The Implementation and Optimisation of Parallel Pattern Matching in Haskell

Lyndon While and Greg Mildenhall
School of Computer Science & Software Eng,
The University of Western Australia,
Western Australia 6009
email: {lyndon, gregm}@csse.uwa.edu.au

Tony Field
Department of Computing,
Imperial College London,
London SW7 2AZ
email: ajf@doc.ic.ac.uk

Abstract

Parallel pattern matching provides true commutative implementation of functions defined by cases in functional languages, because no argument is given precedence over any other. We present a formal semantics of parallel pattern matching and describe an implementation in Haskell that maps individual argument component matches into Concurrent Haskell threads. The performance of the Concurrent Haskell implementation is evaluated via benchmarking. The requirement for concurrency (in general) to support the semantics means that current implementations can often incur a significant performance penalty compared with sequential schemes, such as left-to-right matching. We describe a source-level program transformation scheme that analyses a parallel pattern matching definition and is often able to generate an equivalent definition that can be executed within a single thread. Where sequential implementation is not possible, the scheme is sometimes able to generate an equivalent definition that reduces the number of concurrent threads required.

Keywords: functional languages, pattern matching, semantics, concurrency, program transformation.
ACM computing classification: F.3.1, F.3.2, D.3.1.

1 Introduction

A key feature of most modern functional languages such as Haskell[Peyton Jones *et al*, 1999] is the ability to define algebraic data types and functions over those types using pattern matching. Pattern matching offers a number of benefits when dealing with structured concrete data types and operations over them. In particular, it encourages an equational style of programming and an associated style of reasoning, and it provides an elegant mechanism for testing and decomposing data structures.

The semantics of pattern matching concerns primarily the order in which equations, and their associated patterns, are tested. The first of these questions is less interesting for the purposes of this paper, but broadly there are two approaches: top-down matching, based on the original equation order; and best-fit matching, based on pattern specificity[Kennaway, 1990, Field *et al*, 1992, Courtenage and Poulovassilis, 1995].

The issue that we discuss here is the order of testing arguments within individual equations. Most currently available implementations of pattern matching use a sequential left-to-right ordering. This scheme is easily understood operationally, but it can sometimes conflict with the intuitive reading of a definition. Consider the Haskell function ||, which implements logical disjunction.

```
False || False = False
x     || y     = True
```

Syntactically, this definition is symmetrical in the two arguments, and a naive reading suggests that || is commutative. However, the pattern(s) in a Haskell equation are tested from left to right, so the second argument is evaluated only if the first argument matches. If the evaluation of the first argument fails to terminate or causes an error ($\perp$ in the semantics), then the program will behave likewise. Thus, True || $\perp$ = True but $\perp$ || True = $\perp$, so || is *not* commutative.

Parallel pattern matching (PPM), for example as discussed in [Plotkin, 1977, Field *et al*, 1992, While, 1994, Longley, 1999, While and Mildenhall, 2002], overcomes the asymmetry of sequential matching schemes. When matching a pattern against an argument value, the match fails unconditionally if *any* component of the pattern fails to match the corresponding component of the argument. The order in which the arguments of a function (and their components) are written becomes unimportant. With PPM the || function above is commutative since True || $\perp$ = $\perp$ || True = True. It can thus be argued that PPM provides a more intuitive semantics than any sequential scheme and that it simplifies some aspects of program transformation, specifically transformations that involve permuting arguments and/or restructuring equations.

Parallel pattern matching can also offer potential performance benefits in situations where pattern matching failure can be used to curtail unwanted work or avoid unnecessary delays. As an example, consider the following function that requires data from two input streams simultaneously, where each stream has a built-in timeout mechanism.

```
process (s1, True)  (s2, True)  = ... -- OK
process _           _           = ... -- timeout
```

Each stream is a pair whose first element comprises the stream data (for example from a remote sensor). The second element is set to True when the data becomes available (for example sensor reading OK) and False in the event of a timeout (for example sensor communication failure). The function is able to distinguish the case where both streams generated data from the situation where one or both streams timed out. Note that this is true regardless of the order of matching, provided the timeout signal faithfully delivers either True or False

in a finite time. The difference between a sequential scheme and the parallel scheme is behavioural, rather than semantic, and lies in the fact that the first equation will be aborted *as soon as* one or other stream times out. Under a sequential scheme we may wait for a long period of time whilst one stream gathers its data, even though the other may have already timed out. The advantage of parallel matching in this context is that the order in which the argument streams are applied is unimportant; PPM will always abort the match at the first timeout.

1.1 This Paper

We formalise the semantics of PPM and describe an implementation scheme for PPM via a source-level transformation into Concurrent Haskell[Peyton Jones *et al*, 1996]. When a pattern matches more than one of its arguments, a new process is created to evaluate and match each argument. If each individual match succeeds, then the overall match succeeds. If any of the individual matches fails, all of the unfinished processes are terminated and the overall match fails. The PPM semantics is satisfied, because no precedence is given to any of the individual matches: in particular, no assumption is made about the order in which the matching processes terminate.

It is important to understand that, for the purposes of this paper, parallel pattern matching is treated as *semantic* mechanism that ensures symmetry within equations. Our objective is not to improve performance through the exploitation of parallelism, cf. and-parallelism in Prolog.

Programs implemented with PPM semantics typically incur a performance penalty when compared with sequential matching schemes, due to the overheads of process management. In many cases, we can mitigate this performance penalty to some extent by transforming functions at the source level into equivalent forms that do not require concurrency for their evaluation. A transformation scheme for doing this is also developed in the paper.

The scheme works by considering all possible combinations of evaluation forced by the patterns of the definition, and identifying which arguments always cause errors when they are undefined. The scheme can also "partly sequentialise" a definition that cannot be run in a single thread, reducing the number of concurrent processes required to implement the PPM semantics. The scheme bears some similarity to the process of strictness analysis, but it has three significant advantages in the context of PPM:

- it is much simpler than strictness analysis;
- it naturally handles patterns with nested constructor applications;
- it can optimise some definitions that cannot be fully sequentialised, as described above.

Note that the optimisation scheme is independent of any particular implementation of PPM, and that it is likely to play an important role in any future implementation of PPM.

The paper makes the following contributions:

- We present a formal semantics for parallel pattern matching (Section 2.1) and show how this can be used as the basis of a translation scheme (Section 3.1) that maps parallel pattern matching equations into explicit calls to a multi-way parallel "and" function (ppm).
- We describe an implementation of ppm that executes each of its arguments within a separate Concurrent Haskell thread (Section 3).
- We evaluate the performance of the implementation using a small suite of benchmark programs (Section 4).
- We develop a source-to-source program transformation scheme that is capable of removing calls to ppm in many cases (Section 5); when all instances of ppm can be removed the pattern matching can be implemented within a single thread.

The remainder of the paper is organised as follows. Section 2 presents the background to the paper. In particular, it presents the semantics of parallel pattern matching and details those aspects of Concurrent Haskell that are required for the implementation details to be understood. Section 3 describes a Concurrent Haskell implementation of ppm and Section 4 evaluates its performance in comparison with Haskell's sequential (left-to-right) matching scheme. Section 5 describes the program transformation algorithm. Section 6 works through the analysis and transformation of an example function and Section 7 discusses some further issues with the transformation scheme. Section 8 concludes the paper and outlines some ideas for future work.

2 Background

2.1 Parallel Pattern Matching

We assume that a pattern takes one of three forms.

- An argument name, which matches anything.
- A constructed value, which matches any value built by the right constructor and whose arguments match the patterns in the corresponding arguments.
- A tuple, which matches any value whose fields match the patterns in the corresponding fields.

Thus, if P denotes the set of patterns,

$$\mathsf{P} ::= \mathsf{I} \mid \mathsf{C}\ \mathsf{P}_1 \ \ldots\ \mathsf{P}_n \mid (\mathsf{P}_1, \ \ldots, \ \mathsf{P}_m)$$

where I ranges over variable identifiers and C over constructor names.

2.1.1 Currying

Functions in Haskell are usually defined in curried form, for example

$$\mathsf{f}\ \mathsf{P}_1\ \ldots\ \mathsf{P}_n = \mathsf{E}$$

where the P_i are patterns and E is an expression. We will usually write functions in curried form but will assume that they are uncurried for the purposes of processing. Thus, when processing fun above, we will assume its definition to be

$$\mathsf{f}\ (\mathsf{P}_1, \ldots, \mathsf{P}_n) = \mathsf{E}$$

Note that this does not affect the behaviour of the pattern matching process. We work with uncurried definitions in order to simplify the formalisation of the semantics and of the various translation and transformation rules presented.

2.1.2 Semantics

Consider a function f, defined in an uncurried form by n pattern matching equations.

$$\begin{array}{l}\mathsf{f}\ \mathsf{P}_1 = \mathsf{E}_1\\ \mathsf{f}\ \mathsf{P}_2 = \mathsf{E}_2\\ \ldots\\ \mathsf{f}\ \mathsf{P}_n = \mathsf{E}_n\end{array}$$

The denotation of f above under top-down pattern matching in the environment ρ is given by the expression

$$\xi\ [\![\ \mathsf{f}\ \mathsf{P}_i = \mathsf{E}_i,\ 1 \le i \le n\ ]\!]\ \rho$$

where ξ is defined in Figure 1.

$$\begin{array}{lll}\xi\ [\![\ \mathsf{f}\ \mathsf{P}_i = \mathsf{E}_i,\ 1 \le i \le n\ ]\!]\ \rho\ \mathsf{v} = \xi\ [\![\ \mathsf{E}_k\ ]\!]\ (\mathsf{Bind}\ [\![\ \mathsf{P}_k\ ]\!]\ \mathsf{v}\ \rho), & & \text{if } k > 0\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad = \mathsf{error}, & & \text{if } k = 0\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad = \bot, & & \text{if } k = \bot\end{array}$$

$$k = \Delta_1\ \mathsf{v}\ (\Delta_2\ \mathsf{v}\ (\ \cdots\ (\Delta_n\ \mathsf{v}\ 0)\ \cdots\ ))$$

$$\begin{array}{lll}\Delta_i\ \mathsf{v}\ \mathsf{a} = i, & \text{if } \mathsf{M}\ [\![\ \mathsf{P}_i\ ]\!]\ \mathsf{v} = \mathsf{tt}\\ \qquad = \mathsf{a}, & \text{if } \mathsf{M}\ [\![\ \mathsf{P}_i\ ]\!]\ \mathsf{v} = \mathsf{ff}\\ \qquad = \bot, & \text{if } \mathsf{M}\ [\![\ \mathsf{P}_i\ ]\!]\ \mathsf{v} = \bot\end{array}$$

$$\begin{array}{lll}\mathsf{M}\ [\![\ \mathsf{x}\ ]\!]\ \mathsf{v} = \mathsf{tt}\\ \mathsf{M}\ [\![\ \mathsf{C}\ \mathsf{P}_1\ \ldots\ \mathsf{P}_m\ ]\!]\ \mathsf{v} = \mathsf{M}\ [\![\ (\mathsf{P}_1, \ldots, \mathsf{P}_m)\ ]\!]\ (\mathsf{v} \downarrow 1, \ldots, \mathsf{v} \downarrow m), & & \text{if } \mathsf{tag}\ \mathsf{v} = \mathsf{t}_C\\ \qquad\qquad\qquad\qquad\qquad\qquad\quad = \mathsf{ff}, & & \text{if } \mathsf{tag}\ \mathsf{v} \neq \mathsf{t}_C\\ \mathsf{M}\ [\![\ (\mathsf{P}_1, \ldots, \mathsf{P}_m)\ ]\!]\ \mathsf{v} = \mathsf{M}\ [\![\ \mathsf{P}_1\ ]\!]\ (\mathsf{v} \downarrow 1) \otimes \cdots \otimes \mathsf{M}\ [\![\ \mathsf{P}_m\ ]\!]\ (\mathsf{v} \downarrow m)\end{array}$$

Figure 1: The semantics of top-down pattern matching. Bind extends the environment with the new bindings resulting from a successful match, tag returns the numeric representation of a constructor, and $\downarrow$ is used to index tuples and constructed data.

The combination of matching-results across tuples is specified by the $\otimes$ operator. Varying the definition of this operator varies the strictness and directionality of the semantics. For example, the definitions of $\otimes$ for left-to-right pattern matching (as in standard Haskell), for strict pattern matching (which restores commutativity at some cost in laziness), and for parallel pattern matching are shown in Figure 2.

The parameterised semantics of Figure 1 is closely related to the semantics given in [Field *et al*, 1992, While, 1994, While and Mildenhall, 2002].

$\mathbf{x}$	$\mathbf{y}$	$\mathbf{x} \otimes_{LtoR} \mathbf{y}$	$\mathbf{x} \otimes_{strict} \mathbf{y}$	$\mathbf{x} \otimes_{ppm} \mathbf{y}$
tt	tt	tt	tt	ff
tt	ff	ff	ff	ff
ff	tt	ff	ff	ff
ff	ff	ff	ff	ff
tt	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	tt	$\bot$	$\bot$	$\bot$
ff	$\bot$	ff	$\bot$	ff
$\bot$	ff	$\bot$	$\bot$	ff
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Figure 2: The definitions of $\otimes$ for various pattern matching semantics.

2.2 Concurrent Haskell

Concurrent Haskell[Peyton Jones *et al*, 1996] is an extension to Haskell that supports the scheduling of multiple processes and communication between them. It provides primitives for process creation and termination, synchronisation of concurrently-evaluating processes, and inter-process communication via atomically-mutable shared-state objects.

We describe here only the aspects of Concurrent Haskell that are relevant to the sequel. A basic understanding of Haskell's IO is assumed. We direct the reader to [Peyton Jones *et al*, 1996] for further details.

2.2.1 Processes

Concurrent Haskell provides the following primitive for creating processes.

```
forkIO :: IO ( ) -> IO ThreadId
```

forkIO takes an argument x of type IO ( ) and creates a new process to execute x. Any IO actions associated with x are executed as part of the new process, and are not sequenced with the IO activity of the parent process. The new process therefore executes concurrently with the parent process. forkIO returns an IO value (so the action of forking is sequenced with the other IO actions of the parent process) which encapsulates a value h of type ThreadId. h is used as a handle by which the parent process can refer to the child process.

The function killThread :: ThreadId -> IO () can be used to terminate a process generated by forkIO.

2.2.2 Channels

We implement inter-process communication using the Chan data type of Concurrent Haskell. A value of type Chan a is a (usually shared) reference to a FIFO data-stream (or *channel*) holding values of type a. The state of a channel can be manipulated using several pre-defined actions.

```
newChan :: IO (Chan a)

writeChan :: Chan a -> a -> IO ( )

readChan :: Chan a -> IO a
```

newChan generates a new channel which can hold values of type a. It returns a handle that can be used to refer to the new channel. writeChan takes a Chan ch and a value x, and appends x to the end of the channel referred to by ch. readChan takes a Chan ch and reads the next value from the channel referred to by ch, removing the value from the stream. If there are no values in the channel when readChan ch is executed, the executing process blocks until a value is written to ch by some other process, at which time it is rescheduled. This "blocking read" is the mechanism that we use to force processes to wait for the evaluation of function arguments.

2.2.3 Bracketing

The function

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

can be used to enclose some primary action between pre- and post-handling functions. In the evaluation of the expression

```
bracket before after action
```

the value before is evaluated first, with its result passed to both after and action. Next, the function action is executed, and its result (or any exception it raises) becomes the return value of bracket. Finally, *whether* action *raised an exception or not*, the function after is executed. Asynchronous exceptions are blocked during the execution of both before and after, so their operations will always be performed. However, if an exception is raised while action is being executed, it terminates, and control passes immediately to after.

2.2.4 unsafePerformIO

Because our pattern matching mechanism performs IO activities, it returns an IO value encapsulating the Boolean result of the match, i.e. a value of type IO Bool. Most functions, however, do not have an IO return-type. Since we do not want to change the type of any function, we need a way to extract the encapsulated Boolean value from the IO value returned. GHC[Marlow et al, 2001] provides the unsafePerformIO function for this situation.

```
unsafePerformIO :: IO a -> a
```

unsafePerformIO takes an argument of type IO a and returns the encapsulated value of type a, which is exactly what we require. But unsafePerformIO must be used with care. Extracting an encapsulated value in this way "hides" the IO actions that generated the value, so they cannot be sequenced with other IO actions in the program. There is no way of controlling (or even predicting) the order in which such disconnected streams of actions will be interleaved.

So under what conditions is it "safe" to use unsafePerformIO? When the actions hidden inside its argument are entirely self-contained: there must be no side-effects associated with the actions, for example with respect to the file system. The actions performed by our implementation of parallel pattern matching are restricted to the creation and termination of local processes, communicating only locally via newly-created channels (see Section 3). This means that we can use unsafePerformIO with impunity to invoke our pattern matching mechanism anywhere in a program.

3 Parallel Pattern Matching in Concurrent Haskell

To illustrate how PPM is implemented in our scheme, we first revisit || from above. The idea is to replace pattern matching in the original equation with a guard which returns True iff the original pattern would match given the same argument(s). The definition that we generate for || is shown in Figure 3.

```
(||) :: Bool -> Bool -> Bool
False || False = False
x     || y     = True

- - - - - - - - - - - - - - - - - - - -

(||) :: Bool -> Bool -> Bool
x || y | ppm [not x, not y] = False
       | otherwise          = True
```

Figure 3: The symmetrical definition of || from Section 1, and the Concurrent Haskell definition, implementing parallel pattern matching.

The pattern matching in the original definition is replaced by the guard

```
ppm [not x, not y]
```

This guard should return True iff both of the expressions on the argument to ppm are True, i.e. iff both x and y are False. More importantly, it should return False if either expression is False, even if the other one causes an error or fails to terminate.

In general, the call ppm [a_1, ..., a_n] will return False if *any* of the a_i is False, even if one or more of the a_j, $i \neq j$, gives and error or fails to terminate.

```
module PPM (ppm)

where

import Concurrent
import IOExts

ppm :: [Bool] -> Bool
ppm = unsafePerformIO . ppm'

ppm' :: [Bool] -> IO Bool
ppm' bs =
   do ch <- newChan
      bracket
         (mapM (forkIO . report ch) bs)
         (mapM killThread)
         (foldr (const (check ch)) (return True))

report :: Chan Bool -> Bool -> IO ( )
report ch True  = writeChan ch True
report ch False = writeChan ch False

check :: Chan Bool -> IO Bool -> IO Bool
check ch a = do h <- readChan ch
                if h then a else return False
```

Figure 4: The parallel pattern matching module.

The Concurrent Haskell definition of ppm is shown in Figure 4. The argument to ppm is a list of Bools bs. ppm simply calls the function ppm' and "unwraps" the result using unsafePerformIO. ppm' performs the following four actions, in the order listed. The process structure is illustrated in Figure 5.

1. ppm' creates a new channel ch.
2. The first argument to bracket calls forkIO twice, creating two processes p0 and p1 to independently evaluate the two elements on bs. Each process pi has the form

   ```
   report ch (bs !! i)
   ```

   The function report forces the evaluation of its second argument (using pattern matching!), then writes the result on ch.

   The result of this step is a handle hs to the list of processes created.

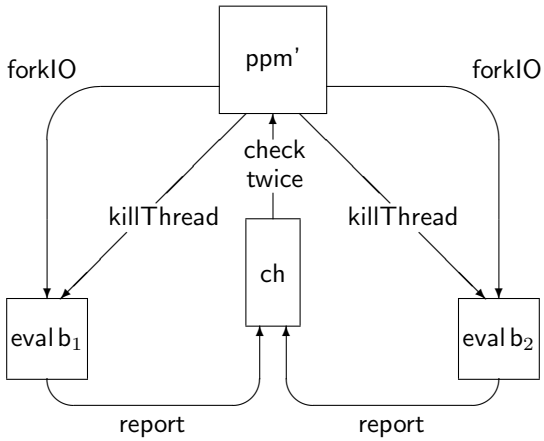

Figure 5: The process and communication structure created for ppm' [b_1, b_2]. ch is the channel created by ppm'.

3. The third argument to bracket waits for one of the processes to complete. The application of foldr creates the expression

   ```
   check ch (check ch (return True))
   ```

   The outer application of check waits for a value to be written on ch. If the value written is False, the result is return False. If the value written is True, the result is its second argument, i.e. check ch (return True). This application of check then waits again for a value to appear: on False, the result again is return False, but on True, this time the second argument is return True. Thus a False from either process causes the result to be False: True is returned only if both processes write True to ch.

   Note that the processes can return in either order: check only looks for a value on ch, it does not care which process writes the value to the channel. Note also that if either process has an error in its execution, and the other process writes True on ch, the system eventually reaches a situation where ppm' is waiting on ch, but no process exists to write to it. The system is able to recognise this deadlock situation automatically, and it returns an error, as required by the semantics.

4. The second argument to bracket tells any remaining processes to kill themselves. If one of the processes returned False, the other process may still be alive, but its result is no longer required. This step is particularly important if one of the processes invokes an infinite computation.

   Note that the work done by any unfinished processes is not necessarily wasted: any expressions that were being evaluated by an unfinished process will be partly-expanded in the program graph. If such an expression must be evaluated later in the program execution, the work done at this stage will not have to be repeated.

If the process is interrupted by its parent whilst doing Step 3, its result is no longer required and it should kill itself. The effect of bracket is that the process skips straight to Step 4: it tells its own children to kill themselves, then it terminates gracefully. This mechanism guarantees that all processes will be killed.

The final point is that if the program graph contains multiple applications of ||, each one will create its own invocation of ppm. These separate invocations of ppm behave entirely independently. Each invocation creates its own channel to control its own set of processes: there is no communication between the processes created by one invocation and those created by another. Interaction is possible only if two processes from different invocations attempt to evaluate the same expression: this is dealt with automatically by the Concurrent Haskell system.

3.1 The Translation Algorithm

We now give a description of the translation algorithm that maps parallel pattern matching into explicit calls to ppm. We assume that the input is a function f with k arguments, defined by n equations.

$$
\begin{aligned}
&\mathsf{f}\ \mathsf{P}_{11}\ \ldots\ \mathsf{P}_{1k} = \mathsf{E}_1\\
&\mathsf{f}\ \mathsf{P}_{21}\ \ldots\ \mathsf{P}_{2k} = \mathsf{E}_2\\
&\ldots\\
&\mathsf{f}\ \mathsf{P}_{n1}\ \ldots\ \mathsf{P}_{nk} = \mathsf{E}_n
\end{aligned}
$$

To simplify the exposition, we assume that the equations have no guards.

The translated definition is composed of three classes of functions.

The auxiliary functions perform the pattern matching.

The result functions ensure that the bindings from the original equations do not need to be changed.

The root function replaces the original definition.

We describe the derivation of each of these classes of functions below.

3.1.1 The Auxiliary Functions

An auxiliary function is defined to test explicitly each of the constructor patterns in the equations of f. The recursive operator T below builds an expression that tests each type of pattern, introducing new definitions as required.

```
T [[ y ]] x                          ⇒ True
T [[ C P1 ... Pm ]] x                ⇒ g x
                                       where g (C x1 ... xm) = T [[ (P1, ..., Pm) ]] (x1, ..., xm)
                                             g _             = False
T [[ (P1, ..., Pm) ]] (x1, ..., xm) ⇒ ppm [T [[ P1 ]] x1, ..., T [[ Pm ]] xm]
```

Each auxiliary function g must be given a unique name within its context. These auxiliary functions test the presence, or otherwise, of a given constructor. They are used in the translation to generate the guards that replace the original pattern matching (see below). Note that the overall form of T reflects the form of the matching operator M from the semantics in Figure 1.

Several transformation rules can be applied to simplify the definitions generated and these will be assumed in the examples that follow. For example,

```
ppm [ ]                                ⇒ True
ppm [x]                                ⇒ x
ppm [x, ..., x', True, y, ..., y']     ⇒ ppm [x, ..., x', y, ..., y']
ppm [x, ..., x', ppm [z, ..., z'], y, ..., y'] ⇒ ppm [x, ..., x', z, ..., z', y, ..., y']
```

Brevity can also be enhanced by using built-in functions to test some of the built-in data types, for example id and not for Bool, and null and not . null for lists. For user-defined data types, often many auxiliary functions are identical, and this repetition can of course be eliminated.

3.1.2 The Result Functions

The result functions duplicate the original equations: the i^{th} equation from f gives one result function f'i. The result functions are invoked only when the result of the matching is already known, so there is no possibility of a pattern matching failure. Thus, in general, we produce n replicate equations of the form

```
f'1 P11 ... P1k = E1
f'2 P21 ... P2k = E2
...
f'n Pn1 ... Pnk = En
```

An explicit result function is required for an equation only if that equation uses pattern-bound names on its right-hand side. For example, no result functions are required in Figure 3, because no names are used on the right-hand sides of the original equations.

3.1.3 The Root Function

The root function replaces the original definition. It invokes the auxiliary functions to perform the pattern matching, and it calls the appropriate result function when the matching-result is known.

```
f x1 ... xk | T [[ (P11, ..., P1k) ]] (x1, ..., xk) = f'1 x1 ... xk
            | T [[ (P21, ..., P2k) ]] (x1, ..., xk) = f'2 x1 ... xk
            | ...
            | T [[ (Pn1, ..., Pnk) ]] (x1, ..., xk) = f'n x1 ... xk
```

The root function is thus responsible for replacing the pattern matching by guards (calls to ppm) for the purposes of equation selection. Notice that the right-hand sides contain some redundancy in that the pattern matching is repeated within the result functions (Section 3.1.2). This source-level treatment of parallel matching means that this redundancy cannot be avoided in general, although some optimisations are possible in simple cases.

It is relatively easy to show that the translation preserves the semantics of each equation, by a structural induction over patterns, although we do not provide details. Observe in particular that the ppm function corresponds to an n-way generalisation of the $\otimes$ operator in Figure 1.

4 The Performance of PPM

The introduction of parallel matching introduces thread management overheads when compared with sequential matching schemes such as left-to-right. We now evaluate these overheads by running a small suite of benchmarks using both parallel and left-to-right matching schemes.

Because the translation algorithm above works at the source level, some additional overheads may be introduced, for example calls to ppm with list arguments, auxiliary predicates that implement constructor testing, and repeated matching of arguments. In order for the comparison to be fair, we compare the execution time of the transformed code using the supplied definition of ppm with that of the same code where the calls to ppm have been replaced by calls to and; the latter processes its argument from left to right and so mimics the process of left-to-right matching. All benchmarks were run on a dedicated 2GHz Intel Xeon PC with 2GB RAM running Linux Mandrake 10.2 and using GHC 6.5. The timings were averages over 5 runs.

To stress-test the framework we first consider a recursive function stress that, under parallel matching semantics, requires one ppm per user-defined function application:

```
stress True True n = if n == 1 then True else stress True True (n – 1)
```

For n ≥ 1, stress True True n induces n function calls, each requiring an invocation of ppm. The translation generates the following:

```
stress b1 b2 n | ppm [b1, b2] = if n == 1 then True else stress True True (n – 1)
```

The sequential version of stress is obtained by replacing ppm with and. The execution times for left-to-right matching (Seq) and PPM (Par) are shown in Table 1. Because every user-defined function call invokes ppm

$n/10^6$	Seq (s)	Par (s)
1	0.18	9.54
5	0.88	47.62
10	1.78	95.51

Table 1: Execution times for stress benchmark.

the overheads are substantial, compared to sequential matching. However, this is an extreme example.

In most contexts parallel matching will be invoked relatively infrequently. As an example, we next consider Dijkstra's all paths shortest path algorithm[Dijkstra, 1959]. The relaxation phase of the algorithm adds two path costs, each of which may be either finite (the cost of moving from one node to another where a path has been found) or infinite (no path has yet been found). In the implementation we benchmark here, these path costs are represented by objects of type Maybe Int. Thus:

```
addCosts (Just w) (Just w') = Just (w + w')
addCosts w        w'        = Nothing
```

This function is the only non-sequential function in the program. That is, the denotational meaning of all other functions is the same under both sequential and parallel matching.

After applying some straightforward optimisations, described above, and renaming the generated functions to aid readability, the translator maps this to the following:

```
addCosts c c' | ppm [just c, just c'] = res c c'
              | otherwise             = Nothing
  where res (Just w) (Just w') = Just (w + w')
        just (Just w) = True
        just _        = False
```

Notice that res replicates the matching in the original definition of addCosts. It is necessary as the right-hand side (Just (w + w')) requires bindings for both w and w'. The auxiliary function just is used to match each argument.

Again, a sequential version of addCosts can be generated by replacing ppm with add. Table 2 compares the sequential and parallel execution times for a range of input graphs. The benchmark parameter n is a scaling factor (for a problem of size n, the graph contains $6n$ nodes and $13n - 1$ edges).

n	Seq (s)	Par (s)
20	0.35	0.44
40	2.71	3.08
60	9.22	10.10
80	22.12	23.27
100	42.70	44.90

Table 2: Execution times for Dijkstra benchmark.

As we noted earlier, parallel matching can sometimes improve the performance of a program – specifically in contexts where unnecessary work can be terminated as a side effect of parallel pattern matching. An example with timeouts was presented in Section 1. Here, we consider a similar example: a search function that determines whether a given search string is a member of *two* given relations. Each relation is a mapping from strings to integers. In the following code prefixand is defined syntactically in the same way as Haskell's built-in (&&), but (crucially) with different semantics.

```
search s r1 r2 = prefixand (mem s r1) (mem s r2)
                 where mem s r = elem s (map fst r)

prefixand True True = True
prefixand b1   b2   = False
```

After translation, prefixand becomes:

```
prefixand b1 b2 | ppm [b1, b2] = True
                | otherwise    = False
```

search is unaltered as it performs no pattern matching.

Under sequential matching the whole of r1 is searched before r2. Thus, if s is absent from both r1 and r2 the execution time will be proportional to the sum of the lengths of r1 and r2. Under parallel matching the

$\|r2\|/10^3$	Time (s)
1	0.12
10	0.59
100	5.01
460	23.64
1000	51.61

Table 3: Execution times for search benchmark.

execution time will be proportional to the length of the smaller of r1 and r2. Table 3 shows the execution times in this case under parallel pattern matching. The time reported is for 100 searches when $|r1| = 10^6$ and where $|r2|$ varies as shown in the table. The average sequential execution time in each case was 23.44s. When r2 is substantially smaller than r1 parallel matching can be faster than sequential matching, specifically when the thread responsible for computing mem s r2 above terminates (with False) before that of mem s r1 and kills the latter. The break-even point is around $|r2|=4.6 \times 10^5$.

5 Optimising PPM

The figures above show that PPM can be expensive in situations, where functions requiring ppm are invoked in high proportion relative to other function calls. The optimisation is based on the observation that calls to ppm can sometimes be avoided by transforming functions at the source-level to avoid the need for concurrency. Consider the function f.

```
f [ ] (y : ys) = 47
f xs (y : ys) = 48
f xs [ ]      = 49
```

Under PPM, f is equivalent to (it always returns the same result as) the function f', which has an obvious sequential implementation:

```
f' xs (y : ys) = f'' xs
f' xs [ ]      = 49

f'' [ ] = 47
f'' xs  = 48
```

f can thus be implemented under PPM with no concurrency. In this section we develop a transformation scheme that is capable of deriving optimised definitions such as the one above. This transformation scheme was first described in [While and Field, 2005].

5.1 Pattern-tables

The transformation works by exploiting a restricted form of strictness analysis. We construct a *pattern-table* for the definition that enumerates all possible combinations of evaluation that may be forced for each pattern component, together with the number of the equation that will be selected in each case. Equations are numbered from 1 in the obvious way, pattern matching failure is denoted by 0, and a result of $\bot$ is denoted by −1. The entries in a pattern-table are always mutually disjoint and where a case does not require an argument to be evaluated, that argument appears as an anonymous variable.

Given the pattern-table for a definition, we can identify any arguments (or components) which are always required to choose an equation. If an argument x is always required, then whenever $x = \bot$, the result of the function application will always be $\bot$. Specifically, x is always required if, in the pattern-table, every form with $x = \bot$ has the result −1.

Once we identify an argument that is always required, we construct a definition that matches *only* that argument, and where each equation calls an subsidiary function to match the rest of the arguments. We split the pattern-table up between the subsidiary functions in order to derive their definitions in recursive fashion. In general, the base case of this recursion is when a pattern-table contains no further matching.

The pattern-table for a definition is built-up in two stages. First a table is created separately for each equation, then the tables are combined in a manner that mirrors the top-down nature of the pattern matching process. We illustrate the two stages with the example f above.

The pattern-table for the first equation of f conveys the following information:

```
f [ ]      [ ] = 0     f [ ]      (_ : _) = 1     f [ ]      ⊥ = −1
f (_ : _)  [ ] = 0     f (_ : _)  (_ : _) = 0     f (_ : _)  ⊥ = 0
f ⊥        [ ] = 0     f ⊥        (_ : _) = −1    f ⊥        ⊥ = −1
```

Each argument has three possible values (including $\bot$), so the table has nine entries. Note that all of the entries are disjoint. The number of 0s in the table is a result of the promotion of failure in the semantics.

The pattern-table for the second equation is simpler, as it matches only one argument:

```
f _ [ ] = 0        f _ (_ : _) = 2        f _ ⊥ = −1
```

The table for the third equation is also straightforward:

```
f _ [ ] = 3        f _ (_ : _) = 0        f _ ⊥ = −1
```

5.1.1 Generating Pattern-tables

The components of a pattern-table are generated by a function G, detailed below, that takes a pattern and an equation number as parameters. The result is a set of pairs, each of the form (p,v), where p is a pattern and v is one of -1 ($\bot$), 0 (fail) or $i > 0$ (successful match with equation i).

In the examples above, the pattern-tables have been presented as sugared forms of output from G. For example, a call to G that generates $\{(p_1,v_1), (p_2,v_2), \ldots, (p_n,v_n)\}$ corresponds to the pattern-table

$$\begin{array}{l} \mathsf{fun}\ p_1 = v_1 \\ \mathsf{fun}\ p_2 = v_2 \\ \ldots \\ \mathsf{fun}\ p_n = v_n \end{array}$$

where fun is the name of the function being processed. We will continue to present sample output from G in this sugared form to aid readability.

$$\begin{array}{lll} \mathsf{G}\ [\![\ \mathsf{x}\ ]\!]\ \mathsf{i} & \Rightarrow & \{(\_, \mathsf{i})\} \\ \mathsf{G}\ [\![\ \mathsf{C}\ P_1\ \ldots\ P_n\ ]\!]\ \mathsf{i} & \Rightarrow & \{(\bot, -1)\} \cup \{(\mathsf{C'}\ \_\ \ldots\ \_, 0) \mid \mathsf{C'} \in \mathsf{Con}(\tau), \mathsf{C'} \neq \mathsf{C}\} \cup \\ & & \{(\mathsf{C}\ P_1\ \ldots\ P_n, \mathsf{v}) \mid ((P_1, \ldots, P_n), \mathsf{v}) \in \mathsf{G}\ [\![\ (P_1, \ldots, P_n)\ ]\!]\ \mathsf{i}\} \\ & & \text{where } \tau = \mathsf{Type}(\mathsf{C}) \\ \mathsf{G}\ [\![\ (P_1, \ldots, P_n)\ ]\!]\ \mathsf{i} & \Rightarrow & \{((p_1, \ldots, p_n), \mathsf{LUB}\ \{v_1, \ldots, v_n\}) \mid (p_k,v_k) \in \mathsf{G}\ [\![\ P_k\ ]\!]\ \mathsf{i},\ 1 \leq k \leq n\ \} \end{array}$$

A variable generates a table with 1 row with no matching, returning success. Note that Haskell disallows non-linear patterns (patterns where a variable is used multiple times).

An application of constructor C generates one row (i.e. one pair in the resulting set) for the $\bot$ case (–1, i.e. non-termination), one row for each non-matching constructor C' in the same data type as C (0, i.e. fail), and a set of additional rows for the matching constructor, generated by recursively processing the constructor arguments. In the corresponding rule for G below, Type(C) returns the data type of which constructor C is a member and Con(τ) returns the set of constructors of type τ.

For a tuple with n fields, G is first applied to each field in turn, each generating its own pattern sub-table (itself a set of pairs). The individual patterns of these sub-tables are combined to form all possible patterns for the original n-tuple; for each such pattern the value associated with it is the least upper bound (LUB) of the values associated with the corresponding individual patterns. The LUB computation mirrors $\otimes$ in the semantics and ensures that failure is promoted maximally. LUB S computes the smallest element of S under the partial ordering $0 < -1 < \ldots$

For example,

- h (x : xs) generates a table with three rows:

  $$\begin{array}{ll} \mathsf{h}\ \bot & = -1 \\ \mathsf{h}\ [\,] & = 0 \\ \mathsf{h}\ (\_ : \_) & = 1 \end{array}$$

  One pattern with $\bot$, one pattern with the wrong constructor, and one pattern with the right constructor. In this case the table components are generated by G [[(x : xs)]] 1 which returns the set $\{(\bot,-1), ([\,],0), ((\_ : \_),1)\}$. The equation number is assumed to be 1.

- h (x : (x' : xs)) generates a table with five rows:

  $$\begin{array}{ll} \mathsf{h}\ \bot & = -1 \\ \mathsf{h}\ [\,] & = 0 \\ \mathsf{h}\ (\_ : \bot) & = -1 \\ \mathsf{h}\ (\_ : [\,]) & = 0 \\ \mathsf{h}\ (\_ : (\_ : \_)) & = 1 \end{array}$$

  One pattern with $\bot$, one pattern with the wrong constructor, and three patterns with the right constructor at the top-level, reflecting the matching in the second argument to :.

- h ((x : xs) : (xs' : xss)) generates a table with eleven rows:

  $$\begin{array}{ll} \mathsf{h}\ \bot & = -1 \\ \mathsf{h}\ [\,] & = 0 \\ \mathsf{h}\ (\bot : \bot) & = -1 \\ \mathsf{h}\ ([\,] : \bot) & = 0 \\ \mathsf{h}\ ((\_ : \_) : \bot) & = -1 \\ \mathsf{h}\ (\bot : [\,]) & = 0 \\ \mathsf{h}\ ([\,] : [\,]) & = 0 \\ \mathsf{h}\ ((\_ : \_) : [\,]) & = 0 \\ \mathsf{h}\ (\bot : (\_ : \_)) & = -1 \\ \mathsf{h}\ ([\,] : (\_ : \_)) & = 0 \\ \mathsf{h}\ ((\_ : \_) : (\_ : \_)) & = 1 \end{array}$$

  One pattern with $\bot$, one pattern with the wrong constructor, and nine patterns with the right constructor at the top-level, reflecting the product of the matching in each argument to :.

- h (x : xs, xs' : xss) generates a table with nine rows:

  h (⊥, ⊥) = −1
  h ([ ], ⊥) = 0
  h (_ : _, ⊥) = −1
  h (⊥, [ ]) = 0
  h ([ ], [ ]) = 0
  h (_ : _, [ ]) = 0
  h (⊥, _ : _) = −1
  h ([ ], _ : _) = 0
  h (_ : _, _ : _) = 1

  The tuple pattern (x : xs, xs' : xss) is precisely that generated when G is used to process the topmost application of : in the previous example. Recall from the second rule for G that constructor arguments are assembled into a tuple for the purposes of transformation.

5.1.2 Combining pattern-tables

The tables for the individual equations are combined in a process that mirrors the top-down semantics of the pattern matching process in Figure 1. Given a sequence of n tables T_i, $1 \leq i \leq n$, the table for the entire definition is given by foldr1 topdown [T_1, ..., T_n], where topdown combines tables pairwise. foldr1 "inserts" applications of its first argument between the elements on its second argument, starting from the right. For example, for $n = 3$, this is equivalent to topdown T_1 (topdown T_2 T_3). The application topdown T_j T_k inspects each entry (p,v) in T_j, and:

- if v ≠ 0 (i.e. success or ⊥: nothing further to be done), e = v is included in the result;
- if v = 0 (i.e. failure: the subsequent equations should be tested), topdown finds each entry p' = v' in T_k where p' overlaps p and includes the *unification*, unify(p,p'), of p and p' in the result.

unify(p,p') delivers the smallest pattern p'' for which M [[p]] v and M [[p']] v implies M [[p'']] v. Note that topdown mirrors the function Δ in the semantics of Figure 1.

Applying topdown to the tables for the second and third equations of f is simple, as they have identical patterns in each row. This operation yields

f _ [ ] = 3 f _ (_ : _) = 2 f _ ⊥ = −1

Applying topdown to the table for the first equation and the above table yields the final pattern-table for f, shown in Figure 6.

f [ ]	[ ] = 3	f [ ]	(_ : _) = 1	f [ ]	⊥ = −1
f (_ : _)	[ ] = 3	f (_ : _)	(_ : _) = 2	f (_ : _)	⊥ = −1
f ⊥	[ ] = 3	f ⊥	(_ : _) = −1	f ⊥	⊥ = −1

Figure 6: The final pattern-table for f.

Tables grow exponentially in size with the number and/or complexity of the patterns, but real patterns are always small, so the size of the tables will not be a significant issue in practice.

5.2 Analysing Pattern-tables

The final stage of the transformation process is quite involved, so we provide to an informal description of the key steps. We illustrate the process using the pattern-table from Figure 6.

Step 1: We first construct a list of "positions" being matched at the top-level. These positions represent the arguments (or the tuple-fields) that may need to be evaluated to match the argument. "Top-level" here excludes nested matchings: obviously a constructor must be matched before its arguments can be matched. Consider a pattern-table containing the row

h _ (_ : [ ]) (_, [ ]) = v

With arguments and fields numbered from the left in the obvious way, the three constructor applications here are at positions [2], [2,2], and [3,2]. But the first [ ] is nested inside the application of :, so it is not at the top-level, and it can be ignored. Thus this row contains two top-level matching positions.

The top-level positions from Figure 6 are [1] and [2].

Step 2: We next find out which top-level positions (if any) are strict. We examine the pattern-table for each top-level position: a position is strict iff for every row in the table where that position is ⊥, the return-value is −1.

In Figure 6, position [1] is not strict because f ⊥ [ ] = 3. This is easily seen from the original definition of f: if the second argument is [ ], the first and second equations will fail without examining the first argument, and the third equation will match, again without examining the first argument.

However, position [2] *is* strict because f xs ⊥ = −1, ∀xs :: [a]. It is impossible to reject both of the last two equations without examining the second argument.

Step 3: For the first strict top-level position we now split the pattern-table into sub-tables for the distinct constructors being matched. For a strict position matching a constructor whose type has n constructors, split the table into n sub-tables, one for each constructor in the type. Note that a partial definition may be undefined for some constructors. This means that all of the rows in the corresponding sub-table will return failure (0). Clearly such sub-tables can be discarded.

The table from Figure 6 gets split into two tables:

```
g1 [ ]      _ _ = 1          g2 [ ]      = 3
g1 (_ : _)  _ _ = 2          g2 (_ : _)  = 3
g1 ⊥        _ _ = -1         g2 ⊥        = 3
```

Notice that the matched constructor applications are replaced by their arguments (if any).

Step 4: For each remaining sub-table we build an equation. Each sub-table represents an equation matching the corresponding constructor. We build an equation matching that constructor only (i.e. with variables in all other positions), and passing on the constructor's arguments.

From Figure 6, we generate two equations for f:

```
f xs (y : ys) = g1 xs y ys
f xs [ ]      = g2 xs
```

Step 5: If all of the rows of a sub-table return the same value, then no further matching is required, and no recursion is necessary. This constitutes the base case. For example, every row in the table for g2 returns the value 3, yielding the definition

```
g2 _ = 3
```

If a sub-table doesn't satisfy the base case, we recurse and build a new function from the sub-table by the same process. Doing this for g1 eventually yields the definition

```
g1 [ ]      = 1
g1 (_ : _)  = 2
```

5.3 Minor Optimisations

Several minor optimisations are available which make marginal improvements to the performance of the transformed definitions.

- Wherever a definition has no matching, it can be in-lined into its caller. Also, wherever a definition has only one equation whose matching is nested inside the matching in its caller, it can be in-lined (Section 6 shows an example).
- Wherever a definition takes an argument that neither it nor its callees matches, that argument can be dropped.
- If at any stage a pattern-table shows that a definition is strict in more than one argument or component, the derived definition can match all of those arguments, as the order in which they are matched is irrelevant: left-to-right is as good as any other.

5.4 Bindings

We have not yet discussed the question of binding arguments: we have derived only definitions that have trivial right-hand sides. Incorporating the requirement to bind arguments and components of arguments induces some minor syntactic contortions that we omit here: they serve only to obscure the discussion.

By way of an example, in the general case the final transformed version of the running example f from Section 5 would be as follows.

```
f xs ys = ff (f' xs ys) xs ys

f' xs (y : ys) = f'' xs
f' xs [ ]      = 3

f'' [ ] = 1
f'' xs  = 2

ff 1 [ ] (y : ys) = 47
ff 2 xs  (y : ys) = 48
ff 3 xs  [ ]      = 49
```

Clearly, where a function's equations have simple bodies (like f), there are many opportunities to simplify the final definition to reduce the overheads of function invocation and argument passing.

6 A Worked Example

To illustrate the transformation steps in action we consider a function involving nested constructor applications:

```
g [ ]       [True]  = 37
g (x : xs)  [False] = 38
g xs        [y]     = 39
```

The pattern-tables for the three separate equations of g are shown in Figure 7. Where possible without affecting the result of the process, we have combined some rows in these tables, to reduce the number of cases. The combined pattern-tables generated for g in the second phase of the scheme are shown in Figure 8.

The final table for g tells us that

- the first argument is not always required (for example g ⊥ [ ] = 0), but
- the second argument *is* always required (g xs ⊥ = –1, ∀xs :: [a]).

Moreover, g xs [ ] = 0, ∀xs :: [a], so we derive just one equation:

```
g xs (y : ys) = g' xs y ys
```

where g' must satisfy the pattern-table shown in Figure 9. It is the same as the final table for g, except:

- The first two rows of entries are gone, as they don't match :, and
- The two arguments to : in each entry appear as separate arguments.

The table for g' tells us that

- the first argument is not always required (for example g' ⊥ [ ] [True] = 0), and
- the second argument is not always required (for example g' ⊥ ⊥ [True] = 0), but
- the third argument *is* always required (g' xs b ⊥ = –1, ∀xs :: [a], b :: Bool).

Moreover, g' xs b (y : ys) = 0, ∀xs :: [a], b, y :: Bool, ys :: [Bool], so again we derive just one equation:

```
g' xs b [ ] = g'' xs b
```

where g'' must satisfy the pattern-table shown in Figure 10. This table is the same as the first three rows of entries in the table for g', with the third argument in each entry (i.e. the [ ]) discarded.

The table for g'' tells us that

- the first argument is always required (g'' ⊥ b = –1, ∀b :: Bool), and
- the second argument is always required (g'' xs ⊥ = –1, ∀xs :: [a]).

In a situation like this, we can match either argument next, and the simplest approach syntactically is to match both:

```
g'' [ ]     True  = 1
g'' [ ]     False = 3
g'' (_ : _) True  = 3
g'' (_ : _) False = 2
```

Each of the equations of g'' matches only one row of the table, so this completes the process. In-lining and simplifying syntactically, we derive the sequential definition:

```
g xs [b] = g'' xs b

g'' [ ]     True  = 1
g'' (_ : _) False = 2
g'' _       _     = 3
```

Note again that this definition returns equation numbers.

7 Discussion

7.1 Reducing Concurrency

The examples f and g can be implemented in a single thread. However, the same technique can also be used to minimise the concurrency in definitions that cannot be sequentialised completely. Consider the function h.

```
h True  True True  = 13
h False True True  = 14
h x     y    z     = 15
```

This is the pattern-table for h, again somewhat abbreviated.

```
h True  True  True  = 1        h _ True  False = 3
h False True  True  = 2        h _ False True  = 3
h ⊥     True  True  = –1       h _ ⊥     False = 3
h _     True  ⊥     = –1       h _ False ⊥     = 3
h _     ⊥     True  = –1       h _ False False = 3
h _     ⊥     ⊥     = –1
```

```
g [ ] [ ]          = 0
g [ ] ⊥            = −1
g [ ] [True]       = 1
g [ ] [False]      = 0
g [ ] [⊥]          = −1
g [ ] (True : ⊥)   = −1
g [ ] (False : ⊥)  = 0
g [ ] (⊥ : ⊥)      = −1
g [ ] (_ : _ : _)  = 0
```

```
g (_ : _) _        = 0
```

```
g ⊥ [ ]            = 0
g ⊥ ⊥              = −1
g ⊥ [True]         = −1
g ⊥ [False]        = 0
g ⊥ [⊥]            = −1
g ⊥ (True : ⊥)     = −1
g ⊥ (False : ⊥)    = 0
g ⊥ (⊥ : ⊥)        = −1
g ⊥ (_ : _ : _)    = 0
```

Figure 7(a): The table for Equation 1 of g.

```
g [ ] _ = 0
```

```
g (_ : _) [ ]          = 0
g (_ : _) ⊥            = −1
g (_ : _) [True]       = 0
g (_ : _) [False]      = 2
g (_ : _) [⊥]          = −1
g (_ : _) (True : ⊥)   = 0
g (_ : _) (False : ⊥)  = −1
g (_ : _) (⊥ : ⊥)      = −1
g (_ : _) (_ : _ : _)  = 0
```

```
g ⊥ [ ]            = 0
g ⊥ ⊥              = −1
g ⊥ [True]         = 0
g ⊥ [False]        = −1
g ⊥ [⊥]            = −1
g ⊥ (True : ⊥)     = 0
g ⊥ (False : ⊥)    = −1
g ⊥ (⊥ : ⊥)        = −1
g ⊥ (_ : _ : _)    = 0
```

Figure 7(b): The table for Equation 2 of g.

```
g _ [ ]            = 0
g _ ⊥              = −1
g _ [_]            = 3
g _ (_ : ⊥)        = −1
g _ (_ : _ : _)    = 0
```

Figure 7(c): The table for Equation 3 of g.

Figure 7: The pattern-tables for the equations of g, treated separately.

```
g [ ] [ ]          = 0
g [ ] ⊥            = −1
g [ ] [_]          = 3
g [ ] (_ : ⊥)      = −1
g [ ] (_ : _ : _)  = 0

g (_ : _) [ ]          = 0
g (_ : _) ⊥            = −1
g (_ : _) [True]       = 3
g (_ : _) [False]      = 2
g (_ : _) [⊥]          = −1
g (_ : _) (True : ⊥)   = −1
g (_ : _) (False : ⊥)  = −1
g (_ : _) (⊥ : ⊥)      = −1
g (_ : _) (_ : _ : _)  = 0

g ⊥ [ ]          = 0
g ⊥ ⊥            = −1
g ⊥ [True]       = 3
g ⊥ [False]      = −1
g ⊥ [⊥]          = −1
g ⊥ (True : ⊥)   = −1
g ⊥ (False : ⊥)  = −1
g ⊥ (⊥ : ⊥)      = −1
g ⊥ (_ : _ : _)  = 0
```

Figure 8(a): The combined table for Equations 2 and 3 of g.

```
g [ ] [ ]          = 0
g [ ] ⊥            = −1
g [ ] [True]       = 1
g [ ] [False]      = 3
g [ ] [⊥]          = −1
g [ ] (True : ⊥)   = −1
g [ ] (False : ⊥)  = −1
g [ ] (⊥ : ⊥)      = −1
g [ ] (_ : _ : _)  = 0

g (_ : _) [ ]          = 0
g (_ : _) ⊥            = −1
g (_ : _) [True]       = 3
g (_ : _) [False]      = 2
g (_ : _) [⊥]          = −1
g (_ : _) (True : ⊥)   = −1
g (_ : _) (False : ⊥)  = −1
g (_ : _) (⊥ : ⊥)      = −1
g (_ : _) (_ : _ : _)  = 0

g ⊥ [ ]          = 0
g ⊥ ⊥            = −1
g ⊥ [True]       = −1
g ⊥ [False]      = −1
g ⊥ [⊥]          = −1
g ⊥ (True : ⊥)   = −1
g ⊥ (False : ⊥)  = −1
g ⊥ (⊥ : ⊥)      = −1
g ⊥ (_ : _ : _)  = 0
```

Figure 8(b): The combined table for all equations of g.

Figure 8: The combined pattern-tables for the equations of g.

```
g' [ ] True  [ ]      = 1
g' [ ] False [ ]      = 3
g' [ ] ⊥     [ ]      = −1
g' [ ] True  ⊥        = −1
g' [ ] False ⊥        = −1
g' [ ] ⊥     ⊥        = −1
g' [ ] _     (_ : _)  = 0

g' (_ : _) True  [ ]      = 3
g' (_ : _) False [ ]      = 2
g' (_ : _) ⊥     [ ]      = −1
g' (_ : _) True  ⊥        = −1
g' (_ : _) False ⊥        = −1
g' (_ : _) ⊥     ⊥        = −1
g' (_ : _) _     (_ : _)  = 0

g' ⊥ True  [ ]      = −1
g' ⊥ False [ ]      = −1
g' ⊥ ⊥     [ ]      = −1
g' ⊥ True  ⊥        = −1
g' ⊥ False ⊥        = −1
g' ⊥ ⊥     ⊥        = −1
g' ⊥ _     (_ : _)  = 0
```

Figure 9: The required pattern-table for g'.

```
g'' [ ] True  = 1
g'' [ ] False = 3
g'' [ ] ⊥     = −1

g'' (_ : _) True  = 3
g'' (_ : _) False = 2
g'' (_ : _) ⊥     = −1

g'' ⊥ True  = −1
g'' ⊥ False = −1
g'' ⊥ ⊥     = −1
```

Figure 10: The required pattern-table for g''.

None of the arguments of h is always required, so we cannot derive an equivalent sequential definition. However, we observe that h x ⊥ ⊥ = –1, ∀x :: Bool, so we can generate the equivalent definition

```
h x True True = h' x
h x y    z    = 15

h' True  = 13
h' False = 14
```

This definition requires fewer threads than the original. and depending on the actual arguments passed to h, we might expect to see a performance benefit in some cases. In particular, if the second or third argument is False, the new definition will perform no evaluation at all on the first argument.

7.2 Optimality

Although we have no formal proof, we conjecture that the pattern-table analysis described will generate a definition that will execute using the minimum possible number of threads. More specifically, we believe that it will generate a sequential definition whenever it is possible to do so. Proving this is left as future work.

7.3 Relation to Strictness Analysis

This scheme is clearly related to strictness analysis: we are trying to determine which arguments are always needed to perform the pattern matching for a function. Indeed, one way to improve the sequentialisation is to incorporate strictness information derived from the bodies of the equations. Consider the function nearlyor.

```
nearlyor False False = False
nearlyor x     y     = not y
```

There is no way to sequentialise nearlyor when looking only at its patterns. However, the second equation of nearlyor clearly requires its rightmost argument in order to return a result, so if we combine this information into the pattern-table for nearlyor, we can derive an equivalent definition nearlyor' which is sequential.

```
nearlyor' x False = x
nearlyor' x True  = False
```

So why use pattern-tables instead of strictness analysis? Partly because the pattern-table technique is much simpler, but also because

- it allows us to reduce concurrency even when a function is not strict in any single argument (for example see the function h in Section 7.1); and
- it deals naturally with nested constructor applications.

8 Summary, Conclusions and Future Work

Parallel pattern matching offers the "maximum laziness" available for a function definition: all arguments are evaluated concurrently and all arguments are given equal precedence. Thus, pattern matching failure within individual equations is promoted and the definition can return a result whenever possible. This is important for the intuitive reading and predictable behaviour of functional programs in the presence of errors and of infinite computations, particularly when non-trivial program transformations are applied.

However, in general, definitions using PPM require concurrency for their implementation. Consequently, programs that make extensive use of parallel matching suffer a significant performance hit. Our "source-level" implementation of parallel matching via Concurrent Haskell (Section 3) suggests that this performance hit can be between one and two orders of magnitude in extreme cases.

We have described a scheme that can improve performance in some cases. A definition written using PPM semantics can sometimes be transformed at the source-level into a definition that is equivalent (it always returns the same result as the original), but that has an obvious sequential implementation, typically because it matches only one argument at a time. This reduces the need for concurrency in the implementation of PPM, and should significantly reduce the performance penalty for many programs. Moreover, even where a definition cannot be fully sequentialised, our scheme will sometimes be able to reduce the number of concurrent processes required to observe the PPM semantics.

Note that the transformation scheme will not directly improve the performance of the examples from Section 4 (they were chosen this way), but it will still improve the overall performance of programs under PPM.

These optimisations are independent of any particular implementation of PPM, and they are likely to play an important role in any future implementation of PPM.

8.1 Future Work

We need to construct proofs of correctness and of optimality for the transformation scheme. Some aspects of the translation and subsequent transformation are easy to verify for correctness. Other aspects, for example pattern-table analysis (Section 5.2), present more significant challenges, but would appear to be viable. A much more difficult task is establishing optimality, in terms of the number of threads that the implementation uses/needs to execute a given program. It is not at all clear how to cast this formally, let alone how to prove that our transformation scheme is optimal in this sense.

The value of the transformation scheme depends partly on the proportion of real functions that can be sequentialised. It will be interesting to apply our scheme to a range of Haskell benchmarks/libraries etc. to

get a feel for the performance hit that might be experienced in practice by a naive use of ppm and the extent to which program transformation can recover the overheads.

An obvious way to improve the performance of our concurrency framework is to support PPM in a production compiler such as GHC, and its associated run-time system. Working at the source level, as we have done here, presents some interesting insights into parallel pattern matching performance, but it is not clear yet how much better we could do given low-level control over any threads/processes required.

Finally, it will be interesting to explore other variations on sequential matching, for example across multiple equations, rather then within individual equations. Here, we have stuck with Haskell's top-down approach to disambiguating overlapping patterns. There are other options, such as best-fit matching[Field *et al*, 1992], which could, in principle, combine pattern matching in the "vertical" with pattern matching in the "horizontal".

Acknowledgements

We should like to thank Nick Jardine for his contributions to earlier versions of this work, and Andy Cheadle for providing technical assistance in the use of GHC.

References

[Bird and Wadler, 1988] Bird, R., and Wadler, P.L. (1988). *Introduction to Functional Programming*. Prentice Hall International Series in Computer Science, Hemel Hempstead, UK.

[Courtenage and Poulovassilis, 1995] Courtenage, S. and Poulovassilis, A. (1995). *Combining Inheritance and Parametric Polymorphism in a Functional Database Language*. Proc. 13th British National Conference on Databases, Manchester 1995, LNCS 940.

[Dijkstra, 1959] Dijkstra, E.W. (1959). *A Note on Two Problems in Connection with Graphs*. Numerische Math, 1: 269–271

[Field *et al*, 1992] Field, A.J., Hunt, L.S., and While, R.L. (1992). *The Semantics and Implementation of Various "Best-fit" Pattern Matching Schemes for Functional Languages*. Departmental Report DoC 92/13, Dept. of Computing, Imperial College.

[Kennaway, 1990] Kennaway, J.R. (1990). *The Specificity Rule for Lazy Pattern Matching in Ambiguous Term Rewrite Systems*. ESOP 1990: 256–270.

[Longley, 1999] Longley, J. (1999). *When is a Functional Program not a Functional Program?* 1999 International Conference on Functional Programming, Paris, France.

[Marlow et al, 2001] Marlow, S., Peyton Jones, S.L., Seward, J., and Thomas, R. (2001). *The Glasgow Haskell Compiler*. The latest information is available at *http://www.haskell.org/ghc*.

[Peyton Jones *et al*, 1996] Peyton Jones, S.L., Gordon, A., and Finne, S. (1996). *Concurrent Haskell*. 23^{rd} ACM Symposium on Principles of Programming Languages, Florida.

[Peyton Jones *et al*, 1999] Peyton Jones, S.L., and others (1999). *Haskell 98: a Non-strict, Purely Functional Language*. The latest version is available at *http://www.haskell.org/definition*.

[Plotkin, 1977] Plotkin, G.D. (1977). *LCF Considered as a Programming Language*, Theoretical Computer Science, 5:223–55.

[While, 1994] While, R.L. (1994). *Parallel Pattern Matching in Lazy Functional Languages*. 2^{nd} Massey Functional Programming Workshop, Massey University, New Zealand.

[While and Field, 2005] While, R.L. and Field, T. (2005). *Optimising Parallel Pattern Matching by Source-level Program Transformation*. Australian Computer Science Communications, 27(1): 239–248.

[While and Mildenhall, 2002] While, R.L. and Mildenhall, G. (2002). *An Implementation of Parallel Pattern Matching via Concurrent Haskell*. Australian Computer Science Communications, 24(1): 293–302.