*JFP* **20** (5 & 6): 537–576, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796810000201 First published online 27 January 2011

# *Implicitly threaded parallelism in Manticore*

MATTHEW FLUET[*]
*Computer Science Department, Rochester Institute of Technology, Rochester, NY, USA*
(*e-mail:* mtf@cs.rit.edu)

MIKE RAINEY
*Department of Computer Science, University of Chicago, Chicago, IL, USA*
(*e-mail:* mrainey@cs.uchicago.edu)

JOHN REPPY
*Department of Computer Science, University of Chicago, Chicago, IL, USA*
(*e-mail:* jhr@cs.uchicago.edu)

ADAM SHAW
*Department of Computer Science, University of Chicago, Chicago, IL, USA*
(*e-mail:* ams@cs.uchicago.edu)

## Abstract

The increasing availability of commodity multicore processors is making parallel computing ever more widespread. In order to exploit its potential, programmers need languages that make the benefits of parallelism accessible and understandable. Previous parallel languages have traditionally been intended for large-scale scientific computing, and they tend not to be well suited to programming the applications one typically finds on a desktop system. Thus, we need new parallel-language designs that address a broader spectrum of applications. The Manticore project is our effort to address this need. At its core is Parallel ML, a high-level functional language for programming parallel applications on commodity multicore hardware. Parallel ML provides a diverse collection of parallel constructs for different granularities of work. In this paper, we focus on the implicitly threaded parallel constructs of the language, which support fine-grained parallelism. We concentrate on those elements that distinguish our design from related ones, namely, a novel parallel binding form, a nondeterministic parallel case form, and the treatment of exceptions in the presence of data parallelism. These features differentiate the present work from related work on functional data-parallel language designs, which have focused largely on parallel problems with regular structure and the compiler transformations—most notably, flattening—that make such designs feasible. We present detailed examples utilizing various mechanisms of the language and give a formal description of our implementation.

## 1 Introduction

Parallel processors are becoming ubiquitous, which creates a software challenge: how do we harness this newly-available parallelism across a broad range of applications?

[*] Portions of this work were completed while the author was affiliated with the Toyota Technological Institute at Chicago.

We believe that existing general-purpose languages do not provide adequate support for parallel programming, while most existing parallel languages, which are largely targeted at scientific applications, do not provide adequate support for general-purpose programming. We need new languages to maximize application performance on these new processors.

A homogeneous language design is not likely to take full advantage of the hardware resources available. For example, a language that provides data parallelism but not explicit concurrency is inconvenient for the coarse-grained concurrent elements of a program, such as its networking and GUI components. On the other hand, a language that provides concurrency but not data parallelism is ill-suited to the components of a program that demand fine-grained parallelism, such as image processing and particle systems.

Our belief is that parallel programming languages must provide mechanisms for multiple levels of parallelism, both because applications exhibit parallelism at multiple levels and because hardware requires parallelism at multiple levels to maximize performance. Indeed, a number of research projects are exploring *heterogeneous* parallelism in languages that combine support for parallel computation at different levels into a common linguistic and execution framework. The Glasgow Haskell Compiler (GHC n.d.) has been extended with support for three different paradigms for parallel programming: explicit concurrency coordinated with transactional memory (Peyton Jones *et al.* 1996; Harris *et al.* 2005), semi-implicit concurrency based on annotations (Trinder *et al.* 1998), and nested data parallelism (Chakravarty *et al.* 2007), the last paradigm inspired by NESL (Blelloch *et al.* 1994; Blelloch 1996).

The Manticore project (Fluet *et al.* 2007a, 2007b) is our effort to address the problem of parallel programming for commodity systems. It consists of a parallel runtime system (Fluet *et al.* 2008b) and a compiler for a parallel dialect of Standard ML (SML) (Milner *et al.* 1997), called Parallel ML (PML). The PML design incorporates mechanisms for both coarse-grained and fine-grained parallelism. Its coarse-grained parallelism is based on Concurrent ML (CML) (Reppy 1991), which provides explicit concurrency and synchronous message passing. PML's fine-grained mechanisms include nested data parallelism in the style of NESL (Blelloch *et al.* 1994; Blelloch 1996; Blelloch & Greiner 1996) and Nepal (Chakravarty & Keller 2000; Chakravarty *et al.* 2001; Leshchinskiy *et al.* 2006), as well as other novel constructs described below.

This paper focuses on the design and implementation of the fine-grained parallel mechanisms in PML. After an overview of the PML language (Section 2), we present four main technical contributions:

- the **`pval`** binding form, for parallel evaluation and speculation (Section 4),
- the **`pcase`** expression form, for nondeterminism and user-defined parallel control structures (Section 5),
- the support of exceptions and exception handlers as a key component of data-parallel programming (Section 6), and
- a formalization of our implementation of all of the above (Section 8).

We describe the nested data parallelism mechanism (parallel arrays) in Section 3, and we illustrate the language design with a series of examples in Section 7. Our method has been to collect together *varied* mechanisms in order to provide the programmer a diverse group of complementary tools for attacking many different kinds of parallel programming problems. The examples are meant to demonstrate PML's flexibility, as well as its suitability for irregular parallel applications. We review related work and conclude in Sections 9 and 10.

## 2 An overview of the PML language

Parallel language mechanisms can be roughly grouped into three categories:

- *Implicit parallelism*, where the compiler and runtime system are solely responsible for partitioning the computation into parallel threads. Examples of this approach include Id (Nikhil 1991), pH (Nikhil & Arvind 2001), and Sisal (Gaudiot *et al.* 1997).
- *Implicit threading*, where the programmer provides annotations, or hints to the compiler, as to which parts of the program are profitable for parallel evaluation, while the mapping of computation onto parallel threads is left to the compiler and runtime system. Examples include NESL (Blelloch 1996) and its descendants Nepal (Chakravarty *et al.* 2001) and Data Parallel Haskell (DPH) (Chakravarty *et al.* 2007).
- *Explicit threading*, where the programmer explicitly creates parallel threads. Examples include CML (Reppy 1991) and Erlang (Armstrong *et al.* 1996).

These design points represent different trade-offs between programmer effort and programmer control. Automatic techniques for parallelization have proven effective for dense regular parallel computations (e.g., dense matrix algorithms) but have been less successful for irregular problems.

PML provides both implicit threading and explicit threading mechanisms. The former supports fine-grained parallel computation, while the latter supports coarse-grained parallel tasks and explicit concurrent programming. These parallelism mechanisms are built on top of a sequential functional language. In the sequel, we discuss each of these in turn, starting with the sequential base language. For a more complete account of PML's language design philosophy, goals, and target domain, we refer the reader to our previous publications (Fluet *et al.* 2007a, 2007b).

### *2.1 Sequential programming*

The PML's sequential core is based on a subset of SML. The main differences are that PML does not have mutable data (i.e., reference cells and arrays) and implements only a subset of SML's module system. PML does include the functional elements of SML (datatypes, polymorphism, type inference, and higher-order functions) as well as exceptions. As many researchers have observed, using a mutation-free language greatly simplifies the implementation and use of parallel features (Hammond 1991; Reppy 1991; Jones & Hudak 1993; Nikhil & Arvind 2001; Dean & Ghemawat

2004). In essence, mutation-free functional programming reduces interference and data dependencies—it provides data separation for free. We recognize that the lack of mutable data means that certain techniques, such as path compression and cyclic data structures, are not supported, but there is evidence of successful languages that lack this feature, such as Erlang (Armstrong *et al.* 1996). The interaction of exceptions and our implicit threading mechanisms adds some complexity to our design, as we discuss below, but we believe that an exception mechanism is necessary for systems programming.

As the syntax and semantics of the sequential core language are largely orthogonal to the parallel language mechanisms, we have resisted tinkering with core SML. The PML Basis, however, differs significantly from the SML Basis Library (Gansner & Reppy 2004). In particular, we have a fixed set of numeric types—`int`, `long`, `float`, and `double`—instead of SML's families of numeric modules; furthermore, integer operations provide modular arithmetic (and do not raise an `Overflow` exception).

## 2.2 *Explicitly threaded parallelism*

The explicit concurrent programming mechanisms presented in PML serve two purposes: they support concurrent programming, which is an important feature for systems programming (Hauser *et al.* 1993), and they support explicit parallel programming. Like CML, PML supports threads that are explicitly created using the **`spawn`** primitive. Threads do not share mutable state; rather they use synchronous message passing over typed channels to communicate and synchronize. Additionally, we use CML communication mechanisms to represent the interface to system features such as input/output.

The main intellectual contribution of CML's design is an abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions called *event values*. This encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. Events can range from simple message-passing operations to client-server protocols to protocols in a distributed system. Further details about the design and implementation of CML's concurrency mechanisms can be found in the literature (Reppy 1991; Reppy 1999; Reppy *et al.* 2009).

## 2.3 *Implicitly threaded parallelism*

PML provides implicitly threaded parallel versions of a number of sequential forms. These constructs can be viewed as hints to the compiler and runtime system about which computations are good candidates for parallel execution. Most of these constructs have deterministic semantics, which are specified by a translation to equivalent sequential forms (Shaw 2007). Having a deterministic semantics is important for several reasons:

- It gives the programmer a predictable programming model;

```
datatype tree
  = Lf of int
  | Nd of tree * tree

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
      (op * ) (|trProd1 tL, trProd1 tR|)
```

Fig. 1. Tree product with parallel tuples.

- Algorithms can be designed and debugged as sequential code before porting to a parallel implementation; and
- It formalizes the expected behavior of the compiler.

The requirement to preserve a sequential semantics does place a burden on the implementation. For example, we must verify that subcomputations in an implicit-parallel construct do not send or receive messages. If they do so, the construct must be executed sequentially. Similarly, if a subcomputation raises an exception, the implementation must delay the delivery of the exception until all sequentially prior computations have terminated. We consider the issues related to the propagation of exceptions in more detail in Section 6.

### *2.3.1 Parallel tuples*

Parallel tuple expressions are the simplest implicitly threaded construct in PML. The expression

$$(|e_1,\ \dots,\ e_n|)$$

serves as a hint to the compiler and runtime system that the subexpressions $e_1, \dots, e_n$ may be usefully evaluated in parallel. This construct describes a fork-join parallel decomposition, where up to $n$ threads may be forked to compute the expression. There is an implicit barrier synchronization on the completion of all of the subcomputations. The result is a normal tuple value. Figure 1 illustrates the use of parallel tuples to compute the product of the leaves of a binary tree of integers.

The sequential semantics of parallel tuples is trivial: they are evaluated simply as sequential tuples. The sequential semantics immediately determines the behavior of an exception-raising subexpression: if an exception is raised when computing its $i$th element, then we must wait until all preceding elements have been computed before propagating the exception.

### *2.3.2 Parallel arrays*

Support for parallel computations on arrays is common in parallel languages. In PML, we support such computations by using a nested parallel array mechanism that was inspired by NESL (Blelloch 1996), Nepal (Chakravarty *et al.* 2001), and DPH (Chakravarty *et al.* 2007). A parallel array expression has the form

$$[|e_1,\ \dots,\ e_n|]$$

which constructs an array of $n$ elements. The delimiters `[| |]` alert the compiler that the $e_i$ may be evaluated in parallel.

Parallel array values may also be constructed using *parallel comprehensions*, which allow concise expressions of parallel loops. A comprehension has the general form

```
[| e | p1 in e1, ..., pn in en where ef |]
```

where $e$ is an expression (with free variables bound in the $p_i$) computing the elements of the array, $p_i$ are patterns binding the elements of $e_i$, which are array-valued expressions, and $e_f$ is an optional boolean-valued expression that is used to filter the input. If the input arrays have different lengths, all are truncated to the length of the shortest input, and they are processed, in parallel, in lockstep.[1] For convenience, we also provide a parallel range form

```
[| el to eh by es |]
```

which is useful in combination with comprehensions. (The step expression "**by** $e_s$" is optional and defaults to "**by** 1.")

### 2.3.3 Parallel bindings

Parallel tuples and arrays provide fork-join patterns of computation, but in some cases, more flexible scheduling is desirable. In particular, we may wish to execute some computations speculatively. PML provides the parallel binding form

```
let pval p = e1
in
  e2
end
```

that hints to the system that running $e_1$ in parallel with $e_2$ would be profitable. The sequential semantics of a parallel binding is similar to lazy evaluation: the binding of the value of $e_1$ to the pattern $p$ is delayed until one of the variables in $p$ is used. Thus, if an exception were to be raised in $e_1$ or the matching to the pattern $p$ were to fail, it is raised at the point where a variable from $p$ is first used. In the parallel implementation, we use eager evaluation for parallel bindings, but computations are canceled when the main thread of control reaches a point where their result is guaranteed never to be demanded.

### 2.3.4 Parallel case

The parallel case expression form is a parallel nondeterministic counterpart to SML's sequential case form. Parallel case expressions have the following structure:

```
pcase e1 & ... & em
 of π1,1 & ... & πm,1 => f1
  | ...
  | π1,n & ... & πm,n => fn
```

[1] This behavior is known as *zip semantics*, since the comprehension loops over the zip of the inputs. Both NESL and Nepal use zip semantics, but Data Parallel Haskell (Chakravarty *et al.* 2007) supports both zip semantics and *Cartesian-product semantics* where the iteration is over the product of the inputs.

```
fun up() = up()
val x = (pcase up() & ... & up() & 1
    of 1 & ... & ? & ? => false
     | ? & ... & ? & 1 => true)
```

Fig. 2. An illustration of the infinite processor assumption.

Here, both $e$ and $f$ range over expressions. The expressions $e_i$, which we refer to as the *subcomputations* of the parallel case, evaluate in parallel with one another. Note that **pcase** uses ampersands (&) to separate both the subcomputations and the corresponding patterns from one another. This syntax simultaneously avoids potential confusion with tuples and tuple patterns and recalls the related join-pattern syntax of JoCaml (Mandel & Maranget 2008).

The $\pi_{i,j}$ in a parallel case are *parallel patterns*, which are either normal patterns or the special *nondeterministic wildcard* `?`. A normal wildcard matches a finished computation and effectively discards it by not binding its result to a name. A nondeterministic wildcard, by contrast, matches a computation (and does not name it) even if it has not yet finished.

Unlike the other implicitly threaded mechanisms, parallel case is nondeterministic. We can still give a sequential semantics, but it requires including a source of nondeterminism, such as McCarthy's **amb** (McCarthy 1963), in the sequential language.

PML operates under an *infinite processor assumption*, which is of particular importance with respect to the subcomputations of parallel case expressions. That is, there is always an additional (virtual) processor available for the spawning of new computational threads: the machine never "fills up." Our semantics asserts that all terminating subcomputations of a parallel case will be computed to completion, even in the presence of diverging subcomputations running in parallel with them. Consider the excerpt in Figure 2. Even though the (arbitrarily many) calls to `up` will never terminate, the constant `1` enables a match with the second branch, and the value `x` must evaluate to `true`. Please note that this semantic detail neither prevents the programmer from writing infinite loops using **pcase** nor guarantees termination in other parallel constructs such as parallel tuples, where, for example, the expression

```
(| up(), 1 |)
```

does in fact diverge, in keeping with its sequential semantics. We delay further discussion of parallel case until Section 5.

## 3 Parallel arrays

Parallel arrays, like lists, vectors, and arrays, are ordered sequences whose values are all of the same type. The elements of parallel arrays can all be computed simultaneously. PML supports a standard complement of parallel functional collection operations over parallel arrays, including mapping, filtering, and reducing with an associative operator.

Parallel comprehensions in PML are similar to those of NESL. For example, to double each positive integer in a given parallel array of integers `nums`, one may use the following expression:

```
[| 2 * n | n in nums where n > 0 |]
```

This expression can be evaluated efficiently in parallel using vector instructions.

Parallel array comprehensions are first-class expressions; hence, the expression defining the elements of a comprehension can itself be a comprehension. For example, the main loop of a ray tracer generating an image of width `w` and height `h` can be written as

```
[| [| traceRay(x,y) | x in [| 0 to w-1 |] |]
                    | y in [| 0 to h-1 |] |]
```

This parallel comprehension within a parallel comprehension is an example of *nested data parallelism.*

A key aspect of nested data parallelism is that the dimensions of the nested arrays do not have to be the same. This feature allows many irregular-parallel algorithms to be encoded as nested data parallel algorithms. One of the simplest examples of this technique is sparse matrices, which can be represented by an array of rows, where each row is an array of index-value pairs. This has the type

```
type sparse_vector = (int * float) parray
type sparse_matrix = sparse_vector parray
```

We can define the dot product of a sparse vector and a dense vector as an array comprehension:

```
fun dotp (sv, v) = sumP [| x * v!i | (i,x) in sv |]
```

Using that operation, multiplying a sparse matrix times a dense vector is

```
fun smvm (sm, v) = [| dotp (row, v) | row in sm |]
```

The sequential semantics of parallel arrays is defined by translating them to lists (see (Fluet *et al.* 2007a) or (Shaw 2007) for details). The main subtlety in the parallel implementation is that if an exception is raised when computing its $i$th element, then we must wait until all preceding elements have been computed before propagating the exception. Section 8 describes our implementation strategy for this behavior.

There is an important difference between parallel tuples and parallel arrays: with parallel tuples, the elements may have different types, while with parallel arrays, the elements must have the same type. In languages with only a parallel array construct, a programmer can evaluate expressions of different types by, for example, injecting them into an *ad hoc* union datatype, collecting them in a parallel array, and then projecting them out of that datatype, but this incurs uninteresting complexity in the program and adds runtime overhead.

## 4 Parallel bindings

Parallel bindings allow more flexibility in decomposing computations into parallel subtasks than the fork-join patterns provided by parallel tuples and arrays. Furthermore, they support speculative parallelism, since the implementation is able to

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = let
      pval pL = trProd tL
      pval pR = trProd tR
      in
        if (pL = 0)
          then 0
          else (pL * pR)
      end
```

Fig. 3. Short-circuiting tree product with parallel bindings.

identify and cancel unneeded computations in progress. Our compiler uses a program analysis to determine those program points where a subcomputation is guaranteed never to be demanded and, thus, a cancelation may be inserted.

As in Figure 1, the function in Figure 3 computes the product of the leaves of a tree. This version short-circuits, however, when the product of the left subtree of a `Nd` variant evaluates to zero. Note that if the result of the left product is zero, we do not need the result of the right product. Therefore its subcomputation and any descendants may be canceled. This cancelation behavior is not visible in the semantics of **pval**, but is an optimization provided by the implementation. The analysis to determine when a **pval**'s computation is subject to cancelation is not as straightforward as it might seem. The following example includes two parallel bindings linked by a common computation:

```
val v = let
  pval x = f 0
  pval y = (| g 1, x |)
  in
    if b then x else h y
  end
```

In the conditional expression, the computation of `y` can be canceled in the **then** branch, but the computation of `x` cannot be canceled in either branch because `y` depends on `x`. Our analysis, explained in detail in Section 8, must respect this and other similar subtle dependencies.

There are many more examples of the use of parallel bindings in Section 7. We discuss the specific mechanisms—most importantly, *futures* (Halstead 1984)—by which we realize their semantics in Section 8. A future is, in brief, a computation whose evaluation is ongoing in parallel with subsequent computations, until its result is demanded, or *touched*, at which point the program blocks until its value is available.

Note that a speculative computation bound with a **pval** can escape its immediate scope by its inclusion in a closure. Consider the following expression:

```
val g = let
  pval x = f 1
  in
    fn y => x + y
  end
```

Note that the value `x` is bound to the computation `f 1`, which may at any point in the program still be ongoing. The value of `x` will be demanded at any applications

```
fun mkFuture susp = let
  pval x = susp ()
  in
    fn () => x
  end
```

Fig. 4. Future encoding with **pval**.

of g, at which point the future will be touched and its result forced if not already evaluated. This behavior enables the following lightweight encoding of futures to be written directly in PML's surface language, as shown in Figure 4. Touching a future encoded in this way is simply applying it to unit.

## 5 Parallel case expressions

We recall the syntax of the parallel case construct first, before giving an account of its semantics. We make a distinction between *sequential patterns*, which are, informally, destructuring patterns as they appear in ML and other languages, and *parallel patterns*, which are defined below.

```
pcase $e_1$ & ... & $e_m$
 of $\pi_{1,1}$ & ... & $\pi_{m,1}$ => $f_1$
  | ...
  | $\pi_{1,n}$ & ... & $\pi_{m,n}$ => $f_n$
```

The $\pi_{i,j}$ metavariables denote parallel patterns. A parallel pattern is either

- $p$, a sequential pattern, or
- ?, a *nondeterministic wildcard pattern*.

We refer to the expressions $e_1 \ldots e_m$ as the *subcomputations* of the **pcase** expression and the patterns as the *discriminants*. Each arm of the expression, with discriminants on the left-hand side and an expression $f_j$ on the right-hand side, is a *branch*.[2] The dynamic behavior of a parallel case is as follows: the expression's subcomputations execute in parallel, and, periodically, the branches are compared with the subcomputations. The branches are compared in order, and a branch that matches the subcomputations transfers control to its right-hand side. Note that if more than one branch matches the evaluating or evaluated subcomputations, then the textually first branch is taken.

A *nondeterministic wildcard pattern*, written ?, can match either any finished computation or a computation that is still running. Whether the computation in question is not yet finished, it has finished normally by evaluating to a value, or it has terminated by raising an exception, the ? matches the computation, does not bind it to any name, and proceeds. Sequential wildcards, by contrast, must wait for the potentially matching computation to complete before matching and proceeding. The choice between a nondeterministic wildcard and a sequential wildcard is consequential and has profound effects on the behavior of a program.

[2] We have dropped the **otherwise** branch form (Fluet *et al.* 2008a) from our design; it is convenient but not necessary.

```
pcase e1 & e2
  of x & ? => x
   | ? & y => y
```

Fig. 5. Parallel choice with **pcase**.

Nondeterministic wildcards can be used to implement *speculative computation.* Speculation is an important tool for programming in PML and other parallel languages, notably Cilk and JCilk. Consider the following **pcase** expression:

```
pcase isPrime(1024) & longRunning()
  of false & ? => 0
```

Once the constant pattern `false` has been matched against the result of the first subcomputation, which we assume happens quickly, the program need not wait for `longRunning` to finish; it can immediately return `0`. During the compilation process, the compiler will enact this improvement by inserting an explicit cancelation of the `longRunning` subcomputation on the right-hand side of the branch, before evaluating and returning `0`. In this way, the resources devoted to the fruitless computation can be dynamically released and made available elsewhere.

Originally, the parallel case expression was designed as a generalization of *parallel choice.* A parallel choice expression nondeterministically returns either of two subexpressions `e1` or `e2`. We write parallel choice with the infix operator `|?|`, as in

$$e_1 \;\texttt{|?|}\; e_2$$

This construct is useful in a parallel context, because it gives the program the opportunity to return whichever of $e_1$ or $e_2$—two computations that might be running in parallel—evaluates first.

As an example, we might want to write a function to obtain the value of a leaf—any leaf—from a given tree. (We use the tree datatype defined in Figure 1.)

```
fun trLeaf (Lf i) = i
  | trLeaf (Nd (tL, tR)) = trLeaf(tL) |?| trLeaf(tR)
```

This function evaluates `trLeaf(tL)` and `trLeaf(tR)` in parallel. Whichever finishes sooner, loosely speaking, determines the value of the choice expression as a whole. Hence, the function is likely, though not required, to return the value of the shallowest leaf in the tree. Furthermore, the evaluation of the discarded component of the choice expression—that is, the one whose result is not returned—is canceled, as its result is known not to be demanded. If the computation is running, this cancelation will make computational resources available for use elsewhere. If the computation is completed, this cancelation will be a harmless nonoperation.

The parallel choice operator is a derived form in PML, as it can be expressed as a **pcase** in a straightforward manner. The expression $e_1$ `|?|` $e_2$ is equivalent to the expression in Figure 5.

By slightly modifying this usage pattern, other powerful parallel idioms arise. For example, parallel case gives us yet another way to write the `trProd` function (see Figure 6). This function will short-circuit when either the first or the second branch

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = (
      pcase trProd(tL) & trProd(tR)
       of 0 & ? => 0
        | ? & 0 => 0
        | pL & pR => pL * pR)
```

Fig. 6. Short-circuiting tree product with parallel case.

```
fun trFind (p, Lf i) =
      if p(i) then SOME(i) else NONE
  | trFind (p, Nd (tL, tR)) =
      pcase trFind(p,tL) & trFind(p,tR)
       of SOME(n) & ? => SOME(n)
        | ? & SOME(n) => SOME(n)
        | NONE & NONE => NONE
```

Fig. 7. Finding an element in a tree, using a parallel abort pattern.

is matched, implicitly canceling the computation of the other subtree. Note that this short-circuiting behavior is symmetric, unlike similar encodings using parallel bindings, for example. Because it is nondeterministic as to which of the matching branches is taken, the programmer must ensure that all branches that match the same results yield sensible answers. Specifically, if both `trProd(tL)` and `trProd(tR)` eventually evaluate to `0`, then either the first or the second branch may be taken, but either right-hand side will yield the correct result, `0`.

As a second example, consider a function to find a leaf value in a tree that satisfies a given predicate `p`. The function should return an `int option` to account for the possibility that no leaf values in the tree match the predicate. We might mistakenly write the following code:

```
fun trFindB (p, Lf i) = (* B for broken *)
      if p(i) then SOME(i) else NONE
  | trFindB (p, Nd (tL, tR)) =
      trFindB(p,tL) |?| trFindB(p,tR)
```

In the case where the predicate `p` is not satisfied by any leaf values in the tree, this implementation will always return `NONE`, as it should. However, if the predicate is satisfied at some leaf, the function will nondeterministically return either `SOME(n)`, for a satisfying `n`, or `NONE`. In other words, this implementation will never return a false positive, but it will, nondeterministically, return a false negative. The reason for this is that as soon as one of the operands of the parallel choice operator evaluates to `NONE`, the evaluation of the other operand might be canceled, even if it were eventually to yield `SOME(n)`.

A correct version of `trFind` appears in Figure 7. When either `trFind(p,tL)` or `trFind(p,tR)` evaluates to `SOME(n)`, the function returns that value and implicitly cancels the other evaluation. The essential computational pattern here is an *abort mechanism.*

We believe that the abort mechanisms that are part of important idioms in, for example, the Cilk and JCilk programming languages can be encoded in PML by means of the **`pcase`** mechanism and, in some cases, the exception system. In JCilk,

```
(* solve : state -> unit *)
  fun solve(s) = if isSol(s)
        then raise Sol(s)
        else (case next(s)
           of NONE => ()
            | SOME(t,u) =>
                (pcase solve(t) & solve(u) of () & () => ())
          (* end case *))

(* main : state -> state option *)
  fun main(init) = (solve(init); NONE) handle Sol(s) => SOME(s)
```

Fig. 8. Searching a tree space speculatively with exceptions.

a spawned computation can be aborted when that computation raises an exception and is in turn caught in a `try/catch` block. A recent JCilk publication (Danaher *et al.* 2006) presents a parallel *n*-queens solver with the following strategy: spawn a solver, which in turn spawns subsolvers in parallel, with any process that finds a solution raising an exception that includes that solution as data. That exception can be caught in a `try/catch` block, bypassing the stack that may have built up in the meantime. The solution value contained in that exception value can then be collected and used elsewhere in the program.

We present an abstract problem solver that roughly follows the strategy given in that parallel *n*-queens solver in Figure 8. We assume the existence of a `state` type, representing the state of whatever space (e.g., chess boards) is being searched; a predicate `isSol` for identifying solution states; and a function `next` which returns `SOME` pair[3] of successor states to the current state or `NONE` if the current state is terminal. As soon as a solution is discovered in the `solve` function, it is wrapped in an exception value and raised to the outer context.

This example highlights an important facet of the semantics of **`pcase`**: if the evaluation of a subcomputation raises an exception, then, nondeterministically, the **`pcase`** reraises the exception (and cancels the other subcomputations).[4] The nondeterminism in exception propagation is due to the fact that a **`pcase`** may successfully match a branch before the subcomputation raises the exception. There is an additional element of nondeterminism in that if multiple subcomputations raise exceptions, then the reraised exception is chosen nondeterministically. This nondeterminism is acceptable for this example, since the solver is allowed to return any solution. If we had used a parallel tuple instead of a **`pcase`**, then we would have imposed additional overhead on the computation. In the expression

```
(| solve(t), solve(u) |)
```

[3] This can be generalized to any finite number of successor states. If `next` were to return an arbitrary number of states, we need some slightly heavier weight tools.

[4] In a prior publication (Fluet *et al.* 2008a), we presented *handle patterns* as an additional kind of parallel pattern that would match an exception-raising subcomputation. We have since removed it from our design in the interest of simplicity.

```
(* solve : state -> state option *)
  fun solve(s) = if isSol(s)
        then SOME(s)
        else (case next(s)
           of NONE => NONE
            | SOME(t,u) => (pcase solve(t) & solve(u)
                 of SOME(s) & ? => SOME(s)
                  | ? & SOME(s) => SOME(s)
                  | NONE & NONE => NONE
                (* end pcase *))
          (* end case *))

(* main : state -> state option *)
  fun main(init) = solve(init)
```

Fig. 9. Searching a tree space speculatively with options.

if an exception were raised in evaluating `solve(u)`, the system would, in keeping with PML's sequential semantics, be obliged to wait for `solve(t)` to complete to see if it too were raising an exception, in which case that exception would take precedence. The combination of **`pcase`** and exceptions is the means by which parallel nonlocal exit is achieved in PML programming.

It is worth noting that in JCilk, all programs that abort speculative computations do so by means of the exception system. In PML, the exception system is only one way of encoding speculation. We present another abstract problem solver in Figure 9, using options in place of exceptions. Thus, we have two stylistically different ways of writing similar parallel programs with similar intentions, but which might be better written in one style or another depending on their purposes and expected uses.

### *5.1 A comparison of **`pval`** and **`pcase`***

There are many computations one can express either as a **`pcase`** or as a **`pval`**, sometimes with no clear advantage either way. At a glance, it might seem that among **`pval`** and **`pcase`**, one form could be used to derive the other. This is not the case. For one, **`pval`** expressions are deterministic, while **`pcase`** expressions are nondeterministic. Another important difference is that ongoing computations are able to escape the scope of a **`pval`**, whereas all such computations are either touched or canceled in the evaluation of a **`pcase`**. They are distinct control structures describing different behaviors, and there are cases where only a **`pval`** or **`pcase`** will do as we show below.

Since **`pval`** is a deterministic expression form, it cannot be used to express parallel choice as shown in Figure 5. The code in Figure 5 enables whichever subexpression finishes evaluating first to match a branch and become the value of the whole expression, enabling a (benevolent) race between two computations that are in some sense interchangeable. Any attempt to encode similar behavior with **`pval`** is doomed. Consider the following expression:

```
let pval f1 = e₁
    pval f2 = e₂
in case coinFlip()
    of Heads => f1
     | Tails => f2
end
```

Like parallel choice, this returns the value of either $e_1$ or $e_2$; however, the choice of which value to return is left entirely to a `coinFlip` and has nothing to do with whichever of the two expressions ran faster than the other. In other words, even if $e_1$ requires much more time to compute than $e_2$, if `coinFlip` chooses $e_1$, the program is stuck waiting for it. Clearly, this is not the desired parallel choice behavior. No matter how we attempt to simulate **pcase**'s nondeterminism, we will encounter similar difficulties.

There is no way to construct a **pcase** that allows an ongoing computation to escape its scope. In Figure 4, we presented a use of **pval** that is designed to allow a running computation to escape its scope by capturing it inside a closure. Any time we evaluate some computation `susp ()` as a subcomputation of a **pcase**, we have two choices. Assuming we are not matching the value against a constant, we can either name it with a pattern

```
pcase susp ()
  of x => (fn _ => x)
```

or ignore it with a wildcard pattern (either `?` or `_`)

```
pcase susp ()
  of ? => ...
```

In the first expression, we complete the evaluation of `susp ()` at the point of binding its result to `x`; in the second, by using a wildcard, we forfeit the ability to refer to the result (and we should cancel the computation if it is still running). In neither case do we allow the computation to continue computing, as we did in Figure 4.

The following **pval** also has no **pcase** equivalent:

```
let pval x = e₁
in if t₁ then x+1
   else if t₂ then x-1 else 0
end
```

In this expression, the computation of $e_1$ is started, and $t_1$ is evaluated in parallel. If $t_1$ is `false`, then $t_2$ starts evaluating. Note that the evaluation of $e_1$ need not be finished at the point $t_2$ begins. If $t_2$ is `true`, then we wait for `x` to be ready in the true branch; otherwise we cancel it in the false branch. The important points are that $e_1$ can run throughout the evaluation of $t_1$ and $t_2$ and $t_1$ is done before any evaluation of $t_2$.

If we attempt to translate this expression to a **pcase**, we evaluate either the second conditional after matching `false`

```
pcase t₁ & e₁
  of true  & x => x+1
   | false & x => if t₂ then x-1 else 0
end
```

or all three expressions at the same time:

```
pcase t1 & t2 & e1
  of true &  ?    & x => x+1
   | false & true  & x => x-1
   | false & false & ?  => 0
```

In the former expression, x is demanded before we begin to compute $t_2$. In the latter, $t_1$, $t_2$, and $e_1$ are all computed in parallel together. In the former, $t_2$ has no influence on whether we wait for x to finish or not; it is finished before we get there. In the latter, if $t_2$ is, for example, nonsense if $t_1$ is true, we nonetheless expend resources computing it; it is no longer guarded by $t_1$. In either case, the would-be translations have different behavior than the original program, so we cannot in general consider either one a proper translation.

There are also some less critical mismatches between **pval** and **pcase**, where although meanings can be preserved in translation, clarity and performance suffer. Order-sensitive pattern matching, for example, a fundamental technique of ML-style programming, is sometimes an awkward fit for **pcase**, since any matching branch can be chosen at any time. For example, the following expression runs $e_1$ and $e_2$ in parallel, returning 1 if both evaluate to Yes and 2 otherwise.

```
datatype answer = Yes | No | Maybe
let pval y = e2
in case (e1, y)
  of (Yes, Yes) => 1
   | _          => 2
end
```

If we use **pcase** to write a similar program, it is much more verbose. Since we cannot depend on the order of branches in a **pcase**, we are forced to write down a set of mutually disjoint branches.

```
pcase e1 & e2
  of Yes & Yes => 1
   | No & ?    => 2
   | ?  & No   => 2
   | Maybe & ? => 2
   | ? & Maybe => 2
end
```

This is not merely a syntactic inconvenience; it compiles to a more expensive piece of object code (per the implementation sketch in Section 8.4). We present an example in Section 7.1, where we favor a **pval** for essentially this reason. This problem gets worse when there are more computations to run in parallel or more datatype variants against which to match.

There remains a middle ground where either a **pval** or a **pcase** would do. In those instances, the programmer is free to choose whichever construct expresses the desired program more naturally.

## 6 Exceptions and exception handlers

The interaction of exceptions and parallel constructs must be considered in the implementation of the parallel constructs. Raises and exception handlers are first-class expressions, and, hence, they may appear at arbitrary points in a program, including in a parallel construct. The following is a legal parallel array expression:

```
[| 2+3, 5-7, raise A |]
```

Evaluating this parallel array expression should raise the exception A.

Note the following important detail. Since the compiler and runtime system are free to execute the subcomputations of a parallel array expression in any order, there is no guarantee that the first **raise** expression observed during the parallel execution corresponds to the first **raise** expression observed during a sequential execution. Thus, some compensation is required to ensure that the sequentially first exception in a given parallel array (or other implicitly threaded parallel construct with a deterministic sequential semantics) is raised whenever multiple exceptions could be raised. Consider the following minimal example:

```
[| raise A, raise B |]
```

Although B might be raised before A during a parallel execution, A must be the exception observed to be raised in order to adhere to the sequential semantics. Realizing this behavior in this and other parallel constructs requires our implementation to include compensation code, with some runtime overhead. In the present work, we give details of our implementation of this behavior, compensation code included, in Section 8.

In choosing to adopt a strict sequential core language, PML is committed to realizing a precise exception semantics in most of the implicitly threaded parallel features of the language (the exception is the nondeterministic **pcase** expression). This is in contrast to an imprecise exception semantics (Peyton Jones *et al.* 1999) that arises from a lazy sequential language. While a precise semantics requires a slightly more restrictive implementation of the implicitly threaded parallel features than would be required with an imprecise semantics, we believe that support for exceptions and the precise semantics is crucial for systems programming. Furthermore, as Section 8 will show, implementing the precise exception semantics is not particularly onerous.

It is possible to eliminate some or all of the compensation code with the help of program analyses. There already exist various well-known analyses for identifying exceptions that might be raised by a given computation (Yi 1998; Leroy & Pessaux 2000). If, in a parallel array expression, it is determined that no subcomputation may raise an exception, then we are able to omit the compensation code and its overhead. As another example, consider a parallel array expression where all subcomputations can raise only one and the same exception.

```
[| if x<0 then raise A else 0,
   if y>0 then raise A else 0 |]
```

The full complement of compensation code is unnecessary here, since any exception raised by any subcomputation must be A.

Exception handlers are attached to expressions with the **handle** keyword:

$e_1$ **handle** $e_2$

Although exception handlers are first-class expressions, their behavior is orthogonal to that of the parallel constructs and mostly merits no special treatment in the implementation. At the present time, our PML compiler does not implement any form of flattening transformation on data-parallel array computations. Once we incorporate flattening into our work, however, we will need to take particular account of exception handlers, since flattening and exception handlers are not orthogonal (Shaw 2007).

Note that when an exception is raised in a parallel context, the implementation should free any resources devoted to parallel computations whose results will never be demanded by virtue of the control flow of the raise. For example, in the parallel tuple

```
(| raise A, fact(100), fib(200) |)
```

the latter two computations should be abandoned as soon as possible. Section 8 details our approaches when this and similar issues arise.

## 7 Examples

We consider a few examples to illustrate the use and interaction of our language features in familiar contexts. We choose examples that stress the parallel binding and parallel case mechanisms of our design, since examples exhibiting the use of parallel arrays and comprehensions are covered well in the existing literature.

### *7.1 A parallel typechecking interpreter*

First we consider an extended example of writing a parallel typechecker and evaluator for a simple model programming language. The language in question, which we outline below, is a pure expression language with some basic features including boolean and arithmetic operators, conditionals, let bindings, and function definition and application. A program in this language is, as usual, represented as an expression tree. Both typechecking and evaluation can be implemented as walks over expression trees, in parallel when possible. Furthermore, the typechecking and evaluation can be performed in parallel with one another. In our example, failure to type a program successfully implicitly cancels its simultaneous evaluation.

While this is not necessarily intended as a realistic example, one might wonder why parallel typechecking and evaluation is desirable in the first place. First, typechecking constitutes a single pass over the given program. If the program involves, say, recursive computation, then typechecking might finish well before evaluation. If it does, and if there is a type error, the presumably doomed evaluation will be spared the rest of its run. Furthermore, typechecking touches all parts of a program; evaluation might not.

Our language includes the following definition of types.

```
datatype ty = Bool | Nat | Arrow of ty * ty
```

For the purposes of yielding more useful type errors, we assume that each expression consists of a location (some representation of its position in the source program) and a term (its computational part). These are encoded as follows:

```
datatype term
  = N of int | B of bool | V of var
  | Add of exp * exp
  | If of exp * exp * exp
  | Let of var * exp * exp
  | Lam of var * ty * exp
  | Apply of exp * exp
    ...
withtype exp = loc * term
```

For typechecking, we need a function that checks the equality of types. When we compare two arrow types, we can compare the domains of both types in parallel with comparison of the ranges. Furthermore, if either the domains or the ranges turn out to be not equal, we can cancel the other comparison. Here we encode this, in the `Arrow` case, as an explicit short-circuiting parallel computation:

```
fun tyEq (Bool, Bool) = true
  | tyEq (Nat, Nat) = true
  | tyEq (Arrow(t,t'), Arrow(u,u')) =
     (pcase tyEq(t,u) & tyEq(t',u')
       of false & ? => false
        | ? & false => false
        | true & true => true)
  | tyEq _ = false
```

We present a parallel typechecker as a function `typeOf` that consumes an environment (a map from variables to types) and an expression. It returns either a type, in the case that the expression is well typed, or an error, in the case that the expression is ill-typed. We introduce a simple union type to capture the notion of a value or an error.

```
datatype 'a or_error
  = A of 'a
  | Err of loc
```

The signature of `typeOf` is

```
val typeOf : env * exp -> ty or_error
```

We consider a few representative cases of the `typeOf` function. To typecheck an `Add` node, we can simultaneously check both subexpressions. If the first subexpression is not of type `Nat`, we can record the error and implicitly cancel the checking of the second subexpression. The function behaves similarly if the first subexpression returns an error.

```
fun typeOf (G, (p, Add (e1,e2))) = let
      pval t2 = typeOf(G, e2)
      in
        case typeOf(G, e1)
         of A Nat => (case t2
               of A Nat => A Nat
                | A _ => Err(locOf(e2))
                | Err q => Err q)
          | A _ => Err(locOf(e1))
          | Err q => Err q
      end
```

Note that we choose to use a sequential **case** inside a **pval** block in this implementation. This way we guarantee that errors in e1 are caught if present. Had we used a pcase, we would catch errors in e1 or e2 nondeterministically (if both were in error). That might be acceptable, but in this example, we demonstrate the former approach.

In the Apply case, we require an arrow type for the first subexpression and the appropriate domain type for the second.

```
| typeOf (G, (p, Apply (e1, e2))) = let
    pval t2 = typeOf(G, e2)
    in
      case typeOf(G, e1)
       of A(Arrow(d,r)) => (case t2
             of A t => if tyEq(d,t) then A r else Err p
              | Err q => Err q)
        | A _ => Err(locOf(e1))
        | Err q => Err q
    end
```

Where there are no independent subexpressions, no parallelism is available:

```
| typeOf (G, (p, IsZero(e))) = (case typeOf(G,e)
     of A Nat => A Bool
      | _ => Err p)
```

Throughout these examples, the programmer rather than the compiler is identifying opportunities for parallelism.

For evaluation, we need a function to substitute a term for a variable in an expression. Substitution of closed terms for variables in a pure language is well suited to a parallel implementation. Parallel instances of substitution are completely independent, so no subtle synchronization or cancelation behavior is ever required. Parallel substitution can be accomplished by means of our simplest parallel construct, the parallel tuple. We show a few cases here.

```
fun subst (t, x, e as (p, t')) = (case t'
       of V(y) => if varEq(x,y) then (p,t) else e
        | Let(y,e1,e2) => if varEq(x,y)
            then (p, Let(y, subst(t,x,e1), e2))
            else (p, Let(|y, subst(t,x,e1), subst(t,x,e2)|))
      ...
```

Like the parallel typechecking function, the parallel evaluation function simultaneously evaluates subexpressions. Since we are not interested in identifying the first runtime error (when one exists), we use a parallel case for expressions for which all subexpressions must be evaluated:

```
| eval (p, Add(e1,e2)) = (pcase eval(e1) & eval(e2)
      of N(n1) & N(n2) => N(n1+n2)
       | _ & _ => raise RuntimeError)
```

Note that a raised `RuntimeError` in either `eval(e1)` or `eval(e2)` will be propagated through the **`pcase`** to the surrounding context.

The `If` case is notable in its use of speculative evaluation of both branches. As soon as the test completes, the abandoned branch is implicitly canceled.

```
| eval (p, If(e1, e2, e3)) = let
    pval v2 = eval(e2)
    pval v3 = eval(e3)
    in
      case eval(e1)
       of B(true) => v1
        | B(false) => v2
        | _ => raise RuntimeError
    end
```

We conclude the example by wrapping typechecking and evaluation together into a function that runs them in parallel. If the typechecker discovers an error, the program implicitly cancels the evaluation. If the typechecking function returns any type at all, we discard it and return the value computed by the evaluator.

```
fun typedEval e =
  (pcase typeOf(emptyEnv,e) &
         (SOME(eval(e)) handle RuntimeError => NONE)
    of (Err p, ?) => Err p
     | (A _, SOME v) => A v)
```

## *7.2 Parallel game search*

We now consider the problem of searching a game tree in parallel. This has been shown to be a successful technique by the Cilk group for games such as Pousse (Barton *et al.* 1998) and chess (Dailey & Leiserson 2002).

For simplicity, we consider the game of tic-tac-toe. In this implementation, every tic-tac-toe board is associated with a score: 1 if X holds a winning position, −1 if O holds a winning position, and 0 otherwise. We use the following polymorphic rose tree to store a tic-tac-toe game tree.

```
datatype 'a rose_tree
  = Rose of 'a * 'a rose_tree parray
```

Each node contains a board and the associated score, and every path from the root of the tree to a leaf encodes a complete game.

A `player` is either of the nullary constructors `X` or `O`; a `board` is a parallel array of nine `player options`, where `NONE` represents an empty square. Extracting the available moves from a given board is written as a parallel comprehension as follows:

```
fun allMoves b = [|i | s in b, i in [|0 to 8|] where isNone(s)|]
```

Generating the next group of boards given a current board and a player to move is also a parallel comprehension:

```
fun successors (b, p) = [| moveTo (b, p, i) | i in allMoves b |]
```

With these auxiliaries in hand, we can write a function to build the full game tree by using the standard minimax algorithm, where each player assumes the opponent will play the best available move at the given point in the game.

```
fun minimax (b : board, p : player) = if gameOver(b)
      then Rose ((b, boardScore b), [||])
      else let
        val ss = successors (b, p)
        val ch = [| minimax (b, other p) | b in ss |]
        val chScores = [| treeScore t | t in ch |]
        in
          case p
           of X => Rose ((b, maxP chScores), ch)
            | O => Rose ((b, minP chScores), ch)
        end
```

Note that at every node in the tree, all subtrees can be computed independently of one another, as they have no interrelationships. Admittedly, one would not write a real tic-tac-toe player this way, as it omits numerous obvious and well-known improvements. Nevertheless, as written, it exhibits a high degree of parallelism and performs well relative to both a sequential version of itself in PML and similar programs in other languages.

Using alpha–beta pruning yields a somewhat more realistic example. We implement it here as a pair of mutually recursive functions, `maxT` and `minT`. The code for `maxT` is shown in Figure 10, omitting some obvious helper functions; `minT`, not shown, is similar to `maxT`, with appropriate symmetrical modifications. Alpha–beta pruning is an inherently sequential algorithm, so we must adjust it slightly. This program prunes subtrees at a particular level of the search tree if they are at least as disadvantageous to the current player as an already-computed subtree. (The sequential algorithm, by contrast, considers every subtree computed thus far.) We compute one subtree sequentially as a starting point, then use its value as the pruning cutoff for the rest of the sibling subtrees. Those siblings are computed in parallel by repeatedly spawning computations in an inner loop by means of **`pval`**. Pruning occurs when the implicit cancelation of the **`pval`** mechanism cancels the evaluation of the right siblings of a particular subtree.

## 8 Implementation

We sketch our implementation of PML by defining a formal translation from its high-level implicitly threaded constructs to a small collection of parallel operations supporting well-known idioms. The set of low-level parallel operations in our translated code includes *futures* (Halstead 1984) and *m-variables* (Barth *et al.* 1991) (the complete set is given in the next section). In the PML compiler, this transformation is performed on the abstract-syntax-tree (AST) representation.

```
fun maxT (board, alpha, beta) = if gameOver(board)
      then Rose ((board, boardScore board), [||])
      else let
        val ss = successors (board, X)
        val t0 = minT (ss!0, alpha, beta)
        val alpha' = max (alpha, treeScore t0)
        fun loop i = if (i = plen ss)
              then [||]
              else let
                pval ts = loop (i+1)
                val ti = minT (ss!i, alpha', beta)
                in
                  if (treeScore ti >= beta)
                    then [|ti|] (* prune *)
                    else [|ti|] |@| ts
                end
        val ch = [|t0|] |@| loop(1)
        val maxScore = maxP [| treeScore t | t in ch |]
        in
          Rose ((board, maxScore), ch)
        end
```

Fig. 10. The `maxT` half of parallel alpha–beta pruning.

We separate the formal translation into three distinct phases:

1. Translation of parallel arrays to *ropes* (Boehm *et al.* 1995) and parallel comprehensions to higher-order parallel functions over ropes. The parallel rope functions are implemented internally using PML's parallel tuples.
2. Translation of parallel tuples and parallel bindings to futures.
3. Translation of parallel cases.

By means of this multiple-phase approach, we achieve a separation of concerns; consequently, it becomes easier to reason about the translation's workings.

### *8.1 Low-level parallel primitives*

Before describing our transformations, we must describe the low-level parallel-language primitives to which we are compiling. The signature of these types and operations is given in Figure 11, and we describe them in the remainder of this section.

We use a variant of futures (Halstead 1984) to implement many of our implicitly threaded parallel constructs. A future value is a handle to a computation that may be executed parallel to the main thread of control. The `new` operation creates a new future from a thunk. The `touch` operation demands the result of the future computation, blocking until the computation has completed. If the subcomputation raised an exception, then that exception will be reraised by the `touch`. Lastly, the `cancel` operation terminates a future computation and any children of the computation. It is an unchecked error to `touch` a future value after it has been canceled, but a program may `cancel` a future value multiple times and may cancel

```
structure Future : sig
    type 'a future
    val new    : (unit -> 'a) -> 'a future
    val touch  : 'a future -> 'a
    val cancel : 'a future -> unit
  end

structure MVar : sig
    type 'a mvar
    val new  : 'a -> 'a mvar
    val take : 'a mvar -> 'a
    val put  : ('a mvar * 'a) -> unit
  end

structure Cancel : sig
    type cancelable
    val new     : unit -> cancelable
    val spawn   : (cancelable * (unit -> unit)) -> unit
    val cancel  : cancelable -> unit
  end
```

Fig. 11. Primitive parallel operations.

a future that has already been touched. In these cases, the `cancel` operation is a no-op. Many of the translations below depend on these properties of the `cancel` operation.

M-variables are a form of synchronous mutable memory (Barth *et al.* 1991). An m-variable has two states: empty and full. The `take` operation changes the state from full to empty and returns the contents of the variable. The `put` operation changes the state from empty to full by storing a value in the variable. Attempting to take a value from an empty variable causes the calling thread to block. This property means that a take–modify–put protocol is atomic.

As we discussed above, our futures support cancelation. That mechanism is built on a more primitive notion of *cancelable* objects that record parent–child relationships. The `spawn` operation creates a new thread of computation and associates it with a given cancelable object. It also makes the cancelable object a child of the object associated with the calling thread. If the `cancel` operation is used on a cancelable object, the associated thread and any children that it might have are all terminated and removed from the scheduling queues. As with futures, it is a no-op to cancel an already canceled object. It is also a no-op to cancel oneself. These properties simplify the use of cancelable objects. Cancelation is implemented using "scheduler actions", which are described elsewhere (Fluet *et al.* 2008b).

### *8.2 Translating parallel comprehensions*

Adopting DPH's strategy, the first phase of our translation rewrites parallel comprehensions as function applications. It is a translation from the source language in Figure 12 to itself such that no comprehensions remain after the rewriting. The rules for this translation are given in Figure 13 and follow the standard pattern developed

$$
\begin{array}{rcll}
e & ::= & x & \text{variables} \\
 & | & \mathbf{fn}\, x \Rightarrow e & \text{functions} \\
 & | & \mathbf{if}\; e_1\; \mathbf{then}\; e_2\; \mathbf{else}\; e_3 & \text{conditionals} \\
 & | & \mathbf{[|}\; e_1 \mid p\; \mathbf{in}\; e_2\; \mathbf{|]} & \text{parallel comprehensions} \\
 & | & \mathbf{[|}\; e_1 \mid p\; \mathbf{in}\; e_2\; \mathbf{where}\; e_3\; \mathbf{|]} & \text{parallel comprehensions with filter} \\
 & | & \mathbf{(|}\, e_1, e_2\, \mathbf{|)} & \text{parallel tuples} \\
 & | & \mathbf{plet}\; p = e_1\; \mathbf{in}\; e_2 & \text{parallel bindings} \\
 & | & \mathbf{pcase}\; e_1\; \mathbf{\&}\; e_2\; \mathbf{of}\; m_1 \mid \cdots \mid m_k & \text{parallel cases} \\
 & | & \ldots & \\
m & ::= & \pi_1\; \mathbf{\&}\; \pi_2 \Rightarrow e & \\
\pi & ::= & ? & \\
 & | & p & \\
p & ::= & x & \\
 & | & b & \\
 & | & \mathbf{(}p_1, p_2\mathbf{)} &
\end{array}
$$

Fig. 12. Source language for translation.

$$
[\![\mathbf{[|}\; e_1 \mid p\; \mathbf{in}\; e_2\; \mathbf{|]}]\!] = \begin{cases} \texttt{mapP}\; (\mathbf{fn}\, p \Rightarrow [\![e_1]\!])\; [\![e_2]\!] & \text{if } p \text{ is irrefutable} \\ \texttt{mapPartialP}\; (\mathscr{M}_1(p, e_1))\; [\![e_2]\!] & \text{otherwise} \end{cases}
$$

$$
[\![\mathbf{[|}\; e_1 \mid p\; \mathbf{in}\; e_2\; \mathbf{where}\; e_3\; \mathbf{|]}]\!] = \begin{cases} \texttt{mapPartialP}\; (\mathscr{M}_2(p, e_1, e_3))\; [\![e_2]\!] & \text{if } p \text{ is irrefutable} \\ \texttt{mapPartialP}\; (\mathscr{M}_3(p, e_1, e_3))\; [\![e_2]\!] & \text{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
\mathscr{M}_1(p, e) & = & \mathbf{fn}\; p \Rightarrow \texttt{SOME}[\![e]\!] \mid \_ \Rightarrow \texttt{NONE} \\
\mathscr{M}_2(p, e_b, e_t) & = & \mathbf{fn}\; p \Rightarrow \mathbf{if}\; [\![e_t]\!]\; \mathbf{then}\; \texttt{SOME}[\![e_b]\!]\; \mathbf{else}\; \texttt{NONE} \\
\mathscr{M}_3(p, e_b, e_t) & = & \mathbf{fn}\; p \Rightarrow \mathbf{if}\; [\![e_t]\!]\; \mathbf{then}\; \texttt{SOME}[\![e_b]\!]\; \mathbf{else}\; \texttt{NONE} \mid \_ \Rightarrow \texttt{NONE}
\end{array}
$$

Fig. 13. Translating parallel comprehensions.

for Nepal (Chakravarty *et al.* 2001). Only the rules rewriting parallel comprehensions are given here; the rules for other expression forms are straightforward recursive applications to subexpressions and are omitted. In the figure, terms to be translated are marked with double brackets, as in $[\![e]\!]$.

Note that we have abbreviated the source language by leaving out sequential forms, such as constants and function application, that do not affect the translation.

We represent parallel arrays by using an immutable balanced-binary-tree data structure called a rope (Boehm *et al.* 1995). Ropes, originally proposed as an alternative to strings, are immutable balanced binary trees with vectors of data at their leaves.

```
datatype 'a rope
  = Leaf of 'a vector
  | Cat of 'a rope * 'a rope
```

Read from left to right, the data elements at the leaves of a rope constitute the data of the parallel array it represents. Ropes support distributed construction and fast concatenation. One disadvantage of ropes is that random access to individual data elements requires logarithmic time. We do not expect this to present a problem for many programs, as random access to elements of a parallel array will, in many cases, not be needed. Nevertheless, a PML programmer should be aware of the potential performance implications of this representation for certain operations.

Since ropes are physically dispersed in memory, they are well suited to being built in parallel, with different processing elements simultaneously working on different parts of the whole. Furthermore, ropes embody a natural tree-shaped parallel decomposition of common parallel array operations like maps and reductions. Note that the rope datatype shown above is an oversimplification of our implementation for the purposes of presentation. In our prototype system, rope nodes also store their depth and data length. These values assist in balancing ropes and make length and depth queries constant time operations.

The parallelism in the translated code is provided by the `mapP` and `mapPartialP` operations over ropes. For example, the `mapP` function has the following implementation:

```
fun mapP f r = (case r
       of Leaf v => Leaf(Vector.map f v)
        | Cat(r1, r2) => Cat(| mapP f r1, mapP f r2 |)
      (* end case *))
```

The `mapPartialP` combinator, named after SML's `mapPartial`, has the following type specification:

```
val mapPartialP : ('a -> 'b option) -> 'a rope -> 'b rope
```

(The analogous Haskell function, on lists, is called `mapMaybe`.) Its implementation is similar to `mapP`, except that at the leaves, it maps a partial function across the elements, which may result in fewer results than elements. Thus, as the new rope is constructed, it must be rebalanced (we implement a parallel version of Boehm's original rope-balancing algorithm, Boehm *et al.* 1995). We prefer to use `mapP` over `mapPartialP` when possible to avoid the extra cost of balancing.

### *8.3 Translation of parallel tuples and parallel bindings*

The second phase of our translation eliminates parallel tuples and bindings and replaces them with futures, touches, and cancelations of unneeded futures. The translation is from the model source language in Figure 12 without comprehensions, which have been eliminated in the first phase, to the model target language in Figure 14. Note that this language supports the future operations discussed above as syntactic forms: specifically, **`future`**, **`touch`** and **`cancel`** in Figure 14 correspond to `Future.new`, `Future.touch`, and `Future.cancel` in Figure 11. We present the rules in Figure 15. The translation introduces futures at each parallel tuple and **`plet`**, binding them to fresh variables. At the end of this phase, only **`pcase`** expressions remain among the implicitly threaded constructs.

Translation rules are written $\mathscr{T}[\![e]\!]\rho\,\varepsilon$, where $e$ is a source language expression, $\rho$ is a finite map from source language variables to pairs of future variables and patterns, and $\varepsilon$ is a set of future variables that should not be canceled, because

$$
\begin{array}{lcll}
e & ::= & x & \text{variables} \\
 & | & \mathtt{\textbf{fn}}\; x \Rightarrow e & \text{functions} \\
 & | & \mathtt{\textbf{if}}\; e_1\; \mathtt{\textbf{then}}\; e_2\; \mathtt{\textbf{else}}\; e_3 & \text{variables} \\
 & | & \mathtt{\textbf{(}}e_1,\, e_2\mathtt{\textbf{)}} & \text{tuples (pairs)} \\
 & | & \mathtt{\textbf{let}}\; p \;=\; e_1\; \mathtt{\textbf{in}}\; e_2 & \text{binding} \\
 & | & \mathtt{\textbf{pcase}}\; e_1\; \mathtt{\textbf{\&}}\; e_2\; \mathtt{\textbf{of}}\; m_1 \mid \cdots \mid m_k & \text{parallel case} \\
 & | & \mathtt{\textbf{future}}\; e & \text{future creation} \\
 & | & \mathtt{\textbf{touch}}\; e & \text{future touch} \\
 & | & \mathtt{\textbf{cancel}}\; x\; \mathtt{\textbf{in}}\; e & \text{future cancelation} \\
 & | & \cdots & \\
m & ::= & p \Rightarrow e &
\end{array}
$$

Fig. 14. Target language for translation.

$$
\begin{array}{rcl}
\rho & : & \mathit{Var} \rightarrow \mathit{Var} \times \mathit{Pat} \\
\epsilon & : & \{\mathit{Var}\} \\
\mathrm{Fu}(\rho,\, e) & = & \{y \mid \exists x \in \mathrm{FV}(e) \text{ such that } \rho(x) = (y, p)\} \\
 & & (\mathrm{FV}(e) \text{ defined in Figure 16}) \\
\mathscr{T}\llbracket x \rrbracket \rho\, \varepsilon & = & \left\{ \begin{array}{ll} \mathtt{\textbf{let}}\; p \;=\; \mathtt{\textbf{touch}}\; y\; \mathtt{\textbf{in}}\; x & \text{if } \rho(x) = (y, p) \\ x & \text{if } x \notin \mathrm{dom}(\rho) \end{array} \right. \\
\mathscr{T}\llbracket \mathtt{\textbf{fn}}\; x \Rightarrow e \rrbracket \rho\, \varepsilon & = & \mathtt{\textbf{fn}}\; x \Rightarrow \mathscr{T}\llbracket e \rrbracket \rho\, \varepsilon \\
\mathscr{T}\llbracket \mathtt{\textbf{(|}}e_1,\, e_2\mathtt{\textbf{|)}} \rrbracket \rho\, \varepsilon & = & \mathtt{\textbf{let}}\; y \;=\; \mathtt{\textbf{future}}\; (\mathscr{T}\llbracket e_2 \rrbracket \rho\, \varepsilon')\; \mathtt{\textbf{in}}\; b \\
 & & \quad \text{where } y \text{ is fresh} \\
 & & \qquad b = \mathtt{\textbf{(}}\mathscr{T}\llbracket e_1 \rrbracket \rho\, \varepsilon',\, \mathtt{\textbf{touch}}\; y\mathtt{\textbf{)}} \\
 & & \qquad \varepsilon' = \varepsilon \cup \mathrm{Fu}(\rho,\, e_1) \cup \mathrm{Fu}(\rho,\, e_2) \\
\mathscr{T}\llbracket \mathtt{\textbf{plet}}\; p \;=\; e_1\; \mathtt{\textbf{in}}\; e_2 \rrbracket \rho\, \varepsilon & = & \mathtt{\textbf{let}}\; y \;=\; \mathtt{\textbf{future}}\; \mathscr{T}\llbracket e_1 \rrbracket \rho\, \varepsilon\; \mathtt{\textbf{in}}\; \mathscr{T}\llbracket e_2 \rrbracket \rho'\, \varepsilon' \\
 & & \quad \text{where } y \text{ is fresh} \\
 & & \qquad \rho' = \rho \pm \{x \mapsto (y, p) \mid x \in \mathrm{Vars}(p)\} \\
 & & \qquad \varepsilon' = \varepsilon \cup \mathrm{Fu}(\rho,\, e_1) \\
\mathscr{T}\llbracket \mathtt{\textbf{pcase}}\; e_1\; \mathtt{\textbf{\&}}\; e_2\; \mathtt{\textbf{of}}\; m_1 \mid \cdots \mid m_k \rrbracket \rho\, \varepsilon & = & \mathtt{\textbf{pcase}}\; \mathscr{T}\llbracket e_1 \rrbracket \rho\, \varepsilon\; \mathtt{\textbf{\&}}\; \mathscr{T}\llbracket e_2 \rrbracket \rho\, \varepsilon \\
 & & \quad \mathtt{\textbf{of}}\; \mathscr{T}\llbracket m_1 \rrbracket \rho_1\, \varepsilon' \mid \cdots \mid \mathscr{T}\llbracket m_k \rrbracket \rho_k\, \varepsilon' \\
 & & \quad \text{where } \varepsilon' = \varepsilon \cup \mathrm{Fu}(\rho,\, e_1) \cup \mathrm{Fu}(\rho,\, e_2) \\
 & & \\
\mathscr{T}\llbracket \pi_1\; \mathtt{\textbf{\&}}\; \pi_2 \Rightarrow e \rrbracket \rho\, \varepsilon & = & \pi_1\; \mathtt{\textbf{\&}}\; \pi_2 \Rightarrow \mathscr{C}\llbracket \mathscr{T}\llbracket e \rrbracket \rho'\, \varepsilon \rrbracket X \\
 & & \quad \text{where } C = \mathrm{dom}(\rho) \setminus \mathrm{FV}(\pi_1\; \mathtt{\textbf{\&}}\; \pi_2 \Rightarrow e) \\
 & & \qquad \rho' = \rho \setminus C \\
 & & \qquad X = \{y \mid \exists x \in C \wedge \rho(x) = (y, p)\} \setminus \varepsilon \\
 & & \\
\mathscr{C}\llbracket e \rrbracket X & = & \mathtt{\textbf{cancel}}\; x\; \mathtt{\textbf{in}}\; \mathscr{C}\llbracket e \rrbracket (X \setminus \{x\}) \qquad \text{where } x \in X \\
\mathscr{C}\llbracket e \rrbracket \emptyset & = & e
\end{array}
$$

Fig. 15. Translation of parallel tuples and bindings.

they have either escaped or been touched. Maintaining the map ($\rho$) and the set ($\varepsilon$) constitutes an analysis that identifies futures that may be canceled in a particular expression, information that is used in inserting cancelations at the appropriate points. This is what makes the parallel binding form speculative.

$$\begin{array}{rcl}
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(\mathbf{fn}\, x \Rightarrow e) & = & \mathrm{FV}(e) \setminus \{x\} \\
\mathrm{FV}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(e_3) \\
\mathrm{FV}(\mathbf{(|}\, e_1, e_2 \,\mathbf{|)}) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\
\mathrm{FV}(\mathbf{plet}\ p\ =\ e_1\ \mathbf{in}\ e_2) & = & \mathrm{FV}(e_1) \cup (\mathrm{FV}(e_2) \setminus \mathrm{Vars}(p)) \\
\mathrm{FV}(\mathbf{pcase}\ e_1\ \mathbf{\&}\ e_2\ \mathbf{of}\ m_1 \mid \cdots \mid m_k) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \cup \mathrm{FV}(m_1) \cup \cdots \cup \mathrm{FV}(m_k) \\
 & & \\
\mathrm{FV}(\pi_1\ \mathbf{\&}\ \pi_2 \Rightarrow e) & = & \mathrm{FV}(e) \setminus (\mathrm{Vars}(\pi_1) \cup \mathrm{Vars}(\pi_2))
\end{array}$$

Fig. 16. Free variables.

We use the notation $\rho \setminus X$ to remove the variables in $X$ from the domain of $\rho$, i.e.,

$$\rho \setminus X = \{y \mapsto \rho(y) \mid y \in (\mathrm{dom}(\rho) \setminus X)\}$$

We define the futures of a source language expression with respect to a given environment by

$$\mathrm{Fu}(\rho, e) = \{y \mid \exists x \in \mathrm{FV}(e) \text{ such that } \rho(x) = (y, p)\}$$

where $\mathrm{FV}(e)$ are the *free variables* of $e$ (see Figure 16).

The mapping $\rho$ records which variables require touches and, if so, which futures to touch. The translation of variables (the first rule) consults $\rho$ to determine if it needs to touch a future to get its value; if so, it inserts the appropriate pattern-matching code in a let binding at the variable site.

The rule translating parallel tuples (pairs) creates a future for the second subexpression only, which is sufficient for computing the two subexpressions in parallel. A fresh variable $y$ is created for this future, but it need not be added to the map $\rho$ for tracking later, since it is immediately touched in the expression $b$. It adds the futures in $e_1$ and $e_2$ into $\varepsilon$, so they will not be canceled in either subexpression. The rule translating parallel bindings also introduces a fresh variable $y$ to name the future. By contrast with the parallel tuple rule, the map $\rho$ must be supplemented with this new future variable and its corresponding pattern, so it may be considered for cancelation in any **pcase** arm where it is definitely not needed.

The rule translating **pcase** manages the cancelation of unneeded futures on the right-hand sides of its matches. The rules on matches ($\pi_1$ **&** $\pi_2 \Rightarrow e$) consult $\rho$ for futures that are candidates for cancelation, taking care to exclude the futures tracked by $\varepsilon$ in the definition of $X$. The actual cancelations are inserted by the $\mathscr{C}[\![]\!]$ rules at the bottom of the figure.

The rules omitted from Figure 15 are generally either straightforward recursive applications to all subexpressions or natural modifications of the presented rules. For function application, for example, translations are simply recursive translations of the function subexpression and the argument expression. For **if** expressions, one can either construct translation rules by mimicking the **pcase** rules or translate **if**s into **pcase**s as a preliminary step and use the **pcase** rules. In summary, we have given all the interesting details of the translation, and no undue difficulty should arise in extending it to standard other forms.

### *8.4 Translation of Parallel Case*

At this stage in the translation, only parallel cases remain to be translated among PML's implicitly-threaded constructs. Our presentation switches here to ML syntax, as it is more readily capable of expressing the generated code.

The key to the efficient implementation of the **pcase** expression is tracking the state of the subcomputations. We use an approach that is inspired by Le Fessant and Maranget's technique for compiling join patterns (1998). The basic idea is to use a finite-state machine to track the state of the subcomputations of the **pcase**. When a subcomputation terminates, either with a value or an exception, it invokes a state-transition function, which updates the state to reflect the completion of the subcomputation. Once an *accepting* state is reached, the remaining subcomputations can be canceled and the result of the **pcase** can be computed.

To give a precise description of the compilation technique, we need to define some notation. We assume that we are translating a **pcase** of the form

$$\begin{array}{l}\textbf{pcase}\ e_1\ \texttt{\&}\ \cdots\ \texttt{\&}\ e_n \\ \quad \textbf{of}\ \pi_{1,1}\ \texttt{\&}\ \cdots\ \texttt{\&}\ \pi_{1,n}\ \texttt{=>}\ f_1 \\ \quad\ \ |\ \ldots \\ \quad\ \ |\ \pi_{m,1}\ \texttt{\&}\ \cdots\ \texttt{\&}\ \pi_{m,n}\ \texttt{=>}\ f_m\end{array}$$

where $n$ is the number of subcomputations, $m$ is the number of rules, the $e_k$ and $f_j$ are expressions that have already been translated, and the $\pi_{j,k}$ are parallel patterns. (Please note this $n$-subcomputation construct differs from the simplified two-subcomputation model in Figures 12 and 14.) We use $\Pi_j$ to denote the left-hand-side of the $j$th rule of the **pcase** (i.e., $\pi_{j,1}$ & $\cdots$ & $\pi_{j,n}$) and we use $BV(\Pi_j)$ to denote the variables that are bound by the patterns $\pi_{j,1}, \ldots, \pi_{j,n}$. The state machine for this **pcase** will have $2^n$ states.[5] We identify these states by bit strings of length $n$, where the $k$th bit is 1 if the subcomputation $e_k$ has terminated. If $\vec{B}$ is a bit string, then we use $\vec{B}[k]$ to denote its $k$th bit. We use $\vec{X}_k$ to denote the bit string with the $k$th bit set and all other bits unset. The operator $\wedge$ denotes bitwise anding of bit strings. We use $S_i$ to denote the $i$th state ($0 \leqslant i < 2^n$), with $S_0$ being the start state, and $\vec{S}_i$ to denote its bit string. We define the *next states* following the completion of the $k$th subcomputation by

$$Next_k = \{S_i \mid \vec{S}_i[k] = 1\}$$

Lastly, if $\Pi = \pi_1$ & $\cdots$ & $\pi_n$ and $\vec{B}$ is an $n$-bit string with $m$ bits set, then

$$\Pi|_{\vec{B}} = (p_{k_1}, \ldots, p_{k_m}) \qquad \text{where } p_k = \begin{cases} \_ & \text{if } \pi_k = \text{?} \\ \texttt{SOME}(\pi_k) & \text{otherwise} \end{cases}$$

where $1 \leqslant k_1 < \cdots < k_m \leqslant n$ and $B[k_l] = 1$ for $1 \leqslant l \leqslant m$. For example, if

$$\Pi = p_1\ \texttt{\&}\ p_2\ \texttt{\&}\ \text{?}\ \texttt{\&}\ p_4$$

then

$$\Pi|_{\texttt{1011}} = (\texttt{SOME}(p_1), \_, \texttt{SOME}(p_4))$$

[5] Sometimes certain states are unreachable and can be removed.

The first step of the compilation process involves identifying which cases (i.e., rows of the **pcase**) can be matched in which states. For each row $j$, we define an $n$-bit string $\vec{P}_j$ as follows:

$$\vec{P}_j[k] = \begin{cases} 0 & \text{if } \pi_{j,k} \text{ is ?} \\ 1 & \text{otherwise} \end{cases}$$

The $j$th rule of the **pcase** is applicable to the $i$th state if there is a value for each of the non-? patterns. This property is captured in the following definition of the *applicable rules* for $S_i$:

$$Rules(S_i) = \{(j, \Pi_j|_{\vec{S}_i}) \mid \vec{P}_j \wedge \vec{S}_i = \vec{P}_j\}$$

For a state $S_i$, the results that are available are defined by

$$Avail(S_i) = \{k \mid S_i[k] = 1\}$$

For the $j$th rule, we define the *don't care* subcomputations to be

$$DC_j = \{k \mid \vec{P}_j[k] = 0\}$$

These are the subcomputations that are not required to fire the rule.

Once we have partitioned the patterns according to the applicable rules for each state, we can generate the code that implements the state machine. We first check to see if $Rules(S_0)$ is not empty, then the **pcase** has a rule that is applicable before any subcomputation is started and we can just generate code to invoke the action of the first rule of $Rules(S_0)$. The more typical (and interesting) case is when $Rules(S_0) = \emptyset$ and we must generate the code that implements the state machine. The resulting code has the structure shown in Figure 17, where the lines of interest have been labeled on the left-hand side. Each framed code fragment represents an indexed sequence of declarations, expressions, etc. (e.g., the line labeled *(3)* represents the $n$ declarations of the variables $r_1, \ldots, r_n$). Each of the labeled lines is defined as follows:

1. The translation of **pcase** makes use of a `pcaseWrapper` function, which abstracts the control-flow mechanisms that are used to manage returning from **pcase**. The majority of the translation of **pcase** is wrapped in an anonymous function, which is passed to the `pcaseWrapper` function. The `return` argument is a function that wraps the normal return continuation, while the `exnReturn` argument is a function that wraps the exception-handler continuation.
2. The current state of the state machine is stored in an m-variable. We rely on the synchronous behavior of this variable to guarantee the atomicity of the state-machine transitions.
3. We allocate a reference cell ($r_i$) to hold the result of each subcomputation; these cells are initialized to `NONE`.
4. We allocate a cancelable object ($c_i$) for each subcomputation, which is used to cancel the subcomputation when its result is no longer required.
5. Each subcomputation has an associated transition function ($trans_i$) that is called when the subcomputation yields a result (see 9). The transition function takes the current state from the `state` m-variable (thus ensuring atomicity)

```
(1)pcaseWrapper (
      fn (return, exnReturn) => let
(2)     val state = MVar.new $S_0$
        fun resume st = (MVar.put(state, st); dispatch())
(3)     $\boxed{\texttt{val } r_k \texttt{ = ref NONE}}_{k=1}^{n}$
(4)     $\boxed{\texttt{val } c_k \texttt{ = Cancel.new ()}}_{k=1}^{n}$
(5)     $\Big[$fun $trans_k$ v = let
              val st = MVar.take state
              in
                $r_k$ := SOME v;
                case st $\vee$ $\vec{X}_k$ of $\boxed{S_i \texttt{ => } state_{S_i}\texttt{()}}_{S_i \in Next_k}$
              end$\Big]_{k=1}^{n}$
(6)     $\Big[$and $state_{S_i}$() = (
              case ($\boxed{\texttt{!}r_k}_{k \in Avail(S_i)}$)
               of $\boxed{\Pi|_{S_i} \texttt{ => } act_j \texttt{ (}\boxed{a}_{a \in BV(\Pi)}\texttt{)}}_{(j,\Pi) \in Rules(S_i)}$
                | _ => resume $S_i$)$\Big]_{i=1}^{2^n-1}$
(7)     $\Big[$and $act_j$($\boxed{a}_{a \in BV(\Pi_j)}$) = (
              $\boxed{\texttt{Cancel.cancel } c_k\texttt{;}}_{k \in DC(\Pi_j)}$
              return($f_j$ handle ex => reraise ex))$\Big]_{j=1}^{m}$
(8)     and reraise x = (
            $\boxed{\texttt{Cancel.cancel } c_k\texttt{;}}_{k=1}^{n}$
            exnReturn x)
        and actExn ex = (MVar.take st; reraise ex)          in
(9)       $\Big[$Cancel.spawn ($c_i$,
            fn () => $trans_k$($e_k$ handle ex => actExn ex));$\Big]_{k=1}^{n}$
(10)      dispatch ()
        end)
```

Fig. 17. The template for translating parallel case (when $Rules(S_0) = \emptyset$).

and records the subcomputation's result in the $r_i$ reference. It then calls the state function corresponding to the next state.

6. Each state (other than $S_0$) has a corresponding state function that tests the applicable rules of the state against the available results. If no applicable rule matches the available results, then the new state is put into the `state` m-variable and the calling thread is terminated as a result of calling `dispatch`. The function `dispatch` consists of scheduling code that picks the next thread to execute on the calling processor.
7. The right-hand-side expression of each rule in the **`pcase`** is lifted into an *action function* that takes the variables bound by the patterns of the left-hand-side of the rule as parameters. This function includes cancelation code for any subcomputations that might still be running (i.e., any subcomputations that are matched by `?` on the left-hand-side). By putting the right-hand-side

expressions in action functions, we avoid code duplication when two states share the same applicable rule.

8. We define two functions, `reraise` and `exnAct`, to help with the propagation of exceptions. The `exnAct` function is is defined for when a subcomputation raises an exception; it first takes the state variable to guarantee that only one exception/result is returned from the **`pcase`** and then calls the `reraise` function to cancel the remaining subcomputations and return the exception. The `reraise` function is also used when an exception is raised by one of the rule right-hand sides. In that case, the state has already been taken.
9. Code to spawn the subcomputations. For each **`pcase`** argument expression, we spawn a cancelable subcomputation that passes the result of the expression to the corresponding transition function. If the expression raises an exception, it is passed to the `actExn` function.
10. After spawning the subcomputations, the processor that was running the **`pcase`** setup code is available for other work, so we call the `dispatch` function to schedule some other thread.

To illustrate how this translation works in practice, recall the implementation of `trProd` from Figure 6, which has a **`pcase`** consisting of two subcomputations and three rules. Applying the translation method described above to `trProd` results in the code shown in Figure 18. There are two transition functions (`trans1` and `trans2`) corresponding to the two subcomputations (*cf.*, item 5 above), three state functions (`state01`, `state10`, and `state11`) (*cf.*, item 6 above), and three action functions (`act1`, `act2`, and `act3`) (*cf.*, item 7 above).

## 8.5 Scheduling

This section gives an overview of the issues of scheduling Manticore's implicitly threaded constructs. In implicit threading, the scheduling policy determines the order in which implicit threads are evaluated and the mapping from implicit threads to processors. Here, we focus on dynamic scheduling policies where the schedule is determined as the program executes. This is opposed to static scheduling policies where the schedule is determined before the program executes. Dynamic scheduling policies are necessary for programs such as `trProd` wherein the implicit-thread graph varies with the input data set.

Implicit threading encourages the programmer to divide the program into small implicit threads because the computations arising from such programs give the scheduler the most flexibility in its goal to distribute subcomputations evenly across processors. Because there is a scheduling cost associated with spawning each implicit thread, the total scheduling costs of such programs can be high. Indeed, if scheduling costs are too high, then the benefit of parallelism is lost altogether. There are many approaches to address this problem, but, broadly speaking, a given approach falls into one of two categories, either *work sharing* or *work stealing*. In work sharing, each processor plays an active role in distributing implicit threads among processors. Whenever implicit threads are spawned by a processor, the scheduler migrates some of them to other processors with the aim of spreading work to idle processors. Work

```
fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = pcaseWrapper (
      fn (return, exnReturn) => let
        val state = MVar.new 0
        fun resume st = (MVar.put(state, st); dispatch())
        val r1 = ref NONE
        val r2 = ref NONE
        val c1 = Cancel.new ()
        val c2 = Cancel.new ()
        fun trans1 v = let
              val st = MVar.take state
              in
                r1 := SOME v;
                case Int.andb(st, 2)
                 of 0 => state10()
                  | 1 => state11()
              end
        and trans2 v = let
              val st = MVar.take state
              in
                r2 := SOME v;
                case Int.andb(st, 1)
                 of 0 => state01()
                  | 2 => state11()
              end
        and state01 () = (case !r2
               of SOME 0 => act2()
                | _ => resume 1)
        and state10 () = (case !r1
               of SOME 0 => act1()
                | _ => resume 2)
        and state11 () = (case (!r1, !r2)
               of (SOME 0, _) => act1()
                | (_, SOME 0) => act2()
                | (SOME pL, SOME pR) => act3(pL, pR))
        and act1 () = (Cancel.cancel c2; return 0)
        and act2 () = (Cancel.cancel c1; return 0)
        and act3 (pL, pR) = return(pL * pR)
        and actExn ex = (
              Cancel.cancel c1; Cancel.cancel c2;
              exnReturn ex)
        in
          Cancel.spawn (c1,
            fn () => trans1 (treeProd tL) handle ex => actExn ex);
          Cancel.spawn (c2,
            fn () => trans2 (treeProd tR) handle ex => actExn ex);
          dispatch ()
        end)
```

Fig. 18. The translation of `trProd` from Figure 6.

stealing takes the dual approach, making idle processors responsible for migrating implicit threads away from busy processors.

Thread migration has a high scheduling cost because each migration involves communication between processors. Consequently, the total number of migrations is a key metric for comparing scheduling policies. Work stealing has the advantage of minimizing the number of migrations. In work stealing, migrations occur only when a processor goes idle, whereas in work sharing, a migration occurs each time a thread is spawned. There is a large body of work analyzing the performance of work stealing (Mohr *et al.* 1990; Feeley 1993; Arora *et al.* 1998; Frigo *et al.* 1998; Blumofe & Leiserson 1999; Acar *et al.* 2000; Spoonhower *et al.* 2008) that shows, in particular, that the total number of migrations is low in theory and practice.

We address the design of efficient work stealing for Manticore in another paper (Fluet *et al.* 2008b). One issue that we consider is how to reduce scheduling costs of spawning individual threads. Our work shows how to adapt *clone compilation*, a technique pioneered in the implementation of the Cilk-5 language (Frigo *et al.* 1998), to Manticore to reduce scheduling costs associated with parallel tuples and parallel value bindings. We also investigate *lazy promotion*, an approach that integrates work stealing with Manticore's parallel memory management system (Rainey 2010).

We also consider how, in the case of data-parallel computations, we can avoid the cost of spawning many small implicit threads by grouping them dynamically into fewer large implicit threads. Recent studies demonstrate that techniques which determine these groupings statically are not effective in general because in some cases, they inevitably overestimate or underestimate the group sizes, consequently leaving some processors idle or degrading performance (Bergstrom *et al.* 2010; Tzannes *et al.* 2010). Our *lazy tree splitting* is an approach based on work stealing that determines groupings dynamically (Bergstrom *et al.* 2010). In lazy tree splitting, we process a given data-parallel computation (e.g., `mapP f xs`) by the following recursive process. Before each subcomputation $c_i$ (e.g., `f x` where `x` is an element of `xs`), we check for idle processors. If we detect zero idle processors, we process $c_i$ and proceed to the next subcomputation (or stop if there are no remaining subcomputations). Otherwise, we split the unprocessed subcomputations in half, spawn an implicit thread that will recursively process the second half, and resume where we left off. In the common case where we do not split, the scheduling cost consists of just one local memory access and a conditional branch.

In another paper, we investigate the mechanism for asynchronously canceling the evaluation futures (Fluet *et al.* 2008b). All of the work summarized in this section is contained in Rainey's dissertation (Rainey 2010).

## 9 Related work

Manticore's support for fine-grained parallelism is influenced by previous work on nested data-parallel languages, such as Nesl (Blelloch *et al.* 1994; Blelloch 1996; Blelloch & Greiner 1996) and Nepal/DPH (Chakravarty & Keller 2000; Chakravarty *et al.* 2001; Leshchinskiy *et al.* 2006). Like PML, these languages have functional sequential cores and parallel arrays and comprehensions. To this mix, PML adds explicit parallelism, which neither NESL or DPH supports; neither does NESL or DPH have any analogs to our other mechanisms—parallel tuples, bindings, and cases. The NESL and DPH research has been directed largely at the topic of *flattening*, an internal compiler transformation which can yield great benefits in the processing of parallel arrays. Our PML compiler does not yet implement flattening, although we expect to devote great attention to the topic as our work moves forward.

Multilisp (Halstead 1984, 1985) is a version of SCHEME extended with a `pcall` construct and parallel futures. Like the parallel-tuple construct, `pcall` offers fork-join parallelism. Multilisp's parallel futures bear a resemblance to the futures we use in Section 8, except that, in Multilisp, a future is touched implicitly whenever that future is passed to a strict operation, whereas our futures must be touched explicitly. Osborne extended Multilisp with support for speculative computation via "parallel or" and "parallel and" operators (Osborne 1990). Osborne's "parallel or" and "parallel and" are not logical operators, as their names might suggest, but control constructs operating on collections of simultaneously evaluating computations. "Parallel or" returns the first nonnil value to finish evaluating, or nil if none exists; "parallel and" returns true if all its computations are nonnil or nil otherwise. Each of these constructs can be encoded by PML's **`pcase`** in a straightforward way (for any particular number of elements).

GpH is a dialect of Haskell in which parallelism is expressed through evaluation strategies (Trinder *et al.* 1998). An evaluation strategy is an expression that determines various properties of the runtime behavior of computations. For example, one such evaluation strategy specifies that three given expressions are allowed to evaluate in parallel, and another evaluation strategy specifies that a recursive function stops spawning parallel threads below a certain recursion depth. Because evaluation strategies are first-class values, it is possible to define functions that are parameterized over evaluation strategies. This ability enables programs where the details of what is to be computed are separate from how the computation is to be performed. PML does not provide evaluation strategies directly, but there is a similar approach called first-class monadic schedules (FCMS) (Mirani & Hudak 1995, 2004) that can be readily encoded on top of **`pval`**s. FCMS offers some additional powerful features, such as the ability to define processor topologies and to determine on which processor each thread executes.

The languages Id (Nikhil 1991), pH (Nikhil & Arvind 2001), and Sisal (Gaudiot *et al.* 1997) represent another approach to implicit parallelism in a functional setting that does not require user annotations. The explicit concurrency mechanisms in PML are taken from CML (Reppy 1999). While CML was not designed with parallelism in mind (in fact, its original implementation is inherently not parallel), we believe that it will provide good support for coarse-grained parallelism. Erlang is a similar language that has a mutation-free sequential core with message passing (Armstrong *et al.* 1996) that has parallel implementations (Hedqvist 1998) but no support for fine-grained parallel computation.

The Cilk programming language (Blumofe *et al.* 1995) is an extension of C with additional constructs for expressing parallelism. Cilk is an imperative language, and, as such, its semantics is different from that of PMLin some obvious ways. Some procedures in Cilk are modified with the **cilk** keyword; those are *Cilk procedures*. Cilk procedures call other Cilk procedures with the use of **spawn**. A spawned procedure starts running in parallel, and its parent procedure continues execution. In this way, spawned Cilk procedures are similar to PML expressions bound with **pval**. Cilk also includes a sophisticated abort mechanism for cancelation of spawned siblings; we have suggested some encodings of similar parallel patterns in Section 7.

SplitC (Krishnamurthy *et al.* 1993) is an extension of C that is intended for programming distributed memory multiprocessor machines. Like Cilk (but unlike PML), SplitC offers imperative features such as in-place updates in memory and pointers. But unlike Cilk and PML, SplitC offers a distributed memory model in which each processor has a local memory and the programmer may control on which processor a given piece of memory is stored. A language similar to SplitC could be used as a compilation target for PML, though we have implemented a different one and reported on it elsewhere (Fluet *et al.* 2008b).

Accelerator (Tarditi *et al.* 2006) is an imperative data-parallel language that allows programmers to utilize GPUs for general-purpose computation. The operations available in Accelerator are similar to those provided by DPH's or PML's parallel arrays and comprehensions, except destructive update is a central mechanism. In keeping with the hardware for which it is targeted, Accelerator is directed toward regular, massively parallel operations on homogeneous collections of data, in marked contrast to the example presented in Section 7.

Programming parallel hardware effectively is difficult, but there have been some important recent achievements. Google's MapReduce programming model (Dean & Ghemawat 2004) has been a success in processing large data sets in parallel. Sawzall, another Google project, is a system for analysis of large data sets distributed over disks or machines (Pike *et al.* 2005). (It is built on top of the aforementioned MapReduce system.) Brook for GPUs (Buck *et al.* 2004) is a C-like language that allows the programmer to use a GPU as a stream coprocessor.

## 10 Conclusion

PML is a heterogeneous parallel functional language. In this paper, we have described its implicitly threaded constructs, which support fine-grained task and data-parallel computations. These include standard implicitly threaded mechanisms such as parallel tuples and nested-parallel arrays, as well as novel ones such as parallel bindings and nondeterministic parallel cases. We have illustrated our language with a number of examples and given a formal description of their implementation in the Manticore system. We have been working on a prototype implementation of the Manticore system since January 2007. Using it, we have successfully exercised our constructs on a variety of canonical test applications, including standard sorting and complex hull algorithms, Barnes-Hut *n*-body simulations, and ray tracing, and our results, detailed elsewhere (Bergstrom *et al.* 2010), have been very promising. Our implementation is largely feature complete and is available to the interested reader on request.

## *Acknowledgments*

We would like to thank the anonymous referees and editor for their helpful comments and suggestions. This work is supported in part by National Science Foundation Grants CCF-0811389, CCF-0811419, and CCF-1010568. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

## References

Acar, U. A., Blelloch, G. E. & Blumofe, R. D. (2000) The data locality of work stealing. In *Proceedings of the 12th ACM Annual Symposium on Parallel Algorithms and Architectures.* ACM, pp. 1–12.

Armstrong, J., Virding, R., Wikström, C. & Williams, M. (1996) *Concurrent Programming in ERLANG*, 2nd ed. Hertfordshire, UK: Prentice Hall International.

Arora, N. S., Blumofe, R. D. & Plaxton, C. G. (1998) *Thread Scheduling for Multiprogrammed Multiprocessors.*

Barth, P., Nikhil, R. S. & Arvind. (1991) M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture (fpca '91) New York, NY.* Lecture Notes in Computer Science, vol. 523. Springer-Verlag, pp. 538–568.

Barton, R., Adkins, D., Prokop, H., Frigo, M., Joerg, C., Renard, M., Dailey, D. & Leiserson, C. (1998) *Cilk Pousse.* Viewed on March 20, 2008 at 2:45 PM.

Bergstrom, L., Fluet, M., Rainey, M., Reppy, J. & Shaw, A. (2010) Lazy tree splitting. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, New York, NY.* ACM, pp. 93–104.

Blelloch, G. E. (1996) Programming parallel algorithms, *Commun. ACM*, 39 (3): 85–97.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J. & Zagha, M. (1994) Implementation of a portable nested data-parallel language, *J. Parallel Distrib. Comput.*, 21 (1): 4–14.

Blelloch, G. E. & Greiner, J. (1996) A provable time and space efficient implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, New York, NY.* ACM, pp. 213–225.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. & Zhou, Y. (1995) Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, New York, NY.* ACM, pp. 207–216.

Blumofe, R. D. & Leiserson, C. E. (1999) Scheduling multithreaded computations by work stealing, *J. ACM*, 46 (5): 720–748.

Boehm, H.-J., Atkinson, R. & Plass, M. (1995) Ropes: An alternative to strings, *Software Pract. Ex.*, 25 (12): 1315–1330.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. & Hanrahan, P. (2004) Brook for GPUs: Stream computing on graphics hardware, *Proc. ACM SIGGRAPH 2004*, 23 (3): 777–786.

Chakravarty, M. M. T. & Keller, G. (2000) More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, New York, NY.* ACM, pp. 94–105.

Chakravarty, M. M. T., Keller, G., Leshchinskiy, R. & Pfannenstiel, W. (2001) Nepal—Nested data parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference on Parallel Computing, New York, NY*. Lecture Notes in Computer Science, vol. 2150. Springer-Verlag, pp. 524–534.

Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G. & Marlow, S. (2007) Data parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, New York, NY*. ACM, pp. 10–18.

Dailey, D. & Leiserson, C. E. (2002) Using Cilk to write multiprocessor chess programs, *J. Int. Comput. Chess Assoc.*, 24 (4): 236–237.

Danaher, J. S., Lee, I.-T. A. & Leiserson, C. E. (2006) Programming with Exceptions in JCilk, *Sci. Comput. Program.*, 63 (2): 147–171.

Dean, J. & Ghemawat, S. (December 2004) MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04), Berkely, CA*. USENIX, pp. 137–150.

Feeley, M. (1993) *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, Waltham, MA, USA.

Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007a) Status report: The Manticore project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML, New York, NY*. ACM, pp. 15–24.

Fluet, M., Rainey, M. & Reppy, J. (2008a) A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, New York, NY*. ACM, pp. 241–252.

Fluet, M., Rainey, M., Reppy, J. & Shaw, A. (2008b) Implicitly-threaded parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, New York, NY*. ACM, pp. 119–130.

Fluet, M., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007b) Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, New York, NY*. ACM, pp. 37–44.

Frigo, M., Leiserson, C. E. & Randall, K. H. (June 1998) The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98), New York, NY*. ACM, pp. 212–223.

Gansner, E. R. & Reppy, J. H. (eds). (2004) *The Standard ML Basis Library*. Cambridge, England: Cambridge University Press.

Gaudiot, J.-L., DeBoni, T., Feo, J., Bohm, W., Najjar, W. & Miller, P. (1997) The Sisal model of functional programming and its implementation. In *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms/Architecture Synthesis (pAs '97), Los Alamitos, CA*. IEEE Computer Society, pp. 112–123.

GHC. (November 1998) *The Glasgow Haskell Compiler* [online]. Available at: `http://www.haskell.org/ghc` Accessed 10 December 2010.

Halstead R. H., Jr. (1984) Implementation of multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming. New York, NY*. ACM, pp. 9–17.

Halstead R. H., Jr. (1985) Multilisp: A language for concurrent and symbolic computation, *ACM Trans. Program. Lang. Syst.*, 7: 501–538.

Hammond, K. (1991) *Parallel SML: A Functional Language and its Implementation in Dactl*. Cambridge, MA: MIT Press.

Harris, T., Marlow, S., Peyton Jones, S. & Herlihy, M. (2005) Composable memory transactions. In *Proceedings of the 2005 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, New York, NY*. ACM, pp. 48–60.

Hauser, C., Jacobi, C., Theimer, M., Welch, B. & Weiser, M. (December 1993). Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, pp. 94–105.

Hedqvist, P. (June 1998) *A Parallel and Multithreaded ERLANG Implementation*. MPhil thesis, Computer Science Department, Uppsala University, Uppsala, Sweden.

Jones, M. P. & Hudak, P. (August 1993) *Implicit and Explicit Parallel Programming in Haskell*. Tech. Rep. YALEU/DCS/RR-982, Yale University.

Krishnamurthy, A., Culler, D. E., Dusseau, A., Goldstein, S. C., Lumetta, S., von Eicken, T. & Yelick, K. (1993) Parallel programming in split-C. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, New York, NY*. ACM, pp. 262–273.

Le Fessant, F. & Maranget, L. (1998) Compiling join-patterns. In *Proceedings of the Third International Workshop on High-Level Concurrent Languages (HLCL '98)*. Electronic Notes in Theoretical Computer Science, vol. 16, no. 3. Elsevier Science, pp. 205–224.

Leroy, X. & Pessaux, F. (2000) Type-based analysis of uncaught exceptions, *ACM Trans. Program. Lang. Syst*., 22 (2): 340–377.

Leshchinskiy, R., Chakravarty, M. M. T. & Keller, G. (2006) Higher order flattening. In *International Conference on Computational Science (ICCS '06)*, Alexandrov, V., van Albada, D., Sloot, P. & Dongarra, J. (eds), LNCS, no. 3992. New York, NY. Springer-Verlag, pp. 920–928.

Mandel, L. & Maranget, L. (December 2008). *The JoCaml Language Release 3.11 Documentation and User's Manual* [online]. Available at: `http://jocaml.inria.fr/manual/index.html` Accessed 10 December 2010.

McCarthy, J. (1963) A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, Braffort, P. & Hirschberg, D. (eds), North-Holland, Amsterdam, pp. 33–70.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1997) *The Definition of Standard ML (revised)*. Cambridge, MA: MIT Press.

Mirani, R. & Hudak, P. (1995) First-class schedules and virtual maps. In *Fpca '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, New York, NY*. ACM, pp. 78–85.

Mirani, R. & Hudak, P. (2004) First-class monadic schedules, *ACM Trans. Program. Lang. Syst*., 26 (4): 609–651.

Mohr, E., Kranz, D. A. & Halstead R. H., Jr. (1990) Lazy task creation: A technique for increasing the granularity of parallel programs. In *Conference Record of the 1990 ACM Conference on Lisp and Functional Programming. New York, NY*. ACM, pp. 185–197.

Nikhil, R. S. (July 1991). *ID Language Reference Manual*. Cambridge, MA: Laboratory for Computer Science, MIT.

Nikhil, R. S. & Arvind. (2001) *Implicit Parallel Programming in pH*. San Francisco, CA: Morgan Kaufmann.

Osborne, R. B. (1990) Speculative computation in multilisp. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming. New York, NY*. ACM, pp. 198–208.

Peyton Jones, S., Gordon, A. & Finne, S. (1996) Concurrent Haskell. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages (popl '96). New York, NY*. ACM, pp. 295–308.

Peyton Jones, S., Reid, A., Henderson, F., Hoare, T. & Marlow, S. (1999) A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), New York, NY*. ACM, pp. 25–36.

Pike, R., Dorward, S., Griesemer, R. & Quinlan, S. (2005) Interpreting the data: Parallel analysis with sawzall, *Sci. Program. J*., 13 (4): 227–298.

Rainey, M. (August 2010) *Effective Scheduling Techniques for High-Level Parallel Programming Languages* [online]. PhD thesis, University of Chicago. Available at: `http://manticore.cs.uchicago.edu` Accessed 10 December 2010.

Reppy, J., Russo, C. & Xiao, Y. (2009) Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, New York, NY*. ACM, pp. 257–268.

Reppy, J. H. (1991) CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91), New York, NY*. ACM, pp. 293–305.

Reppy, J. H. (1999) *Concurrent Programming in ML*. Cambridge, England: Cambridge University Press.

Shaw, A. (July 2007). *Data Parallelism in Manticore* [online]. MPhil thesis, University of Chicago. Available at: `http://manticore.cs.uchicago.edu` Accessed 10 December 2010.

Spoonhower, D., Blelloch, G. E., Gibbons, P. B. & Harper, R. (2008) Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the 20th ACM Annual Symposium on Parallel Algorithms and Architectures, New York, NY*. ACM.

Tarditi, D., Puri, S. & Oglesby, J. (2006) Accelerator: Using data parallelism to program GPUs for general-purpose uses, *Sigops Pper. Syst. Rev.*, 40 (5): 325–335.

Trinder, P. W., Hammond, K., Loidl, H.-W. & Peyton Jones, S. L. (1998) Algorithm + strategy = parallelism, *J. Funct. Program.*, 8 (1): 23–60.

Tzannes, A., Caragea, G. C., Barua, R. & Vishkin, U. (2010) Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, New York, NY*. ACM, pp. 179–190.

Yi, K. (1998) An abstract interpretation for estimating uncaught exceptions in Standard ML programs, *Sci. Comput. Program.*, 31 (1): 147–173.