

Implicitly-threaded Parallelism in Manticore

MATTHEW FLUET

*Toyota Technological Institute at Chicago
fluet@tti-c.org*

MIKE RAINEY

*University of Chicago
mrainey@cs.uchicago.edu*

JOHN REPPY

*University of Chicago
jhr@cs.uchicago.edu*

ADAM SHAW

*University of Chicago
ams@cs.uchicago.edu*

Abstract

The increasing availability of commodity multicore processors is bringing parallel computing to the masses. 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. In this paper, we present PML (named “Manticore” in prior publications), a high-level functional language for building parallel applications on commodity multicore hardware including a diverse collection of parallel constructs for different granularities of work. In this paper, we focus on the implicitly-threaded parallel constructs in our language. We concentrate on those elements that distinguish our design from related ones, namely, a novel parallel binding form, a nondeterministic parallel case form, and exceptions in the presence of data parallelism. These features differentiate the present work from related work on functional data parallel language designs, which has focused largely on parallel problems with regular structure and the compiler transformations — most notably, flattening — that make such designs feasible. We also describe our implementation strategies and present some detailed examples utilizing various mechanisms of our language.

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? 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 parallel 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 data parallelism (Chakravarty *et al.*, 2007), inspired by NESL (Blelloch *et al.*, 1994; Blelloch, 1996).

The Manticore Project (Fluet *et al.*, 2007a; Fluet *et al.*, 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 (Milner *et al.*, 1997), called 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 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 **pva1** binding form, for parallel evaluation and speculation (Section 4),
- the **pcase** expression form, for nondeterminism and user-defined parallel control structures (Section 5),
- the inclusion of exceptions and exception handlers in a data parallel context (Section 6), and
- our implementation strategies for all of the above, as well as our approach to handling exceptions in the context of parallel arrays (Section 8).

We describe the parallel array mechanism in Section 3, and we illustrate the language design with a series of examples in Section 7. The examples are meant to demonstrate PML’s 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 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 (Blueloch, 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, 1999) 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; Fluet *et al.*, 2007b).

2.1 Sequential Programming

PML’s sequential core is based on a subset of Standard ML (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.

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 of CML’s concurrency mechanisms can be found in the literature (Reppy, 1999), and a description of their implementation in PML is given in a recent publication (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,
- 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 for preserving 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.

2.3.1 Parallel Tuples

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

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

```

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.

serves as a hint to the compiler and runtime 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 implication for the parallel implementation is similar to that for parallel arrays: 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 and matrices is common in parallel languages. In PML, we support such computations 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

$$[| e | p_1 \text{ in } e_1, \dots, p_n \text{ in } e_n \text{ where } e_f |]$$

where e is some expression (with free variables bound in the p_i) computing the elements of the array, the p_i are patterns binding the elements of the 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 lock-step.¹ For convenience, we also provide a parallel range form

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.

```
[ | 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 are 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 because of a pattern-match failure, 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
```

(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. The $\pi_{i,j}$ are *parallel patterns*, which are either normal patterns or else the special wildcard “?” that matches non-terminated subcomputations.

Parallel case is flexible enough to express a variety of nondeterministic parallel mechanisms, including, notably, the parallel choice binary operator $| ? |$, which nondeterministically returns either of its operands. A more detailed treatment of the semantics of parallel case, and some examples of its use, appear in Section 5 below.

Unlike the other implicitly-threaded mechanisms, parallel case is nondeterministic. We can still give a sequential semantics, but it requires including a source of non-determinism, 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

```

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

```

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

of a parallel case will be computed to completion, even in the presence of diverging sub-computations running in parallel with it. 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 regardless, 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.

3 Parallel Arrays

Comprehensions can be used to specify both SIMD parallelism that is mapped onto vector hardware (*e.g.*, Intel’s SSE instructions) and SPMD parallelism where parallelism is mapped onto multiple cores. 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

```
[| [| 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 are 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 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.

```

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 mapping 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.

An important contrast between parallel tuples and parallel arrays is this: with parallel tuples, the elements need not all 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 sub-tasks than the fork-join patterns provided by parallel arrays and tuples. Furthermore, they support speculative parallelism, since the implementation Our compiler uses a program analysis to determine those program points where a subcomputation is guaranteed never to be demanded and, thus, a cancellation 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. The short-circuiting behavior is not explicit in the function’s code, nor is it dictated by the semantics. Rather, in the implementation of `pval`, when control reaches a point where the result of an evaluation is known not to be needed, the resources devoted to that evaluation are freed and the computation is abandoned. The analysis to determine when a `pval`’s computation is subject to cancellation 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. Our analysis must respect this dependency and similar subtle dependencies. We consider an example similar to this one in Section 8 in more detail.

There are many more examples of the use of parallel bindings in Section 7. We discuss the specific mechanisms — most importantly, *futures* (Halstead Jr., 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 results are 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 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 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:

```

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

```

Touching a future encoded in this way entails only applying it to the unit constant.

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, de-structuring patterns as they appear in ML and other languages, and *parallel patterns*, which are defined below.

```

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

```

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 $e_1 \dots e_m$ 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 the branches are all compared to the those subcomputations in parallel. Any branch that matches the subcomputations at any point may transfer control to its right hand side. Note that if more than one branch matches the evaluating or evaluated subcomputations, then one of those is nondeterministically chosen as the main branch, and evaluation proceeds accordingly.

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, whether it has finished normally by evaluating to a value, or whether it has terminated by raising an exception, the `?` matches the value, 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. Furthermore, a sequential wildcard does not ignore a raised exception; raised exceptions propagate past sequential wildcards, and the program must either handle them explicitly or deliver them to the awaiting context. The choice between a nondeterministic wildcard and a sequential wildcard is consequential, and has profound effects on the behavior of a program.

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 false & somethingSlow()
of false & ? => 0
```

Once the constant discriminant `false` has been matched against the first subcomputation, the program need not wait for the second subcomputation to finish; it can return `false` straightaway. During the compilation process, the compiler will enact this improvement by inserting an explicit cancellation of the call to `somethingSlow` on the right hand side of the branch, before evaluating to 0. In this way the resources devoted to the fruitless computation are dynamically released and made available elsewhere.

In the design of the nondeterministic wildcard, we considered the possibility that nondeterministic wildcards might fail to match computations that raise exceptions. In other words, we thought about the effects of

```
pcase (raise A) of ? => 0
```

evaluating not to 0 but terminating by raising the exception `A`. We decided against it because we believe it would erode a great deal of the power of `?` as a mechanism for speculation. If we were obliged to deliver any exception value `E` when matched against a `?`, then we would need to wait for all subcomputations to complete whenever those computations might match a nondeterministic wildcard. We might be able to transform some of that waiting away, by, for example, identifying subcomputations wherein no exception can

² We have dropped the **otherwise** branch form (Fluet *et al.*, 2008a) from our design. We no longer believe it is necessary.

ever be raised, but given the prevalence of possible exceptions (whenever an integer division is performed, for example) we felt that even in that case we would be inhibiting the mechanism too much.

It remains possible not to ignore exceptions in **pcase** expressions in various familiar ways. If the expression above is rewritten with a sequential wildcard:

```
pcase (raise A) of _ => 0
```

then the expression will indeed raise *A*, as it would in a similar ML case expression. Furthermore, the exception can be handled directly in the subcomputation with an explicit **handle**³ as follows:

```
pcase ((raise A) handle A => true) of _ => 0
```

Originally, the parallel case expression was designed as a generalization of *parallel choice*. A parallel choice expression nondeterministically returns either of two subexpressions e_1 or e_2 . We write parallel choice with the infix operator `| ? |`, as in

```
 $e_1$  | ? |  $e_2$ 
```

This operator is similar to MultiLisp’s *parallel or* (and not the same as a parallel logical or). This 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 above.)

```
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 cancellation will make computational resources available for use elsewhere. If the computation is completed, this cancellation will be a harmless non-operation.

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

```
pcase  $e_1$  &  $e_2$ 
of x & ? => x
  | ? & y => y
```

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 4). This function will short-circuit when either the first or second branch is matched, implicitly canceling the computation of the other subtree. Note this short-circuiting behavior is

³ In a prior publication (Fluet *et al.*, 2008a), we presented *handle patterns* as an additional kind of parallel pattern. We have since removed it from our design in the interest of simplicity.

```

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

```

Fig. 4. Short-circuiting tree product with parallel case.

```

fun trProd t = let
  fun p (Lf 0) = raise Zero
  | p (Lf n) = n
  | p (Nd(tL, tR)) = (
    pcase p(tL) & p(tR) of pL & pR => pL * pR)
  in
  (p t) handle Zero => 0
end

```

Fig. 5. Short-circuiting tree product with parallel case and exceptions.

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 $\text{trProd}(tL)$ evaluates to 0 and $\text{trProd}(tR)$ evaluates to 1, the first branch or the third branch may be taken, but either right hand side will yield the correct result, 0.

We can also rewrite trProd using exceptions as a non-local exit mechanism (Figure 5). Note that PML borrows the **raise** and **handle** syntactic forms from SML. This example demonstrates what it already a well-known programming idiom in sequential programming; examining its use in combination with **pcase** is instructive. When using exceptions to encode non-local exit, as here, one wants control to sidestep the stack of current recursive calls to return immediately to the context awaiting the initial call. Here the nondeterminism of **pcase** is the means by which this behavior is implemented. Because of the nondeterminism inherent in **pcase**, the semantics do not dictate the delivery of a particular exception (left or right) among $p(tL)$ and $p(tR)$. Either exception is acceptable. Therefore, whether an exception is raised in computing $p(tL)$ or $p(tR)$, it may be raised further; the “other” subcomputation need not be waited for. Had we used a parallel tuple here instead of a **pcase** this would not be so. In the expression

$$(| p(tL), p(tR) |)$$

if an exception were raised in evaluating $p(tR)$, the system would, in keeping with PML’s sequential semantics, be obliged to wait for $p(tL)$ to complete to see if it too were raising an exception, in which case its exception would take precedence. The combination of **pcase** and exceptions is the means by which parallel non-local exit is achieved in PML programming.

```

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

As a third 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 6. 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, 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 7. We assume the existence of a `state` type, representing the state of whatever space (e.g., chess boards) is being searched; a predicate `isSol`

```

(* 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. 7. Searching a tree space speculatively with exceptions.

```

(* 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. 8. Searching a tree space speculatively with options.

for identifying solution states; and a function `next` which returns `SOME` pair⁴ 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.

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 8 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.

⁴ 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.

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) 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 the implicitly-threaded parallel features of the language. 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:

```
e1 handle e2
```

Although exception handlers are first-class expressions, their behavior is orthogonal to that of the parallel constructs and mostly merit no special treatment in the implementation. At the present time, PML 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 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 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. Note the use of a sequential `case` inside a `pval` block to describe the desired behavior.

```

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

```

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 cancellation 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:

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

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)) = (pcase eval(e1) & eval(e2) & eval(e3)
  of B(true) & v & ? => v
  | B(false) & ? & v => v
  | _ & _ & _ => raise RuntimeError)
```

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. Note that if the evaluation function raises a `RuntimeError` exception before the typechecking function returns an error, it will be silently canceled. 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) & eval(e)
  of (Err p, ?) => Err p
     | (A _, v) => A v)

```

7.2 Parallel Contract Checking

In this section, we extend the simple typed language of the previous section with parallel contract checking for a minimal contract system. To illustrate our point, we consider only the simplest kind of contracts, flat contracts on function arguments (Parnas, 1972; Luckham, 1990).

To define the language, we extend the `term` datatype. A contract, in this model language, is a function that consumes a value and returns a boolean. Therefore, there is no distinguished contract variant in the datatype.

```

datatype term
  = ...
  | LamG of exp * var * ty * exp
  | Blame of loc
  ...

```

`LamG` is a special form for the representation of guarded functions. The first component is a contract for the function argument; the second, third, and fourth components are the function argument, the function argument's type, and the function body, respectively. Note that the variable is in scope only in the body, not the contract. If the argument to a `LamG` function fails to meet its contract, the location of the application supplying the argument will be blamed. This blame is reified in the form of a `Blame` expression naming the location of the contract violation.

As before, we define a function

```
eval : exp -> exp
```

to evaluate expressions in this language. We omit most of its predictable definition. The evaluation of expressions in this language is mostly standard, although care must be taken at each step to check if any `Blames` have been generated, as we wish to report the earliest contract violation. In the `Add` case, for example, we evaluate both subexpressions in parallel by launching the evaluation of the second with `pval`.

```

| eval (p, Add (e1, e2)) => let
  pval v2 = eval(e2)
in
  case eval(e1)
  of Blame c => Blame c
     | N n1 => (case v2
      of Blame c => Blame c
         | N n2 => Add(n1+n2)
         | _ => raise RuntimeError)
     | _ => raise RuntimeError
end

```

It is unnecessary to launch the evaluations of both operands with `pvals`; launching just one suffices for running both simultaneously. If neither evaluation blames anything, we proceed to evaluate the addition as usual. Here, a sequential case expression is employed to express the left bias of the contract-checking system. Note how the mixed presence of parallel and sequential constructs can be used to express a particular intention.

When a contract must be checked, its evaluation is done in parallel with the evaluation of the expression as a whole. If the contract is not adhered to, the main evaluation is implicitly canceled; otherwise, it is returned.

```
| eval (p, App (e1, e2)) = let
  pval v2 = eval(e2)
  in
  case eval(e1)
  of Blame c => c
  | Lam(x,_,b) => (case v2
    of Blame c => Blame c
    | _ => eval (subst(v2,x,b)))
  | LamG(C,x,_,b) => (case v2
    of Blame c => Blame c
    | _ => let
      pval res = eval(subst(v2,x,b))
      val chk = eval((p,App(c,v2)))
      in case chk
        of B false => Blame p
         | B true => res
         | _ => raise RuntimeError
      end)
  end
```

7.3 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 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. 9. The maxT half of parallel alpha-beta pruning.

```

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 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 both to a sequential version of itself in PML and to 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 9, omitting some obvious helper functions; minT, not shown, is similar to maxT, with appropriate symmetrical modifications. Alpha-beta pruning is an inherently

```

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. 10. Primitive parallel operations.

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 cancellation of the **pval** mechanism cancels the evaluation of the right siblings of a particular subtree.

8 Implementation

We implement our implicitly-threaded parallel primitives by translating them to lower-level parallel operations, such as *futures* (Halstead Jr., 1984) and *m-variables* (Barth *et al.*, 1991). In the PML compiler, this transformation is performed on the abstract-syntax-tree (AST) representation, but here we use SML syntax to describe the transformed program fragments.

8.1 Low-level parallel primitives

Before describing our transformations, we must describe the low-level parallel-language primitives that are our target. The signature of these types and operations is given in Figure 10 and we describe them in the remainder of this section.

We use a variant of futures (Halstead Jr., 1984) to implement many of our implicitly-threaded parallel constructs. A future value is a handle to a computation that may be exe-

cuted 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 re-raised 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 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 cancellation. 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. Cancellation is implemented using nested schedulers, which are described in another paper (Fluet *et al.*, 2008b).

8.2 Parallel Tuples

For our first transformation, we revisit an earlier example:

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

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

A future is created for each element of the parallel tuple, except the first. Since the first element of the tuple will be the first element demanded, and the main thread will block until the first element is available, there is no need to incur the overhead of a future for the computation of the first element. To manage computational resources properly, when an exception is raised during the evaluation of a parallel tuple, it is necessary to install an exception handler that will cancel any running futures before propagating the exception.

Taking all this into account, the tree product code above is rewritten with futures as follows:

```

fun trProdT (Lf i) = i
  | trProdT (Nd(tL, tR)) = let
    val (l, r) = let
      val fR = Future.new (fn _ => trProdT tR)
      in
        (trProdT tL, Future.touch fR)
      handle e => (Future.cancel fR; raise e)
    end
  in
    l * r
  end

```

In general, a parallel tuple $(|e_1, \dots, e_n|)$ is translated to an expression of the form

```

let
  val f2 = Future.new (fn _ => e2)
  ...
  val fn = Future.new (fn _ => en)
in
  (e1, Future.touch f2, ..., Future.touch fn)
  handle ex => (
    Future.cancel f2; ...; Future.cancel fn;
    raise ex)
end

```

Note that if the expression e_i raises an exception (and `touch fi` raises an exception), then the cancellation of f_2, \dots, f_i will have no effect. Since we expect exceptions to be rare, we choose this transformation rather than one that installs custom exception handlers for e_1 and each `touch fi`.

8.3 Parallel Arrays

We represent parallel arrays using an immutable balanced-tree data structure called a *rope* (Boehm *et al.*, 1995). At the leaves of a rope are small vectors of elements.

```

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

```

Ropes support distributed construction and efficient concatenation, but random access is logarithmic.

Ropes, originally proposed as an alternative to strings, are immutable balanced binary trees with vectors of data at their leaves. Read from left to right, the data elements at the leaves of a rope constitute the data of the parallel array it represents. Ropes admit fast concatenation and, unlike contiguous arrays, may be efficiently allocated in memory even when very large. One disadvantage of ropes is that random access to individual data elements requires logarithmic time. Nonetheless, 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. However, a PML programmer should be aware of this representation.

As they are physically dispersed in memory, ropes 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 the rope datatype shown in

```

fun rmap (f, r) = (case r
  of Leaf v => Leaf (vmap (f, v))
    | Cat (r1, r2) => let
      val f2 = Future.new (fn _ => rmap (f, r2))
      val m1 = rmap (f, r1)
      handle e => (Future.cancel f2; raise e)
      val m2 = Future.touch f2
    in
      Cat (m1, m2)
    end)

```

Fig. 11. Mapping a function over a rope in parallel.

Figure 10 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.

One of the most common operations on parallel arrays is to apply a function in parallel to all of its elements, as in

$$[| f\ x \ | \ x \ \mathbf{in} \ a \ |]$$

for any function f . We transform this computation to the internal function `rmap`, whose definition is given in Figure 11. Note that if `rmap (f, r1)` raises an exception, then the future evaluating the map of the right half of the rope is canceled, and the exception from the map of the left half is propagated. If `rmap (f, r2)` raises an exception, then it will be propagated by the `touch f2`. By this mechanism, we raise the sequentially first exception as discussed above.

The maximum length of the vector at each leaf is controlled by a compile-time option; its default value is currently 256. Altering the maximum leaf length can affect the execution time of a given program. If the leaves store very little data, then ropes become very deep. Per the parallel decomposition shown in Figure 11, small leaves correspond to the execution of many futures. This leads to good load balancing when applying the mapped function to an individual element is relatively expensive. By contrast, large leaves correspond to the execution of few futures; this is advantageous when applying the mapped function to an individual element is relatively cheap. Allowing the user to vary the maximum leaf size on a per-compilation basis gives some rough control over these tradeoffs. A more flexible system would allow the user to specify a maximum leaf size on a per-array basis, although many decisions remain about how to provide such a facility.

Another common operation on ropes is reduction by some associative operator \oplus . Compensation code to ensure that the sequentially first exception is raised is similar to that of `rmap`. A reduction of a parallel array is transformed to the internal function `rreduce`, whose definition is given in Figure 12.

8.4 Parallel Bindings

The implementation of parallel bindings introduces a future for each `pval` to be executed in parallel to the main thread of control and introduces cancellations when variables

```

fun rreduce (f, z, r) = (case r
  of Leaf v => vreduce (f, z, v)
    | Cat(r1, r2) => let
      val f2 = Future.new (fn _ => rreduce (f, z, r2))
      val v1 = rreduce (f, z, r1)
      handle e => (Future.cancel f2; raise e)
      val v2 = Future.touch f2
    in
      f (b1, b2)
    end)

```

Fig. 12. Reducing a function over a rope in parallel.

bound in the corresponding **pval** become unused on a control-flow path. Note that such control-flow paths may be implicit via a raised exception. As with parallel tuples, we do not introduce a future for a **pval** whose result is demanded by the main thread of control without any (significant) intervening computation.

To illustrate the implementation of parallel bindings, we present the translation of the function `trProd` from Figure 3.

```

fun trProdT (Lf i) = i
  | trProdT (Nd (tL, tR)) = let
    val fR = future (fn _ => trProdT tR)
  in let
    val pL = trProdT tL
  in
    if (pL = 0) then (cancel fR; 0)
    else (pL * (touch fR))
  end
  handle e => (cancel fR; raise e)
end

```

Note that the translation introduces a future for only one of the **pval** computations, since `pL` is demanded immediately. The translation also inserts a cancellation of the future if an exception is raised that would exit the function body. As noted above, we expect exceptions to be rare, so we adopt a translation that may introduce redundant (but idempotent) cancellations (*i.e.*, canceling all introduced futures in a universal exception handler), rather than installing custom exception handlers to minimize the number of futures canceled.

Although the introduction of future cancellations is relatively straightforward, care must be taken to properly account for **pval** bound variables that are used in other **pval** computations. Consider the following example:

```

val r = let
  pval x = f 1
  pval y = g 2 + x
  in
    if h 3 then x else y
  end

```

and its translation:

```

val rT = let
  val fx = Future.new (fn _ => f 1)
  val fy = Future.new (fn _ => g 2 + Future.touch fx)
in
  if h 1
    then (Future.cancel fy; Future.touch fx)
    else ((* Future.cancel fx; *) Future.touch fy)
  end
  handle e => (Future.cancel fy; Future.cancel fx; raise e)

```

Note that touching `fx` twice, which might happen in certain executions of this code, is not problematic, since touches after the initial touch simply return the demanded value immediately.

We cannot include the commented `Future.cancel fx`, as the result of `fx` might be demanded to satisfy the demand for the result of `fy`. In the **then** branch of the computation, `fx` is simply canceled as it should be. In the exception handler, `fy` is of necessity canceled before `fx`. Canceling in this order avoids the unsatisfiable touch should `fx` be canceled, then touched in `fy` before `fy` were canceled.

8.5 Parallel Case

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.

Before we can describe the compilation technique, we need to define some notation. We use Π_i to denote the left-hand-side of the i th row of the **pcase** (i.e., $\pi_{i,1} \& \dots \& \pi_{i,n}$). The state machine for this **pcase** will have up to 2^n states. We identify these states by bit strings of length n , where the i th bit is 1 if e_i has terminated. We use \vec{S}_i to denote the i th state’s bit string. If \vec{B} is a bit string, then we use $B[i]$ to denote its i th bit, and we use the \wedge to denote bit-wise anding of bit strings. Lastly, if $\Pi = \pi_1 \& , \dots \& \pi_n$ and \vec{B} is an n -bit string with m bits set, then

$$\Pi|_{\vec{B}} = (\pi_{i_1}, \dots, \pi_{i_m})$$

where $1 \leq i_1 < \dots < i_m \leq n$ and $B[i_j] = 1$ for $1 \leq j \leq m$.

The first step in compiling a parallel-case expression is to lift each right-hand-side action into its own function that takes the variables bound by its left-hand-side pattern as parameters. This transformation allows us to duplicate actions without risking code blowup and results in an expression of the form

```

pcase e1 & ⋯ & en
of π1,1 & ⋯ & π1,n => act1
  | ⋯
  | πm,1 & ⋯ & πm,n => actm

```

where n is the number of parallel clauses, m is the number of cases, and the act_i are function applications.

The second step of the compilation process involves identifying which cases (i.e., rows of the **pcase**) can be matched in which states. For each row, we define an n -bit string as

follows:

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

Then, for each state S , we define the set of *applicable cases* for S to be

$$ACase(S) = \{\Pi_i |_{\vec{S}} \mid \vec{P}_i \wedge \vec{S} = \vec{P}_i\}$$

Once we have partitioned the patterns, we can generate the code that implements the state machine. While the translation is straightforward, there are many pieces to the resulting code. These include the following:

- Allocation of reference cells for the subcomputation results. These are initialized to NONE.
- Allocation of cancelable objects to support cancellation of subcomputations.
- Allocation of an m-variable to hold the current state. We rely on the synchronous behavior of this variable to guarantee the atomicity of the state-machine transitions.
- A transition function for each subcomputation. The i th transition function, $trans_i$, takes the current state S from the m-variable, records the result of the i th subcomputation, and then calls the state function for the next state S' , where $S'[i] = 1$ and $S'[j] = S[j]$ for $i \neq j$.
- A state function for each state other than the initial state. The state function for a state S must test the available values against the cases in $ACase(S)$. If there is a match, the corresponding action function is called. Otherwise, the state function puts S into the m-variable and the scheduler is called.
- An action function for each right-hand-side expression of the **pcase** that takes the variables bound by its left-hand-side pattern as parameters. This function includes cancellation code for any subcomputations that might still be running (*i.e.*, any subcomputations that are matched by $?$ on the left-hand-side pattern).
- An action function act_{exn} for when a subcomputation raises an exception. The body of this action takes the state from the m-variable, cancels each of the subcomputations, and then reraises the exception.
- Code to spawn the subcomputations. For the expression e_i , we generate

```
Cancel.spawn (c_i, trans_i(e_i) handle ex => act_exn ())
```

If e_i terminates normally, the i th transition function is invoked and, otherwise, if e_i raises an exception the exception action is invoked.

To better understand how this translation works, recall the implementation of `trProd` from Figure 4. Figure 13 gives the result of translating the **pcase** in this function. The translation closely follows the description from above, but there are a couple of additional details that require explanation. The body of the translated **pcase** is wrapped as follows:

```
pcaseWrapper (fn (return, exnReturn) => ...)
```

The `pcaseWrapper` function abstracts the control-flow mechanisms that are used to manage returning from the recursive call. The `return` argument function wraps the normal return continuation, while the `exnReturn` argument function wraps the exception-handler continuation.

```

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) = pcaseWrapper (
    fn (return, exnReturn) => let
      val r1 = ref NONE
      val r2 = ref NONE
      val c1 = Cancel.new ()
      val c2 = Cancel.new ()
      val state = MVar.new 0
      fun resume st = (MVar.put(state, st); dispatch())
      fun trans1 v = let
        val st = MVar.take state
        in
          r1 := SOME v;
          case st
            of 0 => state10()
              | 1 => state11()
          end
        and trans2 v = let
          val st = MVar.take state
          in
            r2 := SOME v;
            case st
              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. 13. The translation of trProd from Figure 4.

8.6 Scheduling

The implementation techniques described in this section all give rise to a tree-structured decomposition of the work. It is the responsibility of the scheduler to map this decomposition onto the available processor resources. The details of our scheduling infrastructure are described in another paper (Fluet *et al.*, 2008b), but we cover some of the highlights here.

The scheduling policy chooses a mapping from the tasks generated by our implicitly-threaded constructs to processors. In our implementation, this mapping is determined at run time. There are a wide array of dynamic scheduling policies in the literature, and the choice of one of these policies can have a significant impact on performance. Broadly speaking, scheduling policies fall into one of two categories: *work sharing* and *work stealing* (Blumofe & Leiserson, 1999). In work sharing, each worker plays an active role in distributing threads among processors. Whenever new threads are created by a worker, the scheduler delegates some of them to other workers with the aim of spreading work to idle processors. Work stealing takes the opposite approach, by making idle workers responsible for taking threads away from busy workers. Because of this lazy approach to distributing work, work stealing is well suited to scheduling fine-grained computations.

Our scheduling infrastructure is designed to support a wide variety of scheduling policies. To support the mechanisms described in this paper, we use a work-stealing scheduler that is an extension of the Cilk work-stealing policy (Blumofe *et al.*, 1995; Frigo *et al.*, 1998) with support for cancellation. This scheduling policy gives a roughly breadth-first distribution of work across processors, but each processor pursues a depth-first traversal of the parallel decomposition. To support the infinite-processor abstraction, we dynamically create additional workers when one worker gets hung up on a task. In effect, this increases the multiplexing of the processor resources.

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. PML is yet to implement flattening, although we expect to devote great attention to the topic as our work moves forward.

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 PML’s in 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 cancellation of spawned siblings; we have suggested some encodings of similar parallel patterns in Section 7 above.

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 towards regular, massively parallel operations on homogeneous collections of data, in marked contrast to the example presented in Section 7 above.

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.

Programming parallel hardware effectively is difficult, but there have been a some important recent achievements. Google's MapReduce programming model (Dean & Ghemawat, 2004) has been a success in processing large datasets in parallel. Sawzall, another Google project, is a system for analysis of large datasets 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 which allows the programmer to use a GPU as a stream co-processor.

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 mechanisms include standard mechanisms, such as parallel tuples and nested-parallel arrays, as well as more novel features, such as parallel bindings and non-deterministic parallel cases. We have illustrated these mechanisms with a number of examples and given an overview of their implementation in the Manticore system.

We have been working on a prototype implementation of the Manticore system since January of 2007. The implementation is largely feature-complete and is available on request. The features described in this paper, however, do not have optimized implementations yet, which is why we omit performance measurements.

References

- Armstrong, Joe, Virding, Robert, Wikström, Claes, & Williams, Mike. (1996). *Concurrent programming in ERLANG (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd.
- Barth, Paul, Nikhil, Rishiyur S., & Arvind. (1991). M-structures: Extending a parallel, non-strict, functional language with state. *Pages 538–568 of: Functional programming languages and computer architecture (fpca '91)*. Lecture Notes in Computer Science, vol. 523. New York, NY: Springer-Verlag.

- Barton, Reid, Adkins, Dan, Prokop, Harald, Frigo, Matteo, Joerg, Chris, Renard, Martin, Dailey, Don, & Leiserson, Charles. (1998). *Cilk Pousse*. Viewed on March 20, 2008 at 2:45 PM.
- Blelloch, Guy E. (1996). Programming parallel algorithms. *Communications of the ACM*, **39**(3), 85–97.
- Blelloch, Guy E., & Greiner, John. (1996). A provable time and space efficient implementation of NESL. *Pages 213–225 of: Proceedings of the 1996 ACM SIGPLAN international conference on functional programming*. New York, NY: ACM.
- Blelloch, Guy E., Chatterjee, Siddhartha, Hardwick, Jonathan C., Sipelstein, Jay, & Zagha, Marco. (1994). Implementation of a portable nested data-parallel language. *Journal of parallel and distributed computing*, **21**(1), 4–14.
- Blumofe, Robert D., & Leiserson, Charles E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM*, **46**(5), 720–748.
- Blumofe, Robert D., Joerg, Christopher F., Kuszmaul, Bradley C., Leiserson, Charles E., Randall, Keith H., & Zhou, Yuli. (1995). Cilk: An efficient multithreaded runtime system. *Pages 207–216 of: Proceedings of the 5th ACM SIGPLAN symposium on principles & practice of parallel programming*. New York, NY: ACM.
- Boehm, Hans-J., Atkinson, Russ, & Plass, Michael. (1995). Ropes: an alternative to strings. *Software: Practice and experience*, **25**(12), 1315–1330.
- Buck, Ian, Foley, Tim, Horn, Daniel, Sugerma, Jeremy, Fatahalian, Kayvon, Houston, Mike, & Hanrahan, Pat. (2004). Brook for GPU: stream computing on graphics hardware. *Proceedings of ACM SIGGRAPH 2004*, **23**(3), 777–786.
- Chakravarty, Manuel M. T., & Keller, Gabriele. (2000). More types for nested data parallel programming. *Pages 94–105 of: Proceedings of the fifth ACM SIGPLAN international conference on functional programming*. New York, NY: ACM.
- Chakravarty, Manuel M. T., Keller, Gabriele, Leshchinskiy, Roman, & Pfannenstiel, Wolf. (2001). Nepal – Nested Data Parallelism in Haskell. *Pages 524–534 of: Proceedings of the 7th international Euro-Par conference on parallel computing*. Lecture Notes in Computer Science, vol. 2150. New York, NY: Springer-Verlag.
- Chakravarty, Manuel M. T., Leshchinskiy, Roman, Peyton Jones, Simon, Keller, Gabriele, & Marlow, Simon. (2007). Data Parallel Haskell: A status report. *Pages 10–18 of: Proceedings of the ACM SIGPLAN workshop on declarative aspects of multicore programming*. New York, NY: ACM.
- Dailey, Don, & Leiserson, Charles E. (2002). Using Cilk to write multiprocessor chess programs. *The journal of the international computer chess association*.
- Danaher, John S., Lee, I-Ting Angelina, & Leiserson, Charles E. (2006). Programming with Exceptions in JCilk. *Science of computer programming*, **63**(2), 147–171.
- Dean, Jeffrey, & Ghemawat, Sanjay. 2004 (Dec.). MapReduce: Simplified data processing on large clusters. *Pages 137–150 of: Proceedings of the sixth symposium on operating systems design and implementation*.
- Fluet, Matthew, Rainey, Mike, Reppy, John, Shaw, Adam, & Xiao, Yingqi. (2007a). Manticore: A heterogeneous parallel language. *Pages 37–44 of: Proceedings of the ACM SIGPLAN workshop on declarative aspects of multicore programming*. New York, NY: ACM.
- Fluet, Matthew, Ford, Nic, Rainey, Mike, Reppy, John, Shaw, Adam, & Xiao, Yingqi. (2007b). Status Report: The Manticore Project. *Pages 15–24 of: Proceedings of the 2007 ACM SIGPLAN workshop on ML*. New York, NY: ACM.
- Fluet, Matthew, Rainey, Mike, Reppy, John, & Shaw, Adam. (2008a). Implicitly-threaded parallelism in Manticore. *Pages 119–130 of: Proceedings of the 13th ACM SIGPLAN international conference on functional programming*. New York, NY: ACM.
- Fluet, Matthew, Rainey, Mike, & Reppy, John. (2008b). A scheduling framework for general-purpose

- parallel languages. *Pages 241–252 of: Proceedings of the 13th ACM SIGPLAN international conference on functional programming*. New York, NY: ACM.
- Frigo, Matteo, Leiserson, Charles E., & Randall, Keith H. 1998 (June). The implementation of the Cilk-5 multithreaded language. *Pages 212–223 of: Proceedings of the SIGPLAN conference on programming language design and implementation (PLDI '98)*.
- Gansner, Emden R., & Reppy, John H. (eds). (2004). *The Standard ML basis library*. Cambridge, England: Cambridge University Press.
- Gaudiot, Jean-Luc, DeBoni, Tom, Feo, John, Bohm, Wim, Najjar, Walid, & Miller, Patrick. (1997). The Sisal model of functional programming and its implementation. *Pages 112–123 of: Proceedings of the 2nd AIZU international symposium on parallel algorithms / architecture synthesis (pAs '97)*. Los Alamitos, CA: IEEE Computer Society Press.
- GHC. *The Glasgow Haskell Compiler*. Available from <http://www.haskell.org/ghc>.
- Halstead Jr., Robert H. (1984). Implementation of multilisp: Lisp on a multiprocessor. *Pages 9–17 of: Conference record of the 1984 ACM symposium on Lisp and functional programming*. New York, NY: ACM.
- Hammond, Kevin. (1991). *Parallel SML: a functional language and its implementation in Dactl*. Cambridge, MA: The MIT Press.
- Harris, Tim, Marlow, Simon, Peyton Jones, Simon, & Herlihy, Maurice. (2005). Composable memory transactions. *Pages 48–60 of: Proceedings of the 2005 ACM SIGPLAN symposium on principles & practice of parallel programming*. New York, NY: ACM.
- Hauser, Carl, Jacobi, Cristian, Theimer, Marvin, Welch, Brent, & Weiser, Mark. 1993 (Dec.). Using threads in interactive systems: A case study. *Pages 94–105 of: Proceedings of the 14th ACM symposium on operating system principles*.
- Hedqvist, Pekka. 1998 (June). *A parallel and multithreaded ERLANG implementation*. M.Phil. thesis, Computer Science Department, Uppsala University, Uppsala, Sweden.
- Jones, Mark P., & Hudak, Paul. 1993 (Aug.). *Implicit and explicit parallel programming in Haskell*. Tech. rept. Research Report YALEU/DCS/RR-982. Yale University.
- Le Fessant, Fabrice, & Maranget, Luc. (1998). Compiling join-patterns. *Pages 205–224 of: 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 Publishers.
- Leroy, Xavier, & Pessaux, François. (2000). Type-based analysis of uncaught exceptions. *ACM transactions on programming languages and systems*, **22**(2), 340–377.
- Leshchinskiy, Roman, Chakravarty, Manuel M. T., & Keller, Gabriele. (2006). Higher order flattening. *Pages 920–928 of: Alexandrov, V., van Albada, D., Sloot, P., & Dongarra, J. (eds), International conference on computational science (ICCS '06)*. LNCS, no. 3992. New York, NY: Springer-Verlag.
- Luckham, David. (1990). *Programming with specifications*. Texts and Monographs in Computer Science. Springer-Verlag.
- McCarthy, John. (1963). A Basis for a Mathematical Theory of Computation. *Pages 33–70 of: Braffort, P., & Hirschberg, D. (eds), Computer programming and formal systems*. North-Holland, Amsterdam.
- Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The Definition of Standard ML (revised)*. Cambridge, MA: The MIT Press.
- Nikhil, Rishiyur S. 1991 (July). *ID language reference manual*. Laboratory for Computer Science, MIT, Cambridge, MA.
- Nikhil, Rishiyur S., & Arvind. (2001). *Implicit parallel programming in pH*. San Francisco, CA: Morgan Kaufmann Publishers.
- Parnas, David L. (1972). A technique for software module specification with examples. *Communications of the ACM*, **15**(5), 330–336.

- Peyton Jones, Simon, Gordon, Andrew, & Finne, Sigbjorn. (1996). Concurrent Haskell. *Pages 295–308 of: Conference record of the 23rd annual ACM symposium on principles of programming languages (popl '96)*. New York, NY: ACM.
- Peyton Jones, Simon, Reid, Alastair, Henderson, Fergus, Hoare, Tony, & Marlow, Simon. (1999). A semantics for imprecise exceptions. *Pages 25–36 of: Proceedings of the SIGPLAN conference on programming language design and implementation (PLDI '99)*. New York, NY: ACM.
- Pike, Rob, Dorward, Sean, Griesemer, Robert, & Quinlan, Sean. (2005). Interpreting the data: Parallel analysis with sawzall. *Scientific programming journal*, **13**(4), 227–298.
- Reppy, John, Russo, Claudio, & Xiao, Yingqi. (2009). Parallel Concurrent ML. *Proceedings of the 14th ACM SIGPLAN international conference on functional programming*. New York, NY: ACM. *To appear*.
- Reppy, John H. (1991). CML: A higher-order concurrent language. *Pages 293–305 of: Proceedings of the SIGPLAN conference on programming language design and implementation (PLDI '91)*. New York, NY: ACM.
- Reppy, John H. (1999). *Concurrent programming in ML*. Cambridge, England: Cambridge University Press.
- Shaw, Adam. 2007 (July). *Data parallelism in Manticore*. M.Phil. thesis, University of Chicago. Available from <http://manticore.cs.uchicago.edu>.
- Tarditi, David, Puri, Sidd, & Oglesby, Jose. (2006). Accelerator: using data parallelism to program gpus for general-purpose uses. *Sigops oper. syst. rev.*, **40**(5), 325–335.
- Trinder, Philip W., Hammond, Kevin, Loidl, Hans-Wolfgang, & Peyton Jones, Simon L. (1998). Algorithm + strategy = parallelism. *Journal of functional programming*, **8**(1), 23–60.
- Yi, Kwangkeun. (1998). An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Sci. comput. program.*, **31**(1), 147–173.