An Implementation of Parallel Pattern-matching via Concurrent Haskell

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

Abstract

Parallel pattern-matching offers the maximum laziness for programs written in lazy functional languages. Function arguments are evaluated concurrently and all arguments are given equal precedence, so functions can return results whenever possible in the presence of errors or infinite computations. We describe the motivation for and the semantics of parallel pattern-matching. We also describe the first available implementation of Haskell with parallel pattern-matching. The implementation works via a source-to-source translation into Concurrent Haskell, using the existing facilities of GHC to provide the required concurrency. Various transformation techniques are outlined which can help to reduce the degree and cost of the concurrency required to satisfy the semantics.

Keywords: functional programming, pattern-matching, concurrency, lazy semantics

1 Introduction

One of the principal interesting features of functional programming languages such as Haskell[Peyton Jones et al, 1999] is the use of *pattern-matching* to define functions. A definition written using pattern-matching has different equations for different cases of the input values. A simple example is the function length from the Haskell Prelude.

```
length :: [a] -> Int
-- length xs returns the number of elements on xs
length [ ]      = 0
length (x : xs) = 1 + length xs
```

length is defined by two equations. The first equation applies when length is applied to the empty list, denoted by [ ]. The second equation applies when length is applied to a non-empty list, binding the name x to the first element on the argument and the name xs to the list holding the rest of the elements on the argument. Repeated (recursive) application of the second equation, followed by one application of the first equation as the base case, returns the number of elements on any finite list.

Operationally, like most other functional languages, Haskell matches function definitions *top-down*. This means that when length is applied to a list, the system first tries to match the argument to the pattern in the first equation. If this matching process succeeds (i.e. if the argument is [ ]), 0 is returned as the result of the application, otherwise the system tries to match the argument to the pattern in the second equation. This process extends to definitions with more than two equations in the obvious way. If all of the patterns in a definition fail to match, the system returns an error.

Pattern-matching offers several advantages to the programmer.

- It allows a function definition to be broken into separate, independent parts, contributing to problem decomposition.
- It allows complex, nested conditionals to be written in a concise and easy-to-understand syntax.
- It allows the programmer to give names to the components of arguments, avoiding the need to use access functions like head and tail.
- It facilitates equation-based reasoning [Bird and Wadler, 1988].

However, in a language with lazy semantics[Bird and Wadler, 1988], pattern-matching sometimes combines badly with the intuitive reading of a definition. Consider the function ||, which implements logical disjunction.

```
(||) :: Bool -> Bool -> Bool
-- x || y returns False iff both x and y are False
False || False = False
x     || y     = True
```

Syntactically, this definition is symmetrical in the two arguments, and a naive reading would suggest that || is commutative, which accords with our intuition about disjunction. However, the sequential and lazy semantics of Haskell means that in reality || is *not* commutative. Given an application e || e', Haskell evaluates e first, and it evaluates e' only if e returns False. This is fine if e evaluates successfully, but if the evaluation of e causes an error or an infinite computation, then the application will fail without even considering e'. Thus the truth-table generated by the above definition is as shown in Table 1.

The problem here is with the application $\perp$ || True. The right argument is True, so we know that the first equation can't match: intuitively, the result should be True, from the second equation. However, because Haskell tests arguments from left-to-right, it evaluates the $\perp$ first, giving an error, and the actual result is $\perp$.

This confusion arises directly from the interaction between the use of pattern-matching and the lazy, sequential semantics of Haskell. We suggest an alternative semantics known as *parallel pattern-matching*:

> every argument (or component of an argument) is (conceptually) matched simultaneously, and a failure to match any argument (or component) means that an equation is rejected.

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

x	y	x || y
False	False	False
False	True	True
True	False	True
True	True	True
False	$\perp$	$\perp$
$\perp$	False	$\perp$
True	$\perp$	True
$\perp$	True	$\perp$
$\perp$	$\perp$	$\perp$

Table 1: The truth-table for || using standard Haskell. The value $\perp$ (pronounced "bottom") represents an error in evaluating an argument.

This semantics gives the "maximum laziness" for any function definition: all arguments are given equal precedence and the function can return a result whenever possible. For example, in the case $\perp$ || True from above, the two arguments are matched concurrently and the fact that the right argument fails to match the first equation "overrides" the fact that the left argument gives an error. The first equation is rejected, the result is True, and commutativity is restored.

Parallel pattern-matching has been discussed previously (e.g. [Plotkin, 1977], [While, 1994], [Longley, 1999]), but to our knowledge, no implementation has yet been constructed. We have designed and built a simple implementation that uses the process-based facilities provided in Concurrent Haskell[Peyton Jones et al, 1996] to evaluate and match function arguments "simultaneously". A separate process is created to evaluate and match each argument, and the set of processes is executed concurrently. There are then four possibilities.

- All of the processes match: evaluation proceeds with the body of the equation.
- One of the processes fails to match: all of the other processes are killed and the system tries the next equation.
- No process fails to match, and one of the processes loops indefinitely: the system loops indefinitely.
- No process fails to match or loops, and one of the processes returns an error: the system returns an error.

We have built a source-to-source translator that takes a function definition f and returns a new definition f', such that the semantics of f' in Concurrent Haskell is the same as the semantics of f using parallel pattern-matching. That is: f' runs in the Concurrent Haskell extension of GHC[Marlow et al, 2001] with the semantics that f would have under parallel pattern-matching.

The remainder of the paper is organised as follows. Section 2 gives a semantics for top-down pattern-matching, parameterised so that we can discuss different evaluation orders. Section 3 contains further motivation of and discussion on parallel pattern-matching. Section 4 describes the features of Concurrent Haskell that we use in our scheme. Section 5 is the crux of the paper: it describes how our parallel-matched definitions execute in Concurrent Haskell, using || as an example. Section 6 describes the general translation algorithm that we use and Section 7 discusses how the algorithm applies to two more-significant examples. Section 8 discusses some performance aspects of the scheme and Section 9 concludes the paper.

2 A Parameterised Semantics for Top-down Pattern-matching

Consider a function f, defined by n equations.

$$\begin{aligned} &\mathsf{f}\ \mathsf{P}_1 = \mathsf{E}_1 \\ &\mathsf{f}\ \mathsf{P}_2 = \mathsf{E}_2 \\ &\ldots \\ &\mathsf{f}\ \mathsf{P}_n = \mathsf{E}_n \end{aligned}$$

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

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

ξ is defined in Figure 1. 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.

Note that curried functions can be viewed as though they are defined in uncurried fashion without loss of generality.

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

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

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

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

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 definition of $\otimes$ for left-to-right pattern-matching (as in standard Haskell) is shown in Table 2. The definition of $\otimes$ for strict pattern-matching (which restores commutativity at some cost in laziness) is shown in Table 3.

This parameterised semantics is closely related to the semantics given in [Field et al, 1992] and [While, 1994].

x	y	$x \otimes_{LtoR} y$
tt	tt	tt
tt	ff	ff
ff	tt	ff
ff	ff	ff
tt	⊥	⊥
⊥	tt	⊥
ff	⊥	ff
⊥	ff	⊥
⊥	⊥	⊥

Table 2: The definition of ⊗ for left-to-right pattern-matching. y is needed only if x matches. Note that ⊥ is returned in *four* cases.

x	y	$x \otimes_{strict} y$
tt	tt	tt
tt	ff	ff
ff	tt	ff
ff	ff	ff
tt	⊥	⊥
⊥	tt	⊥
ff	⊥	⊥
⊥	ff	⊥
⊥	⊥	⊥

Table 3: The definition of ⊗ for strict pattern-matching. Both x and y are always needed. Note that ⊥ is returned in *five* cases, corresponding to its extra strictness.

3 The Case for Parallel Pattern-matching

Pattern-matching contributes greatly to the expressive power of functional languages and the readability of programs, as discussed in Section 1. However, it also introduces some problems, particularly in two areas: the use of abstract data types, and the order of evaluation of function arguments.

3.1 Abstract data types

In order to construct functions which pattern-match on values of a data type, a programmer must have knowledge of and access to the constructors used to build values of that type. Thus there is a tension between the use of pattern-matching and the use of abstract data types and information hiding. We do not discuss this problem further here: the reader is referred to [Thompson, 1986] [Wadler, 1987] [Burton and Cameron, 1993] [Gostanza et al, 1996] for further discussion.

3.2 Order of evaluation

Writing an equation which matches on more than one argument (or component of an argument) does not require the programmer to consider the order in which those values are matched. However, a sequential semantics for a language must specify an ordering on the evaluation of function arguments. Haskell, like most languages, specifies that the arguments to a function are matched from left-to-right, in effect giving the ⊗ operator the definition from Table 2. This difference between the intuitive reading of programs and their sequential implementation can lead to the non-termination of programs in counter-intuitive ways and to the invalidation of some types of program transformations.

We have already seen in Section 1 the problems caused even for simple binary logical operators like || and &&. More complex definitions, e.g. with nested patterns over lists, can exhibit even more counter-intuitive behaviour with respect to termination and errors. Modifying such definitions, e.g. by changing the order of the arguments or by combining arguments together, can change their termination properties in subtle ways. Given that program transformation is promoted widely as a way to develop Haskell programs and to improve their performance (e.g. in [Bird and Wadler, 1988]), it seems that this is a problem for which a solution would be very welcome.

We note also that this is not solely a problem for definitions with overlapping patterns. An example of a function definition with disjoint patterns but no symmetrical sequential implementation is the function awkward.

```
awkward :: [a] -> [a] -> [a] -> [a]
-- awkward has no symmetrical sequential impln
awkward (x : xs) [ ]      zs       = xs
awkward [ ]      ys       (z : zs) = zs
awkward xs       (y : ys) [ ]      = ys
```

In fact, the use of overlapping patterns permits the construction of function definitions which are potentially “lazier” than their disjoint counterparts. One example is the function || from Section 1: the use of disjoint patterns to define || would mean that either the definition must nominate which argument is evaluated first, or it must be strict in both arguments.

The solution which we propose to this problem is to remove the implicit ordering from the semantics of the language. This leads to so-called *parallel pattern-matching*:

> every argument (or component of an argument) is (conceptually) matched simultaneously, and a failure to match any argument (or component) means that an equation is rejected.

The definition of the ⊗ operator for parallel pattern-matching is shown in Table 4. This definition of ⊗ is bottom-avoiding: it promotes pattern-matching failure (of individual equations) whenever either argument fails to match.

x	y	$x \otimes_{ppm} y$
tt	tt	tt
tt	ff	ff
ff	tt	ff
ff	ff	ff
tt	⊥	⊥
⊥	tt	⊥
ff	⊥	ff
⊥	ff	ff
⊥	⊥	⊥

Table 4: The definition of ⊗ for parallel pattern-matching. Each argument is needed only if the other matches. Note that ⊥ is returned in only *three* cases, corresponding to its extra laziness.

This proposal is not new: it has been discussed several times previously (e.g. [Plotkin, 1977], [While, 1994], [Longley, 1999]). However, to our knowledge, there has never been a serious attempt to implement parallel pattern-matching in a real compiler: it has always been dismissed as too expensive in execution time, due to the requirement for concurrent evaluation. We hope that our implementation, when mature, will help to answer this question.

Finally, we note that other authors have opined that concurrent evaluation is essential to allow the writing of pure functional programs with complexity as good as their imperative equivalents[Hughes, 1984]. Whilst not directly connected to pattern-matching, this provides further motivation to investigate an evaluation model of this sort.

4 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 pertinent to our scheme for implementing parallel pattern-matching.

4.1 The IO data type

Haskell is a pure functional language, so programs written in Haskell are said to be *referentially transparent*[Bird and Wadler, 1988]. Specifically, they give the same result regardless of execution order. One consequence of this is that most functions are written without regard for the order of evaluation of the parts of expressions. However, some programming tasks require the programmer to have explicit control over the order of their evaluation, in particular programs involving I/O and other operations that involve side-effects.

To satisfy this requirement, Haskell provides the IO data type. A function with a return-type of the form IO t specifies a sequence of actions to be performed that return a value of type t. The actions are usually listed within the do construct, and they are performed in exactly the order specified in this construct. The return-type of the function is derived from the type of the last action in the sequence. A simple example of a function written in this way is cp, which copies a file.

```
cp :: FilePath -> FilePath -> IO ( )
-- cp f f' makes a copy of f with the name f'
cp f f' = do cs <- readFile f
             writeFile f' cs
```

This facility does not compromise referential transparency because the system performs the actions specified in a function only if that function is at the "top-level" of the program.

Note that the return-type IO ( ) indicates a function that specifies a sequence of actions with no value returned. The significance of such a function lies in the side-effects performed by its component actions.

4.2 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.

4.3 Expression-locking

The lazy semantics of Haskell means that an invocation of forkIO may pass a reference to an unevaluated expression to the newly-created process, while keeping a live reference to that expression itself. The possibility then exists that two or more processes will try to evaluate the same expression concurrently, which could cause corruption of that expression and incorrect evaluation. To avoid these problems, Concurrent Haskell implements an implicit locking mechanism which causes a process to block when it attempts to evaluate an expression already undergoing evaluation by another process. The blocked process is rescheduled once the evaluation of the shared expression is complete.

4.4 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.

4.5 Exceptions

When a run-time error occurs in a Haskell program, an *exception* is raised. This terminates the evaluation of the current thread. It is also possible for one thread to asynchronously raise an exception in another thread using the action

```
throwTo :: ThreadId -> Exception -> IO ()
```

Ordinarily, this causes the recipient thread to terminate immediately, but devices are provided by which a thread may either

1. temporarily block asynchronous exceptions, or
2. install a function g as an exception-handler. g is triggered to clean up after an exception is raised.

The function

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

uses both of these devices to enclose its 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.

4.6 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, as its name suggests. 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, e.g. 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 5). This means that we can use unsafePerformIO with impunity to invoke our pattern-matching mechanism anywhere in a program.

5 Parallel Pattern-matching in Concurrent Haskell

To illustrate how applications of parallel-matched functions are evaluated in our scheme, we first consider the || example again. The definition that we generate for || is shown in Figure 2.

The new definition is fairly straightforward. 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 requires an infinite computation.* The precise behaviour required from the function ppm is as follows.

```
(||) :: Bool -> Bool -> Bool
-- x || y returns False iff both x and y are False
False || False = False
x     || y     = True

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

(||) :: Bool -> Bool -> Bool
-- x || y returns False iff both x and y are False
x || y | ppm [not x, not y] = False
       | otherwise          = True
```

Figure 2: The symmetrical definition of || from Section 1, and the Concurrent Haskell definition, implementing parallel pattern-matching.

- If any element on its argument list returns False, ppm returns False; *otherwise*
- if any element goes into a loop, ppm goes into a loop (because it can never be sure that there *isn't* a False coming); *otherwise*
- if any element causes an error, ppm propagates the error; *otherwise*
- all of the elements must have returned True, so ppm returns True.

The Concurrent Haskell definition of ppm is shown in Figure 3. 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 4.

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.
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

```
module PPM (ppm)

where

import Concurrent
import IOExts

ppm :: [Bool] -> Bool
-- ppm bs returns True iff all of the values on bs
-- are True; moreover, it returns False if any of the
-- values on bs is False, in bottom-avoiding fashion
ppm = unsafePerformIO . ppm'

ppm' :: [Bool] -> IO Bool
-- ppm' bs creates a new channel ch;
-- creates a process to evaluate each expression
--         on bs and write the result to ch;
-- waits for either one False or all Trues
--         to be written to ch;
-- kills all remaining processes
ppm' bs =
   do ch <- newChan
      bracket
         (mapM (forkIO . report ch) bs)
         (mapM (`throwTo` PatternMatchFail "die"))
         (foldr (const (check ch)) (return True))

report :: Chan Bool -> Bool -> IO ( )
-- report ch b waits for b to be evaluated,
-- then writes its value on ch
report ch True = writeChan ch True
report ch False = writeChan ch False

check :: Chan Bool -> IO Bool -> IO Bool
-- check ch a waits for a value to appear on ch,
-- then returns either a or False as appropriate
check ch a = do h <- readChan ch
                if h then a
                     else  return False
```

Figure 3: The parallel pattern-matching module.

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

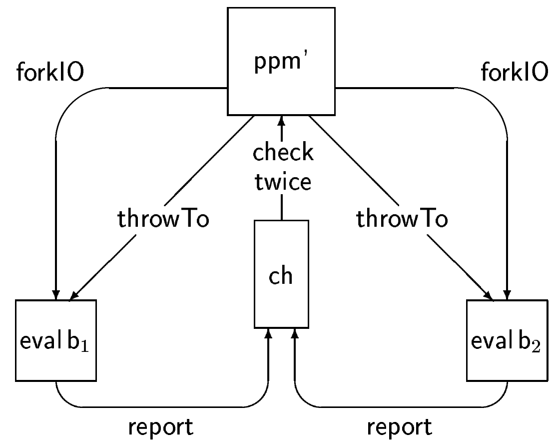

Figure 4: The process and communication structure created for ppm' [b_1, b_2]. ch is the channel created by ppm'.

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, as described in Section 4.3.

6 The General Translation Algorithm

We now give a description of the general translation algorithm. 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. This restriction can be lifted fairly easily, at the cost of some complexity in the algorithm. (Guards are dealt with by a mechanism similar to that described for result functions in Section 6.2.)

As in Section 2, 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.

The translated definition is composed of three classes of functions.

The auxiliary functions perform the actual 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.

6.1 The auxiliary functions

An auxiliary function is defined to test each of the constructor patterns in the equations of f. The recursive operator T 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 function g that is generated must be given a unique name. The overall form of the operator T is similar to the form of the matching operator M from the semantics in Figure 1.

Several transformation rules can be applied to simplify the definitions generated.

```
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, e.g. id and not for Bool (as in Figure 2), and null and not . null for lists (as in Figures 5 and 6). For user-defined data types, often many auxiliary functions are identical, and this repetition can of course be eliminated.

6.2 The result functions

The result functions duplicate the original equations: the ith 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.

```
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 2, because no names are used on the right-hand sides of the original equations.

6.3 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
```

7 Further Examples

We now describe two more-complex examples, showing the general translation algorithm in action.

7.1 awkward

awkward's nine patterns come in three forms.

A variable, e.g. xs

The empty list, i.e. [ ]

A non-empty list, e.g. x : xs

The first of these requires no matching: a variable matches any value. The operator T generates an auxiliary function for each of the other six patterns, giving functions of two general forms.

```
isEmpty [ ] = ppm [ ]
isEmpty _   = False

isNotEmpty (_ : _) = ppm [True, True]
isNotEmpty _       = False
```

Using the transformation rules listed for ppm in Section 6.1, the right-hand side of the first equation of each definition can be reduced to True, giving definitions of these forms.

```
isEmpty [ ] = True
isEmpty _   = False

isNotEmpty (_ : _) = True
isNotEmpty _       = False
```

The two tests in each equation of awkward must be performed "simultaneously", so each application of an auxiliary function appears on the argument to an application of ppm in the root function.

The result functions and the root function for awkward are trivial to derive. The final result of the translation is shown in Figure 5. The execution of the translated definition proceeds in the same way as the execution described for || in Section 5. Note that the separate invocations of ppm in awkward are never active simultaneously: the equations are tested top-down, so each equation is not tested until all preceding equations have failed to match.

7.2 Nested constructor applications

The function nested in Figure 6 is different to the previous examples because its equations have nested constructor applications in their patterns. This manifests itself in the translated definition as auxiliary functions that call other auxiliary functions. Given a pattern like [ys] (≡ ys : [ ]), obviously we cannot test the second argument of the : until we have established that there *is* a :. Furthermore, in a pattern like [[ ]] (≡ [ ] : [ ]), the two arguments to the : both need to be tested, so another invocation of ppm is required in the corresponding auxiliary function.

The result is that the translated definition of nested has calls to ppm at two levels.

- In the root function itself: these calls test the two arguments to nested concurrently.

```
awkward :: [a] -> [a] -> [a] -> [a]
-- awkward has no symmetrical sequential impln
awkward (x : xs) [ ]      zs       = xs
awkward [ ]      ys       (z : zs) = zs
awkward xs       (y : ys) [ ]      = ys

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

awkward :: [a] -> [a] -> [a] -> [a]
-- awkward has no symmetrical sequential impln
awkward xs ys zs | ppm [awk'11 xs, awk'12 ys]
                 = awkward'1 xs ys zs
                 | ppm [awk'21 xs, awk'23 zs]
                 = awkward'2 xs ys zs
                 | ppm [awk'32 ys, awk'33 zs]
                 = awkward'3 xs ys zs

awk'11 (_ : _) = True
awk'11 _       = False

awk'12 [ ] = True
awk'12 _   = False

awk'21 [ ] = True
awk'21 _   = False

awk'23 (_ : _) = True
awk'23 _       = False

awk'32 (_ : _) = True
awk'32 _       = False

awk'33 [ ] = True
awk'33 _   = False

awkward'1 (x : xs) [ ]      zs       = xs
awkward'2 [ ]      ys       (z : zs) = zs
awkward'3 xs       (y : ys) [ ]      = ys

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

awkward :: [a] -> [a] -> [a] -> [a]
-- awkward has no symmetrical sequential impln
awkward xs ys zs
              | ppm [not (null xs), null ys] = tail xs
              | ppm [null xs, not (null zs)] = tail zs
              | ppm [not (null ys), null zs] = tail ys
```

Figure 5: The function awkward from Section 3, and the Concurrent Haskell definition, implementing parallel pattern-matching. The third definition is the final version after post-processing, exploiting a number of built-in list functions.

```
nested :: [a] -> [[b]] -> Int
-- nested's patterns have nested constructor appns
nested [ ]      [[ ]]    = 1
nested (x : xs) [y : ys] = 2
nested xs       [ys]     = 3

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

nested :: [a] -> [[b]] -> Int
-- nested requires nested invocations of ppm
nested xs yss | ppm [n'11 xs, n'12 yss] = n'1 xs yss
              | ppm [n'21 xs, n'22 yss] = n'2 xs yss
              | n'32 yss                = n'3 xs yss

n'11 [ ] = True
n'11 _   = False

n'12 (xs : xss) = ppm [n'121 xs, n'122 xss]
n'12 _          = False

n'121 [ ] = True
n'121 _   = False

n'122 [ ] = True
n'122 _   = False

n'21 (_ : _) = True
n'21 _       = False

n'22 (xs : xss) = ppm [n'221 xs, n'222 xss]
n'22 _          = False

n'221 (_ : _) = True
n'221 _       = False

n'222 [ ] = True
n'222 _   = False

n'32 (_ : xss) = n'322 xss
n'32 _         = False

n'322 [ ] = True
n'322 _   = False

n'1 [ ]      [[ ]]    = 1
n'2 (x : xs) [y : ys] = 2
n'3 xs       [ys]     = 3

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

nested :: [a] -> [[b]] -> Int
-- nested requires nested invocations of ppm
nested xs yss | ppm [      null xs,  n'12 yss] = 1
              | ppm [not (null xs), n'22 yss] = 2
              | n'32 yss                      = 3

n'12 (xs : xss) = ppm [null xs, null xss]
n'12 _          = False

n'22 (xs : xss) = ppm [not (null xs), null xss]
n'22 _          = False

n'32 (_ : xss) = null xss
n'32 _         = False
```

Figure 6: The function nested has nested constructor applications in its patterns, so it requires nested invocations of ppm in its Concurrent Haskell definition. The third definition is the final version.

- In the auxiliary functions responsible for [[ ]] and [y : ys]: these calls test the two arguments to the : concurrently.

The final result of the translation of nested is shown in Figure 6. Note that if the second argument to nested is non-empty, then in the execution of this definition the nested invocations of ppm may be active simultaneously. As noted previously in Section 5, this will not cause any problem: the processes belonging to each invocation are controlled by separate channels, and thus the processes cannot interact in any way.

8 Performance Issues

Using parallel pattern-matching is likely to have a major impact on the performance of a Haskell program. The cost of concurrent evaluation will slow down most programs, often by a significant amount. The best way to limit the impact is by minimising the need for concurrency. One good way to approach this is through the use of program transformation.

Consider, for example, the function nested from Figure 6. If we process the equations separately, as described in Section 6, the first two equations of nested both need the two arguments to be evaluated concurrently. However, if we consider the definition of nested as a whole, we can transform it into the function nested', shown in Figure 7.

```
nested :: [a] -> [[b]] -> Int
-- nested needs concurrency if the equations
-- are processed separately
nested [ ]      [[ ]]    = 1
nested (x : xs) [y : ys] = 2
nested xs       [ys]     = 3

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

nested' :: [a] -> [[b]] -> Int
-- nested' clearly does not need concurrency
nested' xs [ys] = g xs ys

g :: [a] -> [b] -> Int
-- g clearly does not need concurrency
g [ ]      ys = g1 ys
g (x : xs) ys = g2 ys

g1 :: [b] -> Int
-- g1 clearly does not need concurrency
g1 [ ]      = 1
g1 (y : ys) = 3

g2 :: [b] -> Int
-- g2 clearly does not need concurrency
g2 (y : ys) = 2
g2 [ ]      = 3
```

Figure 7: An example transformation. nested and nested' give the same results for all arguments under parallel pattern-matching, but nested' clearly does not need concurrency.

nested' returns the same result as nested for all possible arguments, but nested' has an obvious sequential implementation. The key is the third equation of nested, which requires only the second argument to be evaluated. Using program transformation techniques, we can propagate this asymmetry up through the definition, and we can evaluate the second argument before starting on the first, without compromising the semantics.

(Note that this transformation is valid only for Haskell with parallel pattern-matching. nested and nested' are not equivalent in standard Haskell.)

Figure 8 shows another sort of transformation. The patterns of the function h are identical to those of the function || from Figure 2, which we know requires concurrency. However, if we also consider the right-hand sides of the equations, there is an important difference: the function not is strict[Bird and Wadler, 1988], so the second equation of h needs to know the value of the second argument in order to return a result. Again, we can use this information to derive an equivalent function h' that evaluates sequentially.

```
h :: Bool -> Bool -> Bool
-- h has the same patterns as ||,
-- and appears to need concurrency
h False False = False
h x     y     = not y

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

h' :: Bool -> Bool -> Bool
-- h' clearly does not need concurrency
h' x False = x
h' x True  = False
```

Figure 8: A transformation based on strictness information. h and h' give the same results for all arguments under parallel pattern-matching, but h' clearly does not need concurrency.

We have developed and reported on these techniques previously[Field et al, 1992][While, 1994], and we plan to incorporate them into our implementation before performing a serious analysis of the performance impact of parallel pattern-matching.

There are two other ways in which our scheme would benefit from considering definitions as a whole.

- Under the scheme we have described, equations are matched independently, so often a value is inspected multiple times. Consider the application

  awkward [2] [5, 6] [ ]

  When this application is evaluated, each of the arguments is inspected twice, once by each of two of the equations. Clearly this is wasteful. Say, for example, that the first equation has established that the second argument is not empty: there is no need for the third equation to test this argument again.

- Complementary to this is the issue of killing processes when their work is still needed. Consider an application of the form

  awkward e [9] e'

  and assume that the expression e is expensive to evaluate. When this application is evaluated, the first equation creates processes to evaluate the first two arguments. The second argument soon reports a failed match, at which point the process evaluating e is killed. Then the second equation immediately creates a new process to evaluate e! Again, this is wasteful.

We intend to implement these kinds of optimisations in the next phase of our research. Two other issues are more-generally associated with concurrency and with the use of Concurrent Haskell.

- Having multiple processes working on different parts of the program graph creates two obvious issues. The first is that the work done by some processes may be wasted, in that it doesn't contribute to the final result of the program. The second issue is resource allocation, and in particular the space leaks that may arise from having partially-evaluated expressions sitting in the heap.
- Concurrent Haskell has no notion of process priorities: all processes are scheduled preemptively. However, it is clear that a priority mechanism would be useful in our scheme. One scenario is where there are nested invocations of ppm, e.g. in an expression of the form

  ```
  awkward e (awkward e1 e2 e3) [8]
  ```

  The process evaluating e should really have higher priority than the processes evaluating e1, e2 and e3. If e comes back with the right value (i.e. [ ]), the other processes could be killed, while the same is not true for any other single process. e is "more likely" than the other expressions to contribute to the result of the program.

These seem to be general issues that are difficult to resolve.

9 Conclusions

We have described the semantics of and motivation for parallel pattern-matching in lazy functional languages. Parallel pattern-matching gives the "maximum laziness" for any function definition: arguments are evaluated concurrently and all arguments are given equal precedence, thus pattern-matching failure (of individual equations) is promoted, and the function can return a result whenever possible. This is important for predictable behaviour in the presence of errors and infinite computations, particularly when non-trivial program transformations are applied.

More significantly, we have implemented parallel pattern-matching in a real compiler for the first time. The implementation works by translating definitions at the source-level into Concurrent Haskell. These definitions are executed using the Concurrent Haskell primitives already provided in GHC: concurrently-evaluating processes communicating on atomically-mutable shared channels.

The performance impact of parallel pattern-matching is likely to be considerable. The next phase of our work will be to optimise the implementation as much as possible, so that we can accurately estimate the effect of these semantics on the execution times of programs. The principal optimisation available is to avoid the use of concurrency wherever possible by transforming functions, again at the source-level, into equivalent definitions that have sequential implementations which respect the semantics. Other optimisations revolve mainly around minimising the cost of creating and maintaining processes and channels.

Acknowledgements

We should like to thank Nick Jardine for his contribution to an earlier version of this work, and Luigi Barone for proof-reading an earlier draft of the paper.

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.

[Burton and Cameron, 1993] BURTON, F.W. AND CAMERON, R.D. (1993). *Pattern-matching with Abstract Data Types.* Journal of Functional Programming, 3(2):171–90.

[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.

[Gostanza et al, 1996] GOSTANZA, P.P., PEÑA, R. AND NÚÑEZ, M. (1996). *A New Look at Pattern-matching in Abstract Data Types.* 1996 International Conference on Functional Programming, Philadelphia, Pennsylvania.

[Hughes, 1984] HUGHES, R.J.M. (1984). *The Design and Implementation of Programming Languages.* Ph.D. thesis, University of Oxford.

[Hunt, 1986] HUNT, L.S. (1986). *A Hope-to-FLIC Translator with Strictness Analysis.* M.Sc. dissertation, Dept. of Computing, Imperial College.

[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.* 23rd ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, 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.

[Thompson, 1986] THOMPSON, S. (1986). *Laws in Miranda.* 4th ACM Conference on Lisp and Functional Programming, Boston, Massachusetts.

[Wadler, 1987] WADLER, P.L. (1987). *Views: a Way for Pattern-matching to Co-habit with Data Abstraction.* 14th ACM Symposium on Principles of Programming Languages, Munich, Germany.

[While, 1994] WHILE, R.L. (1994). *Parallel Pattern-matching in Lazy Functional Languages.* 2nd Massey Functional Programming Workshop, Massey University, New Zealand.