

Optimising Parallel Pattern-matching by Source-level Program Transformation

Lyndon While

School of Computer Science & Software Eng,
The University of Western Australia,
Western Australia 6009
email: lyndon@csse.uwa.edu.au

Tony Field

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

Abstract

Parallel pattern-matching (PPM) provides true commutative implementation of functions defined by cases in functional languages, because no argument is given precedence over any other. However, the requirement for concurrency (in general) to support these semantics means that current implementations incur a significant performance penalty over simple, traditional left-to-right semantics. We describe a source-level program transformation scheme that analyses a PPM definition and is often able to generate an equivalent definition that can be executed without concurrency. Where sequential implementation is not possible, the scheme is sometimes able to generate an equivalent definition that reduces the number of concurrent threads required to execute a definition. This transformation scheme promises to deliver a major improvement in the performance of PPM implementations.

Keywords: functional languages, pattern-matching, semantics, concurrency, program transformation.

1 Introduction

A key feature of most modern functional languages such as Haskell[7] 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[4, 3, 1].

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.

$$\begin{array}{l} \text{False} \parallel \text{False} = \text{False} \\ x \parallel y = \text{True} \end{array}$$

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, $\text{True} \parallel \perp = \text{True}$ but $\perp \parallel \text{True} = \perp$, so `||` is *not* commutative. This asymmetry can sometimes affect execution time as well as termination: e.g. if both arguments evaluate to `True` and the evaluation of the first argument involves significantly more work than that of the second, then left-to-right matching will be slower than right-to-left.

Parallel pattern-matching (PPM), e.g. as discussed in [8, 3, 9, 5, 10], 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 $\text{True} \parallel \perp = \perp \parallel \text{True} = \text{True}$. 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.

While and Mildenhall[10] describe an implementation scheme for PPM by a source-level transformation into Concurrent Haskell[6]. When a pattern matches more than one of its arguments, a new process is created to evaluate and match each argument. If all of the individual matches succeed, then the overall match succeeds. But if any of the individual matches fails, all of the unfinished processes are terminated and the overall match fails. The PPM semantics are 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.

However, because of the requirement for concurrency to implement PPM, programs are likely to incur a significant performance penalty. The first (minor) contribution of this paper is a study of the performance of the implementation scheme reported in [10]. Over a range of functions and arguments where both schemes terminate, we observe a slowdown of up to 5–15 times compared to an implementation using simple left-to-right semantics. Such a slowdown would likely make PPM an unattractive option for most applications.

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. The principal contribution of this paper is a scheme that analyses a PPM definition, and transforms it at the source-level into an equivalent sequential definition wherever possible. 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; and it can optimise some definitions that cannot be fully sequentialised, as described above.

Note that the optimisation scheme described in this paper 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 remainder of the paper is organised as follows. Section 2 describes the operational behaviour of PPM definitions under the scheme described in [10]. Section 3 studies the performance of definitions under this scheme, and introduces the idea of sequentialising functions by transformation. Section 4 describes how a simple example definition is analysed and transformed, Section 5 describes the transformation scheme informally, and Section 6 describes the analysis and transformation of a bigger example. Section 7 discusses some further issues with the transformation scheme, including the notion of “partly sequentialising” definitions, and the relationship to strictness analysis. Section 8 concludes the paper and outlines some ideas for future work.

2 The Operational Behaviour of PPM

The implementation of PPM described in [10] is based upon a source-to-source program transformation. The input is a program written using Haskell syntax, but with PPM semantics, and the output is a Concurrent Haskell program. The general idea is to replace the patterns in the original functions with guards in Concurrent Haskell. Each guard expresses the matching explicitly in terms of a function `ppm`, which encodes the pairwise matching operator \otimes from Figure 1.

<code>match</code>	\otimes	<code>match</code>	$=$	<code>match</code>
<code>⊥</code>	\otimes	<code>match</code>	$=$	<code>⊥</code>
<code>match</code>	\otimes	<code>⊥</code>	$=$	<code>⊥</code>
<code>⊥</code>	\otimes	<code>⊥</code>	$=$	<code>⊥</code>
<code>fail</code>	\otimes	<code>match</code>	$=$	<code>fail</code>
<code>match</code>	\otimes	<code>fail</code>	$=$	<code>fail</code>
<code>fail</code>	\otimes	<code>⊥</code>	$=$	<code>fail</code>
<code>⊥</code>	\otimes	<code>fail</code>	$=$	<code>fail</code>
<code>fail</code>	\otimes	<code>fail</code>	$=$	<code>fail</code>

Figure 1: The pairwise matching operator for PPM. Failure to match either argument causes a failure overall. [10] gives a formal semantics for top-down pattern-matching.

To illustrate this, we revisit the `||` example from Section 1. The definition generated for `||` is as follows.

```
x || y | ppm [not x, not y] = False
      | otherwise           = True
```

The pattern-matching in the original definition is replaced by the guard in the first alternative. In keeping with Figure 1, `ppm` should return `True` iff all of the expressions on its argument list are `True`, and it should return `False` if any expression on the list is `False`, even if the others cause an error or loop indefinitely. Full details of the translation scheme are given in [10].

It is instructive to discuss the operation of `ppm` in the context of the `||` example above. Given `[b1, b2] :: [Bool]`, `ppm` performs the following actions. Figure 2 shows the process structure generated for this example.

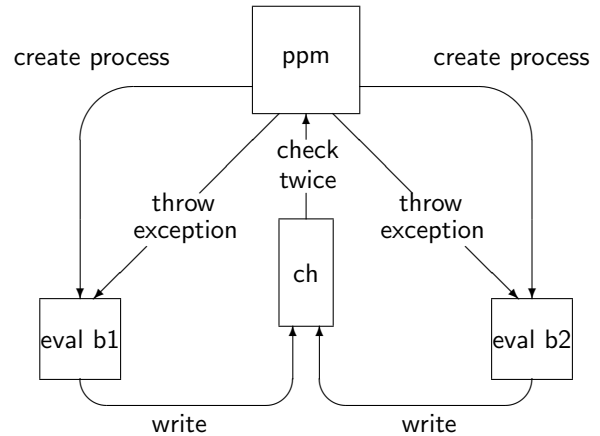


Figure 2: The process and communication structure created for `ppm [b1, b2]`. `ch` is the channel created by `ppm`.

1. `ppm` creates a new channel `ch`.
2. It creates a new process for each of `b1` and `b2`. Each process forces the evaluation of its argument, then writes the result to `ch`. These processes execute independently: in particular, they can write to `ch` in any order.
3. It waits for each of the processes to complete:
 - if either process writes `False` to `ch`, `ppm` kills the other process by throwing an exception to it, then it returns `False`; *otherwise*
 - if either process loops, then `ppm` will never return, because it can never be sure that there isn't a `False` coming; *otherwise*
 - if either process encounters an error, the system will eventually reach a state where `ppm` is waiting on `ch`, but no process exists to write to it. The system can recognise this deadlock and it returns an error, as required in the semantics; *otherwise*
 - if both processes write `True` to `ch`, `ppm` returns `True`.

Note that if the program has nested applications of `ppm`, a process may be interrupted by its parent whilst doing Step 3. In this case, the process tells its own children to kill themselves, and then terminates gracefully. This mechanism guarantees that all processes will be killed.

The final point is that when the program graph contains multiple invocations of `ppm`, each one behaves 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.

A complete definition of `ppm` and the PPM module is given in [10].

3 The Performance of PPM

The scheme described in [10] works exclusively at the source level. To assess its performance fairly, we need to separate the overhead of PPM from any additional overhead associated with working via Concurrent Haskell. To achieve this 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`, which implements left-to-right matching. All benchmarks were run on a dedicated 1GHz Athlon PC with 256MB RAM running Linux Red Hat 7.2 and using GHC 5.02.1. The timings were averages over 5 runs (typically).

We begin with a benchmark which maps `||` over a list of (pairs of) booleans.

```
porTest bs bs' = count (zipWith (||) bs bs')
```

`count` returns the number of `Trues` in its argument, thus forcing the evaluation of the entire list. `bs` and `bs'` are built off-line and so are factored out of the execution time. The execution times for left-to-right matching (T_{seq}) and PPM (T_{par}) are shown in Table 1. The results show that the overheads are

$n/10^3$	$T_{seq}(s)$	$T_{par}(s)$
1	0.19	2.54
2	0.60	7.93
3	1.21	17.23
4	2.05	29.24
5	3.11	44.12
6	4.33	62.08
7	5.85	83.20

Table 1: Execution times for `porTest` benchmark.

quite substantial. Approximately a third of the user-defined function calls involve parallel matching, and each thread created terminates almost immediately, since the two lists are pre-computed and contain exclusively ground elements.

The next benchmark is a function which effects a parallel search using a parallel ‘and’ function (`|&|`) analogous to `||`. This attempts to answer the question of whether PPM might achieve a speedup in some cases.

```
search s r1 r2 | member s r1 |&| member s r2 = "yes"
                | otherwise                = "no"
```

`search` takes a search key and two relations—lists of key/item pairs. It returns “yes” if the key is found in *both* relations and “no” otherwise. Under left-to-right matching, if the key occurs in neither relation, the program executes faster if $|r1| < |r2|$. Using PPM, the order is unimportant: as soon as one match fails, the other is terminated. If the relations differ significantly in terms of size, it will be cheaper overall to use PPM.

The program was executed under PPM for $|r1| = 10^7$ with varying sizes of `r2` when the search key was present in neither `r1` nor `r2`. Table 2 shows the results. Under left-to-right matching the execution time is the same in all cases: 16.50s. Table 2 shows that PPM improves performance when $|r2| < 3.5 \times 10^6$, when the total number of entries searched is about 7×10^6 . Beyond this, PPM performs substantially worse, as expected: PPM always searches part of both relations.

The next benchmark is a function for summing edge weights in an implementation of Dijkstra’s all paths shortest path algorithm[2]. This function is called many times in the relaxation phase of the algorithm. The results are summarised in Table 3. For

$n/10^6$	$T_{par}(s)$
1	4.83
2	9.50
3	14.83
4	19.18
5	28.88

Table 2: Execution times for search benchmark.

n	$T_{seq}(s)$	$T_{par}(s)$
50	1.07	3.71
60	1.61	6.15
70	2.34	10.04
80	3.10	13.82
90	4.17	16.44
100	5.33	27.71

Table 3: Execution times for Dijkstra benchmark.

a problem of size n , the graph contains $6n$ nodes and $12n$ edges. Once again we see a significant performance hit in moving to PPM.

As a final exercise, we consider the effect of changing certain functions in the Standard Prelude from sequential to parallel semantics. The `&&` and `||` functions are good examples. We chose an application which made extensive use of `&&`: a program for building a quad-tree from a list of particles. One core function takes a particle (specified by its location and mass) and the bounding box of the current node in the quad-tree and returns the quadrant in which the particle belongs.

```
quad ((x, y), m) (x0, y0) (x1, y1)
  | x <= xh && y >  yh = ULeft
  | x >  xh && y >  yh = URight
  | x <= xh && y <= yh = LLeft
  | x >  xh && y <= yh = LRight
  where xh = (x0 + x1)/2
        yh = (y0 + y1)/2
```

The results for varying problem sizes (the main program builds a 30-node tree n times and sums the total particle mass over all (identical) trees) are shown in Table 4. The pattern is the same as for the previous benchmarks.

$n/10^3$	$T_{seq}(s)$	$T_{par}(s)$
1	0.79	10.25
2	1.65	21.24
3	2.62	33.50
4	3.65	46.32
5	4.81	60.30
6	5.92	74.81
7	7.29	90.77
8	8.56	107.03

Table 4: Execution times for `quadTree` benchmark.

3.1 Improving Performance

The figures show that PPM causes a significant performance penalty for some Haskell programs. This penalty might be compared to the penalty incurred by the adoption of normal-order semantics. However, in the case of PPM, the penalty 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
```

The first equation of f matches two arguments and appears to require concurrency for its implementation under PPM. However, 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. We expect that transforming definitions in this way will considerably reduce the overall performance penalty associated with PPM.

4 The Optimisation of PPM

Our scheme optimises the performance of PPM definitions by reducing the level of concurrency required to implement the semantics. In particular, given a definition that appears to require concurrency, the scheme tries to construct an equivalent definition that can be evaluated in a single thread.

We achieve this with 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 \perp 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 $x = \perp$, the result of the function application will always be \perp . Specifically, x is always required if, in the pattern-table, every form with $x = \perp$ 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 a 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.

4.1 Creating pattern-tables

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 from Section 3.

The table for the first equation of f is as follows.

```
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 \perp), so the table has nine entries. Note that all of the entries are disjoint. The number of 0s in the table indicates the promotion of failure in the semantics.

The 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 simple.

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

The tables for the individual equations are combined in a process that mirrors the top-down semantics of the pattern-matching process. Given a sequence of n tables T_i , $1 \leq i \leq n$, the table for the entire definition is given by `foldr1 topdown [T1, ..., Tn]`, where `topdown` combines tables pairwise. `foldr1` “inserts” applications of its first argument between the elements on its second argument, starting from the right. For $n = 3$, this is equivalent to `topdown T1 (topdown T2 T3)`. The application `topdown Tj Tk` inspects each entry $e = v$ in T_j , and:

- if $v \neq 0$ (i.e. success or \perp : 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 $e' = v'$ in T_k where e' overlaps e , and it includes `combine e e' = v'` in the result.

e' overlaps e if there are any values matched by both e' and e . `combine e e'` returns a pattern that contains the union of all of the matching done by either e or e' : if an argument is matched in e or in e' , it is matched in `combine e e'`.

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

```
f [] [] = 3      f [] (- : -) = 1      f [] ⊥ = -1
f (- : -) [] = 3  f (- : -) (- : -) = 2      f (- : -) ⊥ = -1
f ⊥ [] = 3       f ⊥ (- : -) = -1      f ⊥ ⊥ = -1
```

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

4.2 Interpreting pattern-tables

Now we examine the final table to determine if any of the arguments is *always* evaluated in the pattern-matching process.

The first argument is not always needed because $f \perp [] = 3$. 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.

The second argument *is* always needed because $f xs \perp = -1, \forall xs :: [a]$. It is impossible to reject both of the last two equations without examining the second argument.

We can therefore generate a version of f that matches its second argument.

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

This will be equivalent to the original definition iff $g1$ and $g2$ satisfy the following pattern-tables (extracted from the table for f).

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

In general, we should now apply the transformation process recursively to the pattern-tables for `g1` and `g2`. However, for the purposes of this example, we can see easily that `g1` needs to match its first argument, whereas because `g2 xs = 3, ∀xs :: [a]`, `g2` does not need to match anything. Their definitions are thus

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

```
g2 _ = 3
```

`g2` here represents the true base case of the process: when all rows of a table return the same (always positive) value, no further matching is required.

Finally, inlining the definition of `g2`, and eliminating the unused arguments from the definition of `g1`, we derive the version of `f` given in Section 3.1.

```
f xs (y : ys) = g1 xs
f xs []       = 3
```

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

Note that this version of `f` returns an equation number, not an actual result: we discuss later the question of binding arguments for right-hand sides.

5 An Informal Description of the Transformation Scheme

We now give an informal description of our transformation scheme. There are two major phases.

5.1 Building Pattern-tables

Pattern-tables are built in two phases.

5.1.1 Build a pattern-table for each equation

The pattern-table for a single argument is constructed as follows.

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).

A constructor application whose type has n constructors generates a table with $n + m$ rows: 1 row matching \perp , returning an error; $n - 1$ failing rows for the other constructors in the type; and m rows involving the correct constructor and the arguments of the application. The value of m will depend on the complexity of the argument patterns. Note that nested constructor applications are handled naturally, e.g.:

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

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

One pattern with \perp , one pattern with the wrong constructor, and one pattern with the right constructor.

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

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

One pattern with \perp , 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 `:`.

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

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

One pattern with \perp , 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 `:`.

A tuple with n fields generates a table with $\prod_{i=1}^n r(i)$ rows, where $r(i)$ is the number of rows in the table generated by the i th field. This is the cross-product of all of the rows generated by each of its fields. Each combination returns the “lowest” value returned by its components: i.e. any failures will cause failure, otherwise any errors will cause an error, otherwise the combination will return success

Generating the pattern-table for a list of arguments is essentially the same process as for a tuple.

5.1.2 Combine the individual pattern-tables

The pattern-tables are combined pairwise, mirroring the rules of top-down matching. For each row $r1$ in the first table: if $r1$'s return value is *not* failure, $r1$ is added to the result, otherwise find each row $r2$ from the second table whose pattern “overlaps” with $r1$, and add to the result a new row that does all of the matching from both $r1$ and $r2$, and that returns the same value as $r2$. Two rows overlap iff there exists an argument that they both match.

5.2 Analysing Pattern-tables

Pattern-tables are analysed recursively in several phases.

5.2.1 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.

5.2.2 Find out which positions (if any) are strict

Examine the pattern-table for each position: a position is strict iff wherever that position is \perp in the table, the return-value is -1 .

5.2.3 For the first strict position: 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. Clearly such sub-tables can be discarded.

5.2.4 For each remaining sub-table: build an equation and recurse if necessary

Each sub-table represents an equation matching the corresponding constructor. Build an equation matching that constructor only (i.e. with variables in all other positions), then if the sub-table doesn't satisfy the base case, build a new function by the same process.

5.2.5 The base case is when a pattern-table has all return-values equal

If all of the rows of a sub-table return the same value, then no further matching is required, and no recursion is necessary.

5.3 Other Issues

5.3.1 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 inlined into its caller. Also, wherever a definition has only one equation whose matching is nested inside the matching in its caller, it can be inlined (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.

5.3.2 Bindings

The astute reader will have noticed that we have not 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 f from Section 4 would be as follows.

$$f \text{ xs ys} = ff (f' \text{ xs ys}) \text{ xs ys}$$

$$\begin{aligned} f' \text{ xs } (y : \text{ys}) &= f'' \text{ xs} \\ f' \text{ xs } [] &= 3 \end{aligned}$$

$$\begin{aligned} f'' [] &= 1 \\ f'' \text{ xs} &= 2 \end{aligned}$$

$$\begin{aligned} ff 1 [] (y : \text{ys}) &= 47 \\ ff 2 \text{ xs } (y : \text{ys}) &= 48 \\ ff 3 \text{ xs } [] &= 49 \end{aligned}$$

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 Bigger Example

As a bigger example of a transformation, in particular one involving nested constructor applications, consider the function g .

$$\begin{aligned} g [] & \quad [True] = 37 \\ g (x : \text{xs}) & [False] = 38 \\ g \text{ xs} & \quad [y] = 39 \end{aligned}$$

The pattern-tables for the three separate equations of g are shown in Figure 3. (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 4.

The final table for g tells us that

- the first argument is not always required (e.g. $g \perp [] = 0$), but
- the second argument *is* always required ($g \text{ xs } \perp = -1, \forall \text{xs} :: [a]$).

Moreover, $g \text{ xs } [] = 0, \forall \text{xs} :: [a]$, so we derive just one equation:

$$g \text{ xs } (y : \text{ys}) = g' \text{ xs } y \text{ ys}$$

where g' must satisfy the pattern-table shown in Figure 5. This table is the same as the final table for g , except that:

- 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 (e.g. $g' \perp [] [True] = 0$), and
- the second argument is not always required (e.g. $g' \perp \perp [True] = 0$), but
- the third argument *is* always required ($g' \text{ xs } b \perp = -1, \forall \text{xs} :: [a], b :: Bool$).

Moreover, $g' \text{ xs } b (y : \text{ys}) = 0, \forall \text{xs} :: [a], b, y :: Bool, \text{ys} :: [Bool]$, so again we derive just one equation:

$$g' \text{ xs } b [] = g'' \text{ xs } b$$

where g'' must satisfy the pattern-table shown in Figure 6. 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 ($g'' \perp b = -1, \forall b :: Bool$), and
- the second argument is always required ($g'' \text{ xs } \perp = -1, \forall \text{xs} :: [a]$).

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

$g [] []$	$= 0$	$g (- : -) -$	$= 0$	$g \perp []$	$= 0$
$g [] \perp$	$= -1$			$g \perp \perp$	$= -1$
$g [] [True]$	$= 1$			$g \perp [True]$	$= -1$
$g [] [False]$	$= 0$			$g \perp [False]$	$= 0$
$g [] [\perp]$	$= -1$			$g \perp [\perp]$	$= -1$
$g [] (True : \perp)$	$= -1$			$g \perp (True : \perp)$	$= -1$
$g [] (False : \perp)$	$= 0$			$g \perp (False : \perp)$	$= 0$
$g [] (\perp : \perp)$	$= -1$			$g \perp (\perp : \perp)$	$= -1$
$g [] (- : - : -)$	$= 0$			$g \perp (- : - : -)$	$= 0$

Figure 3(a): The table for Equation 1 of g .

$g [] - = 0$		$g (- : -) []$	$= 0$	$g \perp []$	$= 0$
		$g (- : -) \perp$	$= -1$	$g \perp \perp$	$= -1$
		$g (- : -) [True]$	$= 0$	$g \perp [True]$	$= 0$
		$g (- : -) [False]$	$= 2$	$g \perp [False]$	$= -1$
		$g (- : -) [\perp]$	$= -1$	$g \perp [\perp]$	$= -1$
		$g (- : -) (True : \perp)$	$= 0$	$g \perp (True : \perp)$	$= 0$
		$g (- : -) (False : \perp)$	$= -1$	$g \perp (False : \perp)$	$= -1$
		$g (- : -) (\perp : \perp)$	$= -1$	$g \perp (\perp : \perp)$	$= -1$
		$g (- : -) (- : - : -)$	$= 0$	$g \perp (- : - : -)$	$= 0$

Figure 3(b): The table for Equation 2 of g .

$g - []$	$= 0$
$g - \perp$	$= -1$
$g - [-]$	$= 3$
$g - (- : \perp)$	$= -1$
$g - (- : - : -)$	$= 0$

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

Figure 3: The pattern-tables for the equations of g , treated separately.

$g [] [] = 0$	$g (-: -) [] = 0$	$g \perp [] = 0$
$g [] \perp = -1$	$g (-: -) \perp = -1$	$g \perp \perp = -1$
$g [] [-] = 3$	$g (-: -) [True] = 3$	$g \perp [True] = 3$
	$g (-: -) [False] = 2$	$g \perp [False] = -1$
	$g (-: -) [\perp] = -1$	$g \perp [\perp] = -1$
$g [] (-: \perp) = -1$	$g (-: -) (True : \perp) = -1$	$g \perp (True : \perp) = -1$
	$g (-: -) (False : \perp) = -1$	$g \perp (False : \perp) = -1$
	$g (-: -) (\perp : \perp) = -1$	$g \perp (\perp : \perp) = -1$
$g [] (-: -: -) = 0$	$g (-: -) (-: -: -) = 0$	$g \perp (-: -: -) = 0$

Figure 4(a): The combined table for Equations 2 and 3 of g .

$g [] [] = 0$	$g (-: -) [] = 0$	$g \perp [] = 0$
$g [] \perp = -1$	$g (-: -) \perp = -1$	$g \perp \perp = -1$
$g [] [True] = 1$	$g (-: -) [True] = 3$	$g \perp [True] = -1$
$g [] [False] = 3$	$g (-: -) [False] = 2$	$g \perp [False] = -1$
$g [] [\perp] = -1$	$g (-: -) [\perp] = -1$	$g \perp [\perp] = -1$
$g [] (True : \perp) = -1$	$g (-: -) (True : \perp) = -1$	$g \perp (True : \perp) = -1$
$g [] (False : \perp) = -1$	$g (-: -) (False : \perp) = -1$	$g \perp (False : \perp) = -1$
$g [] (\perp : \perp) = -1$	$g (-: -) (\perp : \perp) = -1$	$g \perp (\perp : \perp) = -1$
$g [] (-: -: -) = 0$	$g (-: -) (-: -: -) = 0$	$g \perp (-: -: -) = 0$

Figure 4(b): The combined table for all equations of g .

Figure 4: The combined pattern-tables for the equations of g .

$g' [] [True] [] = 1$	$g' (-: -) [True] [] = 3$	$g' \perp [True] [] = -1$
$g' [] [False] [] = 3$	$g' (-: -) [False] [] = 2$	$g' \perp [False] [] = -1$
$g' [] \perp [] = -1$	$g' (-: -) \perp [] = -1$	$g' \perp \perp [] = -1$
$g' [] [True] \perp = -1$	$g' (-: -) [True] \perp = -1$	$g' \perp [True] \perp = -1$
$g' [] [False] \perp = -1$	$g' (-: -) [False] \perp = -1$	$g' \perp [False] \perp = -1$
$g' [] \perp \perp = -1$	$g' (-: -) \perp \perp = -1$	$g' \perp \perp \perp = -1$
$g' [] - (-: -) = 0$	$g' (-: -) - (-: -) = 0$	$g' \perp - (-: -) = 0$

Figure 5: The required pattern-table for g' .

$g'' [] [True] = 1$	$g'' (-: -) [True] = 3$	$g'' \perp [True] = -1$
$g'' [] [False] = 3$	$g'' (-: -) [False] = 2$	$g'' \perp [False] = -1$
$g'' [] \perp = -1$	$g'' (-: -) \perp = -1$	$g'' \perp \perp = -1$

Figure 6: The required pattern-table for g'' .

```

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. Inlining 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 Further Issues

7.1 Reducing (not eliminating) 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

```

None of the arguments of h is always required, so we cannot derive an equivalent sequential definition. However, we observe that $h\ x\ \perp\ \perp = -1, \forall x :: \text{Bool}$, 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 would expect sometimes to see a significant performance benefit. 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 (yet!), we believe that the pattern-table analysis described guarantees to generate a definition that will execute using the minimum possible number of threads. In particular, we believe that it will generate a sequential definition wherever possible.

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 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 $\text{nearlyor}'$ 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 (e.g. see the function h in Section 7.1); and
- it deals naturally with nested constructor applications.

8 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 (of 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: as a consequence, they can run significantly slower than under simple left-to-right semantics. For the implementation via Concurrent Haskell described in [10], we observe a slowdown of up to 5–15 times for some programs.

We have described a scheme that can reduce this slowdown 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 3 (they were chosen this way), but it will still improve the overall performance of programs under PPM.

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

Three avenues of future work are clear.

- We need to construct proofs of correctness and of optimality for the transformation scheme.
- The value of the transformation scheme depends partly on the proportion of real functions that can be sequentialised. This is non-trivial to estimate, because no-one has yet written code for the PPM semantics! However, we plan to quantify the performance gain roughly by taking a

suite of existing code and running it through a benchmarking process.

- [10] describes a number of ways in which their translation scheme via Concurrent Haskell is sub-optimal, relative to a more direct implementation “inside” the compiler. It remains a long-term goal of ours to take this more efficient (but more difficult!) route for implementing PPM.

Acknowledgements

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

References

- [1] 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.
- [2] DIJKSTRA, E.W. (1959). *A Note on Two Problems in Connection with Graphs*. *Numerische Math*, 1: 269–271
- [3] 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.
- [4] KENNAWAY, J.R. (1990). *The Specificity Rule for Lazy Pattern-Matching in Ambiguous Term Rewrite Systems*. ESOP 1990: 256–270.
- [5] LONGLEY, J. (1999). *When is a Functional Program not a Functional Program?* 1999 International Conference on Functional Programming, Paris, France.
- [6] PEYTON JONES, S.L., GORDON, A., AND FINNE, S. (1996). *Concurrent Haskell*. 23rd ACM Symposium on Principles of Programming Languages, Florida.
- [7] 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>.
- [8] PLOTKIN, G.D. (1977). *LCF Considered as a Programming Language*, *Theoretical Computer Science*, 5:223–55.
- [9] WHILE, R.L. (1994). *Parallel Pattern-matching in Lazy Functional Languages*. 2nd Massey Functional Programming Workshop, Massey University, New Zealand.
- [10] WHILE, R.L. AND MILDENHALL, G. (2002). *An Implementation of Parallel Pattern-matching via Concurrent Haskell*. *Australian Computer Science Communications*, 24(1): 293–302.