain deep truths about their code to the compiler, and thereby extend its ability to optimise client programs. We assume also a willingness to cooperate in the optimisation, to the extent of adapting library code to take advantage of the optimisation rules as well as the other way round. In return, we hope to preserve a level of simplicity, in which the correctness of the optimisation rules (but not their effectiveness, unfortunately) is as easy to establish as that of all the other clauses in a declarative program.

In GHC the rewrite rules defined in a module are embedded in the compiler-readable meta-data (its "\( .hi \) file") that accompanies the module's object code. The client of the library never sees the rules, but GHC can nevertheless use them to optimise compositions of calls to functions supplied by the library. Rules are not explicitly exported or imported. Instead, when compiling module \( k \), GHC can "see" all the rules given in any module imported by \( k \), or in any module imported by these imports, and so on transitively. (Haskell’s instance declarations have exactly the same property.)

Rewrite rules make perfect sense even if the library is written in another language, in which case the rules express facts about the foreign library. For example, in Reid’s graphics library for Haskell he provides a whole section of the user manual devoted to algebraic optimisation laws that are satisfied by the library interface (Reid, 2000).

### 3 Rules in practice

In the rest of the paper we report on our experience of applying rewrite rules in practice. We have found two main classes of applications:

- **Programmer-written rules in library code.** This was our initial motivation, and we have used it to achieve list fusion (this section) and more ambitious tree fusion (Section 7).
- **Automatically-generated rules, derived from some kind of program analysis, invisibly to the programmer (Section 5).** This was an unexpected, but very persuasive, practical benefit of implementing the rewrite-rule technology.

#### 3.1 Short-cut Deforestation

Our initial motivating example for adding rewrite rules was the case of list fusion. In earlier work we described so-called short-cut deforestation, a technique for eliminating intermediate lists from programs (Gill et al., 1993). At the centre of the method is the single rewrite rule "foldr/build":

```haskell
foldr :: (a->b->b) -> b -> [a] -> b
foldr k z [] = z
foldr k z (x:xs) = k x (foldr k z xs)
build :: (forall b. (a->b->b) -> b -> b) -> [a]
build g = g ()
```

```haskell
{-# RULES
"foldr/build"
forall k z (g::forall b. (a->b->b) -> b -> b) -> foldr k z (build g) = g k z
#-}
```

The definition of foldr is conventional. The function build takes a "list" \( g \), functionally abstracted over its cons and nil constructors, and applies \( g \) to the ordinary list constructors (\( \cdot \) and \( \square \) to return an ordinary list. (\( g \)'s type is a rank-2 polymorphic type, as discussed in (Gill et al., 1993).) The rule states that when foldr consumes the result of a call to build, one can eliminate the intermediate list by applying \( g \) directly to \( k \) and \( z \).

To give an example of applying this rule we must write list-consuming and producing functions using foldr and build respectively. For example:

```haskell
-- (sum [5,4,3,2,1]) = 15
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

-- (down 5) = [5,4,3,2,1]
down :: Int -> [Int]
down v = build (\n -> down' v c n)

(down' 0 cons nil = nil
down' v cons nil = cons v (down' (v-1) cons nil)

Again, the definition of sum in terms of foldr is conventional. The function down returns a list of integers. from its argument down to 1. We express it as a call to build, using an auxiliary function down' which is abstracted over
```
the functions it uses to construct its result. (We have called these functions \texttt{cons} and \texttt{nil} for old times’ sake, but they are simply the formal parameters to \texttt{down’} and their names are insignificant.) It is somewhat inconvenient to write \texttt{sum} and \texttt{down} in this way, but that is the task of the author of the List library.

Now we can try fusion on the call (\texttt{sum (down 5)}):

\[
\begin{align*}
\text{sum (down 5)} &= \{\text{inline sum and down}\} \\
\text{foldr (+) 0 (build (down’ 5))} &= \{\text{apply the foldr/build rule}\} \\
\text{down’ 5 (+) 0}
\end{align*}
\]

The intermediate list has been eliminated; instead \texttt{down’} does the arithmetic directly.

3.2 A real (albeit small) example

List fusion works well when the programmer does “build” operations over lists, and then it can be stunningly effective. Here is an example taken verbatim from the paraffins code (Parlait, 1992), a small program that computes a list of all the hydrocarbon paraffins of a given size:

\[
\begin{align*}
\text{three_partitions} :: \text{Int} \to [(\text{Int}, \text{Int}, \text{Int})] \\
\text{three_partitions} \text{ m} &= \{ \text{i} < [0..(\text{m ‘div’ 3})], \\
& \quad \text{j} < [1..(\text{m-i ‘div’ 2})], \\
& \quad \text{k} = \text{m} - (\text{i} + \text{j})\} \\
& \}
\end{align*}
\]

\[-- \text{A test harness}\]
\[\text{main} = \text{print} \ (\text{length} \ (\text{three_partitions} 4000))\]

The form [0..n] is Haskell’s notation for the list of integers between 0 and n. The list comprehension builds the list of all triples (i,j,k) where \text{i} is drawn from the list [0..(\text{m ‘div’ 3})], and \text{j} is drawn from a similar list, and \text{k} is computed directly from \text{i} and \text{j}. Finally, the test harness prints the length applying \text{three_partitions} to 4000.

\text{GHC} translates range notation, [0..n], into an application of \text{build}, much as we did for \text{down} above. It translates a list comprehension into a \text{build}, using \text{foldr} to consume the sub-lists. Finally, the Prelude library function \text{length} is implemented using a \text{foldr}.

So in this program, all the intermediate lists are removed, leading to a dramatic drop in allocation. When fusion is enabled, this program allocates 16 Mbyte; when fusion is switched off it allocates 188 Mbytes. (Most of the allocation for the fused version is used for the stack, because the length computation is not properly tail-recursive, so the stack grows 1.3M activation records.)

3.3 Benchmark Results

Over a broader range of programs from the nofib benchmark set (Parlait, 1992) the effect of enabling list fusion is very patchy, as Figure 1 shows. Fusion has no measurable effect on most programs but it gives a useful 5-25% reduction in allocation for a few. Only a very few programs are made worse, and the worst of these by less than 4%. One program, a parser called parsetof, shows a 96% improvement; this turns to be because fusion transforms the (artificial) outer loop of the benchmark, causing the sample test input to be parsed once instead of 40 times!

The geometric mean improvement, about 5% if we omit parsetof, seems disappointingly low, but we are undismayed. Compiler optimisations are like therapeutic drugs. Some, like antibodies, are effective on many programs; such optimisations tend to be built into a compiler. Others are are targeted at particular “diseases”, on which they are devastatingly effective, but have no effect at all on most other programs. The rules mechanism allows library authors to add targeted, domain-specific optimizations without modifying the internals of the compiler.

We also hope that programmers may adopt a more modular programming style if they expect fusion to take place. For example, it is clearer to write

\[
\text{concat (map f xs)}
\]

than it is to write

\[
\text{foldr ((++) . f) [] xs}
\]

Yet programmers will sometimes write the latter form because it does not build an intermediate list. Section 7 gives an extended example of the way in which fusion can make modular programming practically efficient.

Finally, note that our measurements relate to un-modified benchmark programs. None of the functions in these programs use \text{build}, so fusion only occurs for compositions of functions from the Standard Prelude, whose functions we re-implemented using \text{foldr} and \text{build}. If the compiler were to transform user-written functions to use \text{foldr} and \text{build} we might see greater benefits — but that is beyond the scope of this paper, and in any case certainly would require compiler modification (Lauchlurby and Sheard, 1995).

4 The sticky details

So far we have implied that one simply needs to add one rewrite rule, and re-implement some key functions using
foldr and build. In practice, though, we encountered a number of obstacles that we discuss in this section.

4.1 Phases

First, there is a subtle interaction between function inlining — a transformation that GHC does aggressively (Peyton Jones and Marlow, 1999) — and rule application. Returning to our sum/down example, we can see:

- **sum** and **down** must both be inlined before the rule can fire.
- On the other hand **foldr** and **build** must *not* be inlined. For example, inlining **build** before firing the rule would give

  \[
  \text{foldr} (+) 0 \ (\text{down'} \ 5 \ (\_ : []))
  \]

  and we have lost the fusion opportunity.

On the other hand, once we have run out of opportunities to use the foldr/build rule, there is no further point in not inlining build. Indeed, recall that its definition is both small and higher-order:

\[
\text{build} \ g = g \ (\_ : [])
\]

Inlining a function like this is very beneficial. So we are led inevitably to a phase ordering: first apply rules and then inline build.

Alas, two phases are not necessarily enough. In general, a program uses many layers of abstract data types, each implemented using the layer below. First we want to apply rewrite rules for the top-level ADT; then we want to expose its implementation (only to the compiler, of course) by inlining, and apply rewrite rules for the next layer; then we want to inline that layer and apply rewrite rules for the layer below; and so on.

Organising rules into phases is a form of rewriting strategy, a subject that has received considerable attention (Visser, 1999; Clavel et al., 1996; Visser et al., 1998) However, one of the merits of rewrite rules is their simple, declarative nature: “here is a true fact: please use it whenever possible”. We resist polluting this story with elaborate rewrite strategies. Nevertheless, it seems that some very simple strategy, such as a phase organisation is necessary. To gain experience, we have implemented a simple scheme, whereby the programmer can specify in which phase a function should be inlined. Thus we might say:

```haskell
{-# INLINE 2 build #-}
build g = g (:) []
```

to mean “inline build in phase 2”. Of course, this means the programmer must know something about GHC’s phases, which is undesirable. Though various more elaborate schemes have occurred to us — using the module hierarchy, for example — we have not yet found one we regard as satisfactory.

4.2 Backing out

Suppose fusion does not take place. That is, suppose we have an isolated call (down 34). It would be bad to actually implement down using build and down’, because doing so involves much more run-time function-passing than a straightforward implementation of down. It is unacceptable for programs to run slower in the (common) places when fusion fails than using the original library.

One solution is to rewrite down’ to be non-recursive, and inline vigorously:

\[
\text{down} :: \text{Int} \to \text{[Int]}
\]

\[
\text{down} \ v = \text{build} \ ((\_ \ n \to \text{down'} \ v \ c \ n) \ n)
\]

\[
\text{down'} \ v \ \text{cons} \ \text{nil} = \text{go} \ v
\]

\[
\text{where}
\]

\[
\text{go} \ 0 = \text{nil}
\]

\[
\text{go} \ v = \text{cons} \ v \ (\text{go} \ (v-1))
\]

Now suppose we have inlined down at a call (down 34), but alas it has not fused with a foldr. We can now inline as follows:

\[
\text{build} \ ((\_ \ n \to \text{down'} \ 34 \ c \ n) \ n)
\]

\[
\text{Did not fuse}
\]

\[
\text{--}
\]

\[
\text{Inline build}
\]

\[
\text{--}
\]

\[
\text{Inline down'}
\]

\[
\text{--}
\]

\[
\text{go} \ 34 \ \text{where}
\]

\[
\text{go} \ 0 = []
\]

\[
\text{go} \ v = v : \text{go} \ (v-1)
\]

This code is as good as the original, straightforward implementation of down — because is is the original, straightforward implementation of down! The trouble is that we have effectively made a complete copy of the straightforward code at every call site. While this is acceptable for a function as small as down, it would be quite undesirable for larger functions.

An alternative solution, and the one we generally adopt, is to add a new definition and rewrite rule:

```
{-# RULES "downList" -
forall v. down' v (:) [] = downList v #-
```  

An isolated call to (down 34) would now transform as follows:

```
down 34
```

```
= (Inline down)
```

```
build (down' 34)
```

```
= (Inline build)
```

```
down' 34 (;) []
```

```
= (Apply "downList" rule)
```

```
downList 34
```

The "downList" rule solves the special case in which down’ is applied the standard list constructors, and transforms the call to use the directly-code downList function.

4.3 One-shot lambdas

Here is the definition of map in terms of foldr and build:

```haskell
map g = foldr g []
```
map f xs = build (\c n -> foldr (c . f) n xs)
Now, suppose we find an application (map f (build g)).
We want to transform the call like this:

map f (build g)
= (Inline map) DANGER!
   build (\c n -> foldr (c . f) n (build g))
= (Apply foldr/build rule)
   build (\c n -> g (c . f) n)
The difficulty is in the step marked DANGER!. Here we substi-
tute (build g) for xs in the body of map, but this occurrence
of xs is under a lambda abstraction. In general, one
can make a program run arbitrarily more slowly by substitut-
ing a redefine inside a lambda abstraction, so GHC usually
does something more conservative:

map f (build g)
= (Inline map) SAFE!
   let xs = build g
   in build (\c n -> foldr (c . f) n xs)
Alas now the foldr/build rule cannot fire!
The solution is to observe that the abstraction
(\c n -> ...) is a one-shot lambda; that is, it is a
function that is only called once. Why? Because it is the
argument to build, and build simply calls its argument,
passing (t) and [ ]. Substituting inside one-shot lambdas
is perfectly safe.
The Right Thing To Do is to analyse the program for
one-shot lambdas and act accordingly. A type-based analy-
sis that achieves this (among other things) is described
by Wansbrough (Wansbrough and Peyton Jones, 1990), but it
is not yet fully implemented in GHC. Instead we have a tem-
porary hack that spots the special case of an application of
build

4.4 Sharing
Consider this function

\( f \ x = \text{sum} (\text{filter} (\geq \ x) [1..10]) \)
One might expect all intermediate lists to be eliminated from
this function, but GHC correctly spots that the expression
[1..10] can be floated out:

one_to_ten = [1..10]
f \ x = \text{sum} (\text{filter} (\geq \ x) \text{one_to_ten})
Alas, now the filter consumer cannot fuse with the [1..10]
producer. Floating out one_to_ten would be a good transfor-
mation if the producer — in this case [1..10] — were
more expensive. It would be worth losing the fusion, in or-
der to share the computation of one_to_ten among all calls
to \( f \). But in the case of [1..10], it would be better to lose
sharing to gain fusion.
This problem turned out to be central when Elliott et al.
tried to use rewrite rules to optimise Pan programs (Elliott
et al., 2000). In Pan, it is crucial to inline absolutely every-
thing, caring nothing for sharing, apply rewrite rules, and
then do aggressive common sub-expression and code-motion
transformations to make up for the loss.
This is a problem that is unlikely to have a cut-and-dried
solution, but we are exploring the idea of using virtual data
types. The programmer declares some data types as virtual,
5.2 Evaluated arguments

In array-intensive code, one often encounters a loop like this:

```haskell
f :: Int -> Int -> Int
f x y = case x of { 0y -> f y x }

fv :: Int -> Int -> Int
fv x y = if (x > 0) then 0
else case y of { 0y -> fv y x }
```

GHC represents values of type `Int` using the following data type:

```haskell
data Int = # Int#
```

where `Int#` is the type of unboxed, 32-bit integers. GHC will compile `f` thus:

```haskell
f :: Int -> Int -> Int
f x y = case x of { 0y -> f y x }
```

`fv` has turned into a mere “wrapper” that evaluates `x` before calling the “worker”, `fw` (Peyton-Jones and Launchbury, 1991). It can do this because `f` is sure to evaluate `x`. However, `f` is not certain to evaluate `y`, so the evaluation of `y` must be in the `else` branch of the conditional in the worker, `fw`. That means that the worker must box `y` before calling itself (“# (y + # 1)”), and in the common case, `y` will immediately be un-boxed again. This is bad.

What can be done? Again, it is a matter of specialisation. Recognising that there is a recursive call to `fw` in which the second argument is a constructor application, GHC can make a specialised version of `fw`, and generate an appropriate rule, thus:

```haskell
fw1 :: Int# -> Int# -> Int
fw1 x yv = let
  y = # yv
  in ...original RHS of fw....
```

After simplifying the right hand side of `fw1`. using the rule, we get just what we want:

```haskell
fw1 :: Int# -> Int# -> Int
fw1 x yv = if (x > 0) then # yv
else case yv of { 0y -> # (y + # 1) }
```

`fw` remains as an “impedance matcher” embodying the first iteration of the loop, before calling `fw1`. However the rule remains to transform any call of `f` with an already-evaluated second argument into a call to `fw1`.

All of this is done invisibly by the compiler — the programmer is not involved at all. The transformation is fully implemented in GHC, enabled by “-02”. The analysis, generation of specialised code, and generation of the rewrite rule, takes only 225 lines of Haskell. The rewrite-rule infrastructure automatically takes care of applying the rule when it is relevant, and propagating the rule across separate compilation boundaries.

5.3 Usage types

We are exploring another example of the same pattern. Wansbrough’s work on usage types suggests that considerable efficiency gains can be made by specialising functions based on their usage patterns. For example, consider this map again:

```haskell
map f (xs) = map f xs
```

If `map` is called in a context in which the result list is consumed at most once, then the thunks for its arguments can be made by using `map` to map, both at its definition and at its call sites. Once this is done, a specialised version of `map` can be compiled for the case when the usage-type argument is “once”, and a rule generated to match such calls, in exactly the same way as for specialising overloading.

5.4 Summary

In each example, we can discern the same pattern:

- Based on pragmas or program analysis, perform a local transformation (e.g., generating the specialised version of `invert`).
- Generate a rule that explains how that transformation can be useful to the rest of the program. In some cases the rule looks at the type arguments, in others at value arguments.
- Apply the rule throughout the rest of the program.

This may not sound like much, but it is extremely helpful to have a single, consistent way to propagate the benefits of a transformation to the rest of the program. For example, it is not enough for the specialiser to generate specialised versions of a function and find all appropriate call sites for the specialised function. There may not be any calls to `invert` at type `Int` when the specialiser runs. Such calls may only show up after some inlines have exposed them. Or they may be in other modules altogether, so the rule must be propagated across module boundaries (which is relatively easily done).

Programmer-defined RULES pragmas are only allowed at top level, but this is a purely syntactic restriction. Rewrite rules make perfect sense for nested functions bound by a local `let` or `letrec`, and GHC will indeed generate dynamic rules using the ideas of this section for local functions. This is important in practice, because inlining generates many nested function definitions.

6 Implementation

The implementation of the rule rewriting mechanism within GHC is straightforward. The front-end was extended to
handle rule parsing, type checking, and translation into the Core intermediate language. The GHC optimiser is structured as a number of separate passes over Core expressions (Peyton Jones and Santos, 1998; Peyton Jones and Marlow, 1999). The most fundamental pass — iterated many times — is the simplifier, which performs inlining, case simplification, and eta-expansion in the course of a single top-to-bottom traversal of the program. To support rewriting, we just modified the simplifier to check each function application it encounters against a list of active rules; if the application matches the rule LHS pattern, it is replaced by a suitably instantiated version of the RHS. We need to take a little care to make sure that the rule remains attached to the right function if alpha-renaming takes place.

Including rules adds a modest overhead to GHC compilation time. For example, using the list fusion rules described in Section 3 increases compilation times an average of 35% over the nofib benchmark suite. Some of this increase is probably due to performing conventional optimisations that are enabled by rule-based rewrites. In any case, we have made no serious attempt to analyse or optimise this aspect of compiler performance, so it can probably be speed up should this prove important.

7 Application: Constraint Satisfaction Problems

Next we give an example user application, solving constraint satisfaction problems (CSPs), in which rewrite rules help support high-level, modular programming style. The added rules, which describe short-cut deforestation on core trees, are confined to a library, and they make a representative kernel of the application run three times faster, by eliminating essentially all the overhead due to the modular style.

7.1 Modular search

Many interesting algorithms for solving CSPs are conceptually based on trees, whose nodes represent states in the search space; solutions to the search problem are found by locating complete, consistent nodes. In a conventional imperative recursive implementation, these search trees are merely notional; they correspond to the tree of procedure activation histories. In Haskell, one can make the state tree into an explicit (lazy) data structure instead (Hughes, 1989; Bird and Wadler, 1988). This approach permits search algorithms to be modularized into separate functions (really coroutines) that communicate via a lazily-constructed tree labeled with consistency information. The component functions perform generation of all possible states, consistency labeling, pruning of inconsistent states, and collection of solutions. A large variety of useful algorithms — which look quite different from one another when written imperatively — can be obtained in the lazy framework just by varying the labeling and pruning functions (Nordin and Tømch, 2000).

The underlying algorithm is a simple composition of functions, where all the intermediate results are trees or lists.

cvalidator :: Labeler a -> Pruner a -> CSP -> [State]
cvalidator pruner pruner pruner =
  (filter (complete pruner) . map fst . leaves .
   pruner pruner . (labeler pruner) .
   mkSearchTree) csp

Here CSP is a type describing instances of constraint satisfaction problems; for example, we might have a function

queens :: Int -> CSP

to generate instances of the familiar n-queens problem.

State is the type of partial solutions. Function

mkSearchTree :: CSP -> Tree State

constructs a tree of all possible partial solutions to a given CSP. Here Tree is the type of ordinary “rose trees,” in which each node has a value and an arbitrary number of children. The labeler argument to solver has this type:

type Labeler a =
  CSP -> Tree State -> [State]

It specifies how to attach consistency annotations to each node in the tree. The pruner argument, of type

type Pruner a = (State, a) -> Bool

says how to inspect the annotations to determine whether the node is consistent; prune removes subtrees rooted at inconsistent nodes. leaves returns the leaves of the tree as a list in left-to-right order. The subsequent list operations throw away the annotations and weed out nodes representing incomplete solutions.

To obtain simple back-tracking search, we can provide a Labeler that checks the consistency of each node individually, and annotates the node with the boolean result of the check.

labelInconsistencies ::
  CSP -> Tree State -> [State]

labelInconsistencies csp = mapTree f
  where f s = (s, not (consistent csp s))

btsolver :: CSP -> [State]
btsolver = solver labelInconsistencies snd

More sophisticated algorithms use labelers that may look at more than one node at a time or store more information in the annotations. For example, a well-known algorithm called forward checking can be implemented by a labeler that stores a (lazily constructed) cache table of consistency information at each node.

labelCSCache ::
  CSP -> Tree State ->
  Tree (State, Cache ConflictSet)

extractConflict ::
  (State, Cache ConflictSet) -> Bool

fcsolver :: CSP -> [State]
fcsolver = solver labelCSCache extractConflict

Interesting new combinations of algorithms can be obtained by appropriate composition of labeling functions, giving us a “mix and match” approach to algorithm construction. The modular algorithms that result are much simpler to read, write, and modify than their imperative counterparts, and have the same asymptotic behavior (in both space and time).

However, the modular Haskell code is much slower than equivalent C code, if only by a constant factor. We measured performance of a representative kernel of code that implements standard backtracking search on the n-queens problem and counts the number of solutions found. The modular version of this function is written
quote

On the 11-queens problem, *qsoins* runs about 30 times slower than a conventional recursive C algorithm that doesn’t use trees at all. More strikingly, perhaps, it is almost four times slower than a non-modular Haskell transliteration of the C algorithm. This difference suggests that we try to fuse the tree traversals to avoid building the nodes of the several intermediate trees.

In the remainder of this section, we describe short-cut deforestation for rose trees, and discuss our experience in using rules with this application. Full code for the kernel modular code and the corresponding monolithic function are given in the Appendix.

7.2 Fusion on rose trees

We treat rose trees as an abstract data type, with public functions *initTree*, *mapTree*, *prune*, and *leaves*. The internal representation data type and *foldTree* operation are standard:

```haskell
data Tree a = T a [Tree a]

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f t = go t (map f)
  where go (T a ts) = f a (map go ts)
```

We introduce a *buildTree* analogous to *buildOn*, and the corresponding fusion rule:

```haskell
buildTree ::
  forall a.
  (forall b. (a -> [b] -> b) -> b) -> Tree a

buildTree g = g T
```

{-# RULES
  "foldTree/buildTree"
  forall (g:forall b. (a->[b]->b) -> b)
    foldTree k (buildTree g) = g k
#-}

Now we must take care that all tree-producing functions use *buildTree*, and all tree-consuming functions use *foldTree*. Since *Tree* is as an ADT, we don’t need to worry about client code using the *Tree* constructor directly.

Function *initTree* generates a tree from a function that computes the children of a node (Hughes, 1989); *mapTree* is the analogue of the familiar functions on lists.

```haskell
initTree :: (a -> [a]) -> a -> Tree a
initTree f a = buildTree g
  where g n = go a
        where go a = n a (map go (f a))

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f t = buildTree g
  where g n = foldTree h t
        where h a ts = n (f a) ts
```

*prune* removes every subtree of *t* whose root value matches *p*. Since we cannot represent empty trees, we require that *p* always return *False* on the root node of the entire tree, which is always appropriate in our applications.

```haskell
prune :: (a -> Bool) -> Tree a -> Tree a
prune p t = buildTree g
  where
    g n = head (foldTree f t)
      where f a ts = if p a
                      then a
                      else n a (concat ts)
```

Finally, *leaves* extracts the values at the leaves of a tree into a list in left-to-right order.

```haskell
leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _ ts = concat ts
```

Ideally, we would like *leaves* to be written as a list *build*, so that it can fuse with list consumers further down the pipeline. Unfortunately, this seems to require doing a higher-order tree fold, which produces an intermediate list of function closures; GHC doesn’t handle such lists very effectively, and it proves more efficient to stick with the simple definition shown here.

We mark all the functions to be refined if possible.

7.3 Short-cut deforestation pays again

Given these definitions, GHC is able to completely fuse away all the rose trees in *qsoins*; i.e., no *T* constructors are applied at all! Indeed, modifying the implementation of our rose tree ADT to perform cheap deforestation improves performance of *qsoins* by a factor of more than three, bringing it to within 15% of the running time of a hand-rolled, non-modular Haskell implementation. Moreover, this improvement comes **without requiring any changes to the search application code itself**.

All is not quite so straightforward as it may seem, however. All the problems we examined in the context of list fusion appear again for trees:

* Effective application of the fusion law requires that GHC inline more enthusiastically than it normally would. For example, our pipeline of tree operations generates many fusion opportunities that require inlining underneath the lambda of a *buildTree* argument. This is, in fact, a safe thing to do, since the lambda is “one shot.” But GHC doesn’t know this – and since we are thinking of trees as a user-defined library, it would be obviously inappropriate to hack this fact about *buildTree* into the compiler, the way we did for list *build*. As it happens, for the particular kernel of code we show here, GHC can discover for itself – after repeated iteration of inlining – that these lambdas are one shot. But in general, we need linearity analysis.

* If fusion fails, the tree library should make sure that the resulting code is not worse than it would have been had fusion never been attempted. As with lists, we must either ensure that inlining *foldTree* produces good code, or provide a “back-out” mechanism, with appropriate attention to phasing of inlining (c.f. Section 4.2).

* For full effectiveness, we need to make sure that inlining of list functions (e.g., on the lists of children in nodes) occurs only after inlining of tree functions (c.f. Section 4.1). A simple phasing strategy based on module dependencies would handle this requirement.
• Most seriously, we might easily write programs for which fusion fails for legitimate reasons, e.g. because there are several consumers for a given producer, or simply because we've made a mistake when writing a rule. But we'll get no feedback from the compiler about such failures. This is clearly a crucial area for further work.

8 Related Work

The basic concepts of our rules system are far from new. There have been a great many attempts to build frameworks for user-directed or application-specific optimization, often by adding additional semantic specifications to functions. These ideas have been of particular interest in the high-performance computing community. Scientific codes often use well-established, high-level libraries, such as LINPACK or PLAPACK. Because these libraries need to work efficiently over a wide range of machine architectures and data sets, they typically have multiple implementations, each with its own complex interface. For portability and maintainability, client code should be written using portable, high-level library calls, leaving the compiler to determine the appropriate low-level calls to use and optimizing the client code accordingly. To achieve this, library interfaces can be annotated with additional specification information. Systems and proposals along these lines include TAMPR (Boyle et al., 1997). Broadway (Gayer and Lin, 1999; Gayer and Lin, 2000), MetaScript (Kennedy et al., 2000), and Active Libraries (Yelduan and Gannon, 1998).

Another set of systems has developed from the algebraic specification community. For example, the OPAL language (Didrich et al. 1994) combines functional programming and algebraic specification in a uniform framework: OPAL laws are used to justify or guard rewrite rules of functional code; since laws are first-order predicate formulas over equality of functional expressions, this makes the system very powerful (and of course undecidable). It is unclear to what extent the existing implementation of OPAL supports automated optimization.

Compared to existing systems and proposals, ours is notable primarily for what it leaves out. More precisely, we can identify the following contrasts between our systems and others:

No meta language. Our rules are source-to-source, and their right-hand sides are simple source expressions, so they can be defined just using Haskell. With the exception of TAMPR (Boyle et al., 1997), most of the other tools known to us operate on internal program representations, such as abstract syntax trees or control-flow graphs and they typically allow right-hand sides to be defined using some kind of meta-programming facility. The choice of a meta-programming language is delicate. A specialized language or notation such as metal (Engler et al., 2000) is concise, but must be learned from scratch by the library author and can be unduly constraining; using a general-purpose programming language such as LISP (as in early work on Aspect-Oriented Programming (Kiczales et al., 1997; Mendhekar et al., 1997)) is more flexible, but requires the author to take great care to maintain essential invariants.

Simple rewrite strategy. We rely on a very simple, built-in strategy, modified by "phases", for determining when and where rules should be applied. As rule sets become more elaborate, authors may need to exercise explicit control over strategy, e.g., as in Stratego (Visser et al., 1998).

Simple pattern-matching. We rely on the programmer to use high-level operators, such as foldr, that encapsulate control flow. Thus we don't need to provide sophisticated contextual pattern matching to identify loops or recursions, unlike systems like OTRAN (Lipps et al., 1988), Dora/Tess (Furnum, 1990), and KHEPERA (Faith et al., 1997). Nor do we have to deal with the unpredictability and possible high cost of higher-order matching, as used in MAG (de Moor and Sittampalam, 1999).

No side conditions. We work with a purely functional language, which means that many useful optimizing transformations are context independent and don't require elaborate side-conditions. By contrast, most useful transformations on imperative programs must be justified by non-syntactic, and often non-trivial, analysis, e.g., of control flow, dependence, aliasing, etc. Thus many tools for imperative languages focus on specifying analyses in addition to transformations; examples include DFM/OPT-MetaFrame (Klein et al., 1996), Shalit (Tjiang and Hennessey, 1992), Genesis (Whitefield and Soffa, 1994), OPTIMIX (Assmann, 1996), Intentional Programming (Aitken et al., 1998), and recent work of Lacey and de Moor (Lacey and de Moor, 2001).

No termination guarantees; no AC rewriting. Our rules are all directed, and we cannot easily express commutative laws without causing endless rewriting. In a modern algebraic transformation system like Maude (Clavel et al., 1996), equations are entirely symmetric in their left and right hand sides which can be arbitrary terms; they can be used for transformation in either direction. Common algebraic properties of an operator can be declared by built-in keywords such as [assoc] and [comm]; in executing the transformations in a program, all pattern matching is conducted modulo these properties, which makes for shorter and more elegant programs.

In summary, we offer simplicity in exchange for more limited functionality. Simplicity is important, both for implementors and library authors. From an implementation point of view, our experience is that simple ideas are seldom easy to implement in a full-scale, optimising compiler; while complex ideas require heroism that is hard to sustain in the long term.

From a programming point of view, too simplicity is important. Most particularly, the fact that the transformations are expressed entirely in Haskell itself, and not in some (necessarily different, and more indirect) meta-language is a huge advantage. We know of no optimising compiler in widespread use that supports domain-specific extensions; we suspect that this is partly due to the complexity of their
meta-programming mechanisms. Of course, GHC’s rules are not in widespread use by programmers either — but they are used behind the scenes in every run of GHC, both for list fusion (Section 3) and specialisation (Section 5). It is also possible that our approach is just too simple: we do not yet know how the tradeoff between simplicity and expressiveness will play out.

9 Conclusions and further work

We have described a simple, but fully implemented and deployed, way to write domain-specific extensions to a compiler for Haskell, by means of rewrite rules. We have demonstrated that, though simple, rewrite rules are useful in practice. Indeed, the list fusion rules have been deployed in the Prelude of the released GHC compiler for two years. In recent work, Chakravarty and Keller are using GHC’s rewrite rules to perform array fusion in their work on nested datal parallel programming (Chakravarty and Keller, 2001); their application is more sophisticated than any we have described here.

The previous section described many directions in which one could imagine make our system more expressive, but we plan to develop more experience of its practical use before elaborating it much further. Indeed, the most pressing area for further work is not even mentioned in Section 8: it is the question of how best to provide feedback to the programmer about which rules have fired and, more especially, which have not and why not. Since writes are done on Core, which is quite far from Haskell, providing comprehensible feedback is a hard problem.

The status of this paper is as a report of work in progress. We present it in the hope that it will attract the interest of the writers of library packages, and will encourage them to experiment with the feature and report on its inadequacies. For the longer term, we wish to promote the principle that a programmer should supply further declarative information together with the code of the program; and suggest that compilers and other programming tools should take maximum advantage of these declarations.

Acknowledgements

We gratefully thank Manuel Chakravarty, Andy Gill, Oege de Moor and Eelco Visser for their helpful feedback on earlier versions of this paper.

References


**Appendix: Constraint Satisfaction Problems**

Here is the complete code for the constraint satisfaction problem (CSP) search kernel described in Section 7

**Problem Definition**

A CSP is characterized by a number of variables vars, a number of values vals, and a consistency relation rel between pairs of assignments of values to vars. We represent assignments using an infix constructor :-: To solve the CSP, we must assign a value to each variable such that all pairwise combinations of assignments are in rel. A well-known example is the n-queens problem, under the standard optimization that we only try to place one queen in each column; this can be modeled as a CSP with n variables (the columns), n values (the rows), and a relation that permits two assignments provided the corresponding positions are on different rows or different diagonals.

```haskell
type Var = Int
type Value = Int

data Assignment = Var := Value

type Relation = Assignment -> Assignment -> Bool

data CSP = C {vars, vals :: Int, rel :: Relation}

queens :: Int -> CSP
queens n = C{vars=n, vals=n, rel=safe}
  where safe (col1 := row1) (col2 := row2) =
    (row1 /= row2) &&
    abs (col1 - col2) /= abs (row1 - row2)
```


Search States

We model each state in the space of possible solutions as a sequence of assignments, together with the number of the most recently assigned variable. States are built from `emptyState` by repeated use of `extensions`, which takes a state and constructs a list of extended states formed by assigning each possible value to the next variable.

```plaintext
data State = S [Assignment] Var
emptyState :: CSP -> State
emptyState C{vars=v} = S [] 0
extensions :: CSP -> State -> [State]
extensions C{vars=v, vals=v} (S as lastvar) =
  [S ((nextvar := val) as) nextvar | let nextvar = lastvar+1, nextvar < vars, val <- [1..vars]]
```

A solution is a complete, consistent state.

Rose Trees

Here is sample library code for rose trees written without concern for fusion. For convenience, we do use `foldTree` in the definition of `prune` and `leaves`.

```plaintext
data Tree a = T a [Tree a]
initTree :: (a -> [a]) -> a -> Tree a
initTree f a = go a
  where go a = T a (map go (f a))
foldTree :: (a -> [b]) -> Tree a -> b
foldTree f t = go t
  where go (T a cs) = f a (map go cs)
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (T a ts) = T (f a) (map (mapTree f) ts)
prune :: (a -> Bool) -> Tree a -> Tree a
prune p t =
  head (foldTree f t)
  where f a ts | p a = []
              | otherwise = [T a (concat ts)]
leaves :: Tree a -> [a]
leaves = foldTree f
  where f leaf [] = [leaf]
        f _ ts = concat ts
```

Backtracking Search for CSPs

```plaintext
mkSearchTree :: CSP -> Tree State
mkSearchTree csp =
  initTree (extensions csp) (emptyState csp)

type Labeler a =
  CSP -> State -> Tree (State, a)

type Pruner a = (State,a) -> Bool

labelInconsistencies :: Labeler Bool
labelInconsistencies csp = mapTree f
  where f s = (s, not (consistent csp s))

solver :: Labeler a -> Pruner a -> CSP -> [State]
solver labeler pruner csp =
  filter (complete csp) . map fst . leaves .
  prune pruner . (labeler csp) .
  mkSearchTree csp
```

```plaintext
tbtsolver :: CSP -> [State]
tbtsolver = solver labelInconsistencies snd

qsoins :: Int -> Int
qsoins n = length (tbtsolver (queens n))
```

Hand-fused Code

A hand-fused version of `qsoins` in Haskell:

```plaintext
qsoins' :: Int -> Int
qsoins' n = f (emptyState csp)
  where csp = queens n
        f state | complete csp state = 1
              | otherwise = g (extensions csp state)
        g [] = 0
        g (s':rest) | consistent csp s' = f s' + g rest
              | otherwise = g (s':rest) * g rest
```

Rose trees supporting fusion

The code for these was shown in Section 7.2.