

THE UNIVERSITY OF CHICAGO

DRAFT: DATA PARALLELISM IN MANTICORE

A PAPER SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ADAM SHAW

CHICAGO, ILLINOIS
JULY 2007

ABSTRACT

Parallel hardware is becoming increasingly widespread. Modern game consoles are multicore vector machines, and a growing percentage of desktop computers are parallel machines as well. These machines support parallelism at multiple granularities, from SIMD to multiple processors.

At the same time, parallel abstractions are already a common desideratum in software. In individual systems we have had to make do with concurrency in place of parallelism, so we could at least simulate simultaneous computations in a given program. But with the recent rise of parallel hardware, applications are now in a position to exploit multiple granularities of true parallelism.

As an example, consider a parallel ray tracer consisting of a GUI for user interaction sitting on top of an ray-tracing layer that does as much of its work in parallel as possible. Such an application would be conveniently expressed with threads at the user interface layer and data parallelism at the image processing layer.

The Manticore programming language is designed to support applications of this general shape. That is, it is designed to bring together the various parallel abstractions of software and the heterogeneous parallel elements of hardware.

Manticore is a strict, type-safe, mostly pure functional programming language built around a pared-down core of SML '97 that omits stateful computations. Explicit nested data parallel constructs in the style of NESL and its successors are layered on top of this substrate for high-level parallel programs. We also include parallel concurrency abstractions inspired by CML for explicit management of parallel computations.

We address implementational issues, focusing on how parallel constructs are compiled with the flattening transformation to realize parallel performance benefits. We also

formalize a sequential semantics of the language and consider some subtleties related to respecting it in a parallel implementation.

TABLE OF CONTENTS

ABSTRACT	ii
LIST OF FIGURES	vii
Chapter	
1 INTRODUCTION	1
2 LANGUAGE FEATURES	3
2.1 Parallel Arrays	4
2.2 Parallel Comprehensions	6
2.3 Parallel Ranges	7
2.4 Parallel Tuples	8
2.5 Parallel Bindings	10
2.5.1 Futures	10
2.5.2 Parallel Evaluation Semantics	12
2.5.3 Sequential Semantics	13
2.5.4 Remarks	13
2.6 Parallel Choice	16
2.7 Exceptions	18
2.8 Remarks	19
3 APPLICATIONS OF MANTICORE	22
3.1 Parallel Tuples for Parallel Searches	22
3.2 Some Operations on Matrices	23

3.2.1	Dense Matrix Transposition	23
3.2.2	Sparse Vector Matrix Multiplication	24
3.3	Quicksort	24
3.4	Selecting the Distinct Elements of an Array	25
3.5	Image Transformations	27
3.5.1	Minification	27
3.5.2	Image Compression	29
3.5.3	The Mandelbrot Set	31
3.6	The Barnes-Hut force calculation algorithm	32
3.7	Google's MapReduce	34
4	COMPILING MANTICORE BY PROGRAM TRANSFORMATION . . .	36
4.1	The Flattening Transformation: An Overview	36
4.2	Ropes	41
4.3	Exceptions	43
4.3.1	Raising the Leftmost Exception	43
4.3.2	Handling Flattened Structures	45
4.3.3	Inadvertently Introducing Nontermination	48
4.4	Flattening Tuples	49
4.4.1	Flattening Tuples, Formally	50
4.5	Fusion	52
4.6	The Compiler as a Formal System	53
4.6.1	\mathbf{M}	54
4.6.2	\mathbf{FM}	54
4.6.3	\mathbf{DM}	56
4.6.4	\mathbf{TM}	56
5	RELATED WORK	64

5.1	StreamIt	64
5.2	Cilk	67
5.3	Other Contemporary Parallel Languages	74
5.3.1	<i>pH</i>	74
5.3.2	Eden	76
5.3.3	The DARPA HPCS Languages	76

Appendix

A	IMPLEMENTATION SKETCHES	78
---	-----------------------------------	----

	REFERENCES	83
--	----------------------	----

LIST OF FIGURES

2.1	Parallel arrays.	5
2.2	Parallel comprehensions.	7
2.3	Parallel ranges.	8
2.4	Parallel tuples.	9
2.5	Canceling substitution. Canceling substitution is distinguished from normal substitution by a subscript c on the right bracket of the mapping.	13
2.6	Parallel bindings. In the dynamic semantics, the superscript annotation on the substitution means “for all variables x that appear in the pattern p .”	14
2.7	The interactions of parallel and sequential bindings and tuples.	15
2.8	Small-step evaluation rules for amb	17
2.9	Parallel choice.	18
2.10	Raising exceptions.	18
2.11	Handling exceptions.	19
3.1	The slice of the complex plane extending two units from 0 on both axes. All points in the shaded regions have modulus > 2	31
4.1	A New Yorker cartoon.	37
4.2	The cartoon from Figure 4.1, partitioned.	39
4.3	The grammar of \mathbb{M}	55
4.4	The grammar of \mathbb{FM}	57
4.5	The $\xrightarrow{\mathbb{F}}$ transformation, which proceeds in three stages. The first stage flattens nested parallel tuples; the operator \mathcal{T} is defined in Section 4.4. The second stage desugars parallel tuples and parallel assignments, and the third stage flattens nested parallel comprehensions.	58

4.6	The grammar of \mathbb{DM}	59
4.7	Syntactic sugar for \mathbb{DM}	60
4.8	The $\xrightarrow{\mathbb{D}}$ transformation. The functions <i>joinN</i> and <i>splitN</i> create the “rumpled” structures discussed in Subsection 4.3.2.	60
4.9	Ropes in \mathbb{TM} . See Appendix A for more details.	61
4.10	The grammar of \mathbb{TM}	62
4.11	$\xrightarrow{\mathbb{T}}$	63
A.1	Ropes in \mathbb{TM}	78
A.2	Rope construction functions in \mathbb{TM}	79
A.3	Distributed map and filter in \mathbb{TM}	80
A.4	The trap abstraction in \mathbb{TM}	81
A.5	Distributed reduction in \mathbb{TM}	81
A.6	Futuristic traversals in \mathbb{TM}	82

CHAPTER 1

INTRODUCTION

Parallel hardware is becoming increasingly widespread. Modern game consoles are multicore vector machines, and a growing percentage of desktop computers are parallel machines as well. These machines support parallelism at multiple granularities, from SIMD to multiple processors.

At the same time, parallel abstractions are already a common desideratum in software. In individual systems we have had to make do with concurrency in place of parallelism, so we could at least simulate simultaneous computations in a given program. But with the recent rise of parallel hardware, applications are now in a position to exploit multiple granularities of true parallelism.

As an example, consider a parallel ray tracer consisting of a GUI for user interaction sitting on top of an ray-tracing layer that does as much of its work in parallel as possible. Such an application would be conveniently expressed with threads at the user interface layer and data parallelism at the image processing layer.

The Manticore programming language is designed to support applications of this general shape. That is, it is designed to bring together the various parallel abstractions of software and the heterogeneous parallel elements of hardware.

Manticore is a strict, type-safe, mostly pure functional programming language built around a pared-down core of SML '97 that omits stateful computations. Explicit nested data parallel constructs in the style of NESL and its successors are layered on top of this substrate for high-level parallel programs. We also include parallel concurrency abstractions inspired by CML for explicit management of parallel computations.

We address implementational issues, focusing on how parallel constructs are compiled with the flattening transformation to realize parallel performance benefits. We also formalize a sequential semantics of the language and consider some subtleties related to respecting it in a parallel implementation.

This paper is organized as follows.

- We present Manticore the programming language and an overview of its semantics. We discuss its type system and formulate a sequential semantics for the language.
- We develop sample applications in Manticore.
- We discuss compilation of Manticore, devoting particular attention to compilation of nested data parallel constructs. Under the hood, parallel arrays are represented by a *rope* data structure similar to Boehm et al.’s ropes [6]; this representation is favorable from a variety of standpoints. We describe how our implementation respects our sequential semantics in certain sensitive cases.
- We consider Manticore in the context of related work. In particular, we contrast Manticore with StreamIt [31], a modern stream-based programming language, Cilk [5], a version of C augmented with explicit parallel annotations, NESL and Nepal/DPH, functional parallel languages and our closest predecessors, and other related languages and technologies.

The work here is complemented by the simultaneous building of a prototype implementation.

CHAPTER 2

LANGUAGE FEATURES

Manticore can be loosely described as core ML plus parallel CML plus nested data parallelism. By “core ML” we mean a subset of Standard ML [22] that includes higher-order functions, algebraic datatypes, parametric polymorphism, pattern matching, and type inference, while excluding stateful features such as ref cells. CML [26] provides support for threads and synchronous message passing; in Manticore (as opposed to the original CML) we run concurrent computations in parallel: hence, “parallel CML.” The incorporation of data parallel features into a functional language was pioneered in NESL [4] and has been explored further by Nepal [9], since renamed to Data Parallel Haskell [17, 8, 19]. Manticore’s CML features can be considered standard CML, and the present work does not treat them. Data parallelism in Manticore is the main subject of the present work and is described in detail below.

Manticore’s basic data parallel constructs are *parallel arrays*, *parallel comprehensions*, *parallel tuples*, *parallel bindings* and *parallel choices*. (Throughout this paper we will use the terms *array* and *comprehension* freely for parallel constructs when there is no ambiguity.) *Parallel ranges* are a special case of parallel arrays. We address each of these features in turn in the following sections. Each feature is presented in terms of its syntax, static semantics, and dynamic semantics. Manticore also includes facilities for raising and handling exceptions. The inclusion of exceptions introduces some potentially troublesome semantic issues, which we address.

In the following exposition, the metavariable e stands in for expressions, p for SML-style patterns, and pb for bindings of the form p **in** e , which appear in comprehensions.

The static semantics of each parallel expression form is presented in the style of the Definition of Standard ML [22]. The dynamic semantics is given as a rewriting of each

term into SML extended with McCarthy’s **amb** [21] (see Section 2.6), whose strict, sequential evaluation semantics then apply.

We place a hat on an expression (as in \widehat{e}) to indicate the translation of that expression into its sequential counterpart. Our presentation of the sequential semantics is inexhaustive in that we omit rules that do not pertain to the parallel features of the language; it stands to reason that sequentializing non-parallel features of the language is not complicated, consisting only of pushing the sequentialization into the term structure, as in, for example, $\widehat{e_1 :: e_2} = \widehat{e_1} :: \widehat{e_2}$.

There are brief discussions of execution time of the evaluation of parallel expressions in the sections that follow. Please assume, unless otherwise noted, that the evaluation takes place on a machine with a finitely many but unbounded number of processors, so that we may execute as many separate computations in parallel as we need, provided they are free of interdependencies.

[[notation for substitution; other notation]]

2.1 Parallel Arrays

The parallel array is an array-like structure whose components may be computed in parallel. Dynamically, the computation of the subexpressions of a parallel array may occur in parallel on arbitrarily many processing elements, to be joined on mutual completion. The parallel array delimiters are `[` and `]` in ASCII, or `[[` and `]]` in the typesetting of this paper. The syntax and semantics of parallel arrays are given in Figure 2.1. An example parallel array is

`[[1, 2, 4, 8, 16]]`

This constant expression has the type `int parray`. Contrast it with the following dynamically-computed array:

`[[pow(2,0), pow(2,1), pow(2,2), pow(2,3), pow(2,4)]]`

syntax	$[[e_1, \dots, e_n]]$
static semantics	$\frac{C \vdash e_1 \Rightarrow \tau \dots C \vdash e_n \Rightarrow \tau}{C \vdash [[e_1, \dots, e_n]] \Rightarrow \tau \text{ parray}}$
dynamic semantics	$[\hat{e}_1, \dots, \hat{e}_n]$

Figure 2.1: Parallel arrays.

In this case, the elements of the array can be computed in parallel, and, ignoring for the moment the time needed to launch the individual computations and join them when done, the amount of time needed to compute the whole parallel array is only the maximum time needed to compute any of its individual elements.

Unlike, for example, sequential linked lists, parallel arrays are implemented such that they may be consumed and produced in parallel by multiple processing elements working simultaneously. We expect the ML program

```
List.map (fn x => x + 1) [1, 2, 3, 4, 5, 6, 7, 8]
```

to perform eight incrementing operations in some time $8t$, where t is the time it takes to increment an integer, even though there are no interdependencies between any of those operations. In theory, we expect the corresponding Manticore program

```
[[ x+1 | x in [[ 1, 2, 3, 4, 5, 6, 7, 8 ]] ]]
```

to take time $8t/n$ where n is the number of processing elements in the given idealized (i.e., one that manages parallel computations without incurring overhead) machine. In practice, we have designed a system for efficient parallel maps (and other computations) over arrays, the details of which appear in Chapter 4.

2.2 Parallel Comprehensions

Parallel comprehensions provide a high-level expression form for what are essentially iterative computations over parallel arrays. The syntax of comprehensions is inspired by definitional set notation and has been present in programming languages since at least the advent of SETL [29]. Consider the following parallel comprehension:

```
[[ x * 2 | x in [[ 1, 2, 4, 8 ]] where x < 5 ]]
```

This particular comprehension will evaluate to the following `int parray`:

```
[[ 2, 4, 8 ]]
```

The formal description of parallel comprehensions appears in Figure 2.2. Comprehensions are the central expression form of the flattening transformation, which is set forth in detail in Chapter 4.

Parallel comprehensions can draw values from multiple parallel arrays as in the following:

```
[[ (i, f) | i in [[ 0, 1, 2 ]], f in [[ 3.0, 3.1, 3.14 ]] ]]
```

This expression evaluates to an array of three pairs, `[(0, 3.0), (1, 3.1), (2, 3.14)]`. In comprehensions the comma-delimited binding expressions to the right of the bar (`|`) obey “zip semantics,” as in NESL’s parallel comprehensions, rather than “product semantics,” as in Haskell’s list comprehensions. That is to say, the elements from the various arrays to the right of the bar are processed in lockstep. This lockstep processing terminates when we reach the end of the shortest array. Under product semantics (currently those of Data Parallel Haskell’s comprehensions), the expression above would evaluate to an array of nine pairs.

syntax	$[[e_0 \mid p_1 \text{ in } e_1, \dots, p_n \text{ in } e_n \langle \text{where } e_c \rangle \mid]]$
static semantics	$\frac{C \vdash e \Rightarrow \tau \text{ parray} \quad C \vdash p \Rightarrow (VE, \tau)}{C \vdash p \text{ in } e \Rightarrow VE}$ $\frac{C \vdash pb \Rightarrow VE \quad C + VE \vdash e \Rightarrow \tau \quad \langle C + VE \vdash e' \Rightarrow \text{bool} \rangle}{C \vdash [[e \mid pb \langle \text{where } e' \rangle \mid]] \Rightarrow \tau \text{ parray}}$
dynamic semantics	<pre> let fun f (x1::r1, ..., xn::rn, acc) = (case (x1, ..., xn) of (p1, ..., pn) => if \widehat{e}_c then f (r1, ..., rn, $\widehat{e}::\text{acc}$) else f (r1, ..., rn, acc) _ => f (r1, ..., rn, acc) (* end case *)) f (_, ..., _, acc) = rev acc in f ($\widehat{e}_1, \dots, \widehat{e}_n, []$) end </pre>

Figure 2.2: Parallel comprehensions.

2.3 Parallel Ranges

Manticore includes parallel range expressions, an expression form that is present in both NESL and the various parallel Haskell languages. Ranges are determined by two (inclusive) endpoint expressions and an expression giving the size of the gaps between elements. Consider the following example:

```
[[ 1 to 31 by 10 ]]
```

This constant range expression happens to evaluate to `[[1,11,21,31]]` and has type `int parray`. If a range appears with no “by clause”, as in `[[1 to 20]]`, it is assumed that the step size is 1.

syntax	$[[e_1 \text{ to } e_2 \text{ by } e_3]]$
static semantics	$\frac{C \vdash e_1, e_2, e_3 \Rightarrow \text{int}}{C \vdash [[e_1 \text{ to } e_2 \text{ by } e_3]] \Rightarrow \text{int parray}}$
dynamic semantics	<pre> let fun build (lo,hi,s) = if (lo > hi) then nil else lo :: build (lo+s,hi,s) in build (\hat{e}_1, \hat{e}_2, \hat{e}_3) end </pre>

Figure 2.3: Parallel ranges.

Ranges are more than a notational convenience, as, in some contexts, they are relatively easy for the compiler to optimize or fuse away. The general idea with regards to fusion of ranges is that the compiler will avoid building intermediate data structures when doing so is unnecessary. To be more specific, in the expression

```
sumP [| sin(real(x)) | x in [| 1 to 100000 |] |]
```

where `sumP` is a sum operator over parallel arrays, we would prefer not to build the 100,000 element parallel array in the binding part of the comprehension only to feed it to the aggregating context in which it appears. In such a case, fusion may allow us to avoid building the array. Fusion is briefly discussed further in Section 4.5.

2.4 Parallel Tuples

Parallel tuples are like ML tuples, but delimited by `(|` and `|)` in ASCII and `(|` and `|)` in the present paper. The subexpressions of a parallel tuple are evaluated in parallel.

syntax	(e_1, \dots, e_n)
static semantics	$\frac{C \vdash e_1 \Rightarrow \tau_1, \dots, e_n \Rightarrow \tau_n}{C \vdash (e_1, \dots, e_n) \Rightarrow \tau_1 * \dots * \tau_n}$
dynamic semantics	$(\hat{e}_1, \dots, \hat{e}_n)$

Figure 2.4: Parallel tuples.

Unlike parallel arrays, parallel tuples may include values of heterogeneous types. Consider the following parallel tuple:

`(| 1, 2.2, true |)`

The type of this value is simply `int * real * bool`; there is no type distinction between tuples and parallel tuples. Parallel tuples can go wherever sequential tuples can go, and as such we can express programs with a high degree of parallelism in a very concise way. Consider the following example:

```
fun choose (n, k) =
  if (n = k) then 1
  else if (k = 1) then n
  else op+ (| choose (n-1, k), choose (n-1, k-1) |)
```

Here the use of parallel tuple delimiters induces, or has the potential to induce, factorially many parallel recursive calls to `choose`.

A parallel tuple is essentially a *fork/join* form; each of its components is evaluated in parallel, and the computation of the whole value blocks until all its subcomputations are complete. In this way, there is a deliberate similarity between the evaluation behavior of Manticore with respect to parallel arrays and parallel tuples. Since tuples

can include values of different types, this gives us a form of heterogeneous parallelism which is not immediately available to us by means of simple parallel arrays.¹

```
val nums = (| thousandthRoot 3.14159, ack (4,2), allPerms "sandwich" |)
```

2.5 Parallel Bindings

The parallel binding form allows the program to launch the evaluation of an expression and continue with program execution in parallel. Parallel bindings use `pval` in place of `val`, as in the following expression:

```
let pval x = f 10 in (x * 10) end
```

As is the case with parallel tuples, there is no type distinction between the parallel binding form and the sequential binding form. In the expression above, for example, if `f 10` and thus `x` and `x * 10` are of type `int`, then the expression as a whole has type `int` just like its sequential analogue.

Parallel bindings enable speculative computation because expressions in parallel bindings whose values are not needed in a given computation may be dynamically cancelled. In order to explain this behavior more precisely, we first need to consider the concept of a *future*.

2.5.1 Futures

The notion of a future comes from the Multilisp programming language [27]. In Multilisp, the expression `(future e)` begins concurrent evaluation of `e` and returns a token value. When the concurrent evaluation of `e` is finished, the resulting value replaces the token. When the value of `e` is needed by some other operation, that

1. One could of course encode similar expressions by means of algebraic datatypes.

operation either takes place immediately if e is done evaluating, or suspends until e is ready. Expressions that require the value of any such future are said to *touch* it.

In Manticore, constructs relating to futures are not available in the surface language. The intermediate languages of the Manticore compiler, however, do contain expressions for computing with futures. Those languages include one introduction form and two elimination forms for futures.² Chapter 4 formalizes the languages in questions are gives corresponding expression forms.

The future introduction form **future** consumes a suspended computation, that is, a function of type $\text{unit} \rightarrow \alpha$, and returns a value of type α **future**. The behavior of such future expressions is essentially the same as the Multisp behavior sketched above.

The first future elimination form **touch** consumes a value of type α **future** and produces a value of type α . If a given future expression is done evaluating, applying **touch** to it yields the resulting value. If the expression is not done evaluating, the application of **touch** will suspend control until the value is ready. Note that if a particular future is touched several times in sequence, control will only ever be suspended on the first touch; the value will necessarily be ready on all subsequent touches.

The second future elimination form **cancel** halts evaluation of a given future. In the expression **cancel** e_1 **in** e_2 , e_1 is a value of type α **future**. (e_2 has some type β which is irrelevant to the cancellation.) When a future is cancelled, whatever resources are devoted to the computation of its value are liberated from their duties as close to instantaneously as possible. A future can be cancelled only when it has been determined that its value will definitely not be needed at any later point in the program.

We do not describe the implementation of Manticore’s future constructs in the present work; we assume they are correctly implemented and available wherever needed.

2. The syntax of these forms in Chapter 4 differs insubstantially from the current presentation.

Next we describe two semantics for expressions including parallel bindings: the “parallel semantics,” which defines the evaluation of parallel bindings in a running program, and the sequential semantics similar to those given above.

2.5.2 Parallel Evaluation Semantics

The parallel evaluation semantics of parallel binding forms makes use of futures and a form of substitution that we call *canceling substitution*. Canceling substitution is defined in Figure 2.5. The essence of canceling substitution is to cancel those futures whose values are known not to be needed in a given expression. We identify candidates for cancellation as variables that are bound but not live in a given expression.

The following example gives the intuition of the parallel semantics of these forms. Consider the expression

```
let pval y = f x
in
  if p z then y else 0
end
```

Note that if the predicate *p* is true for *z*, then the conditional expression will evaluate to 0; *y* will not be needed. If possible, we would like to cancel the superfluous evaluation of *y*. Therefore we rewrite the expression into one with explicit future, touch and cancel forms as follows.

```
let val yf = future (fn () => f x)
in
  if p z then (touch yf) else (cancel yf in 0)
end
```

This rewriting can be generalized to give the parallel evaluation semantics of single-variable parallel bindings as follows:

```
let pval p = e in e' end
```

$$\begin{aligned}
[x \mapsto e_1]_c e_2 &\rightarrow \mathbf{cancel} \ x \ \mathbf{in} \ e_2 && \text{when } x \text{ is uncanceled and not live in } e_2 \\
&\rightarrow [x \mapsto e_1] e_2 && \text{otherwise}
\end{aligned}$$

Figure 2.5: Canceling substitution. Canceling substitution is distinguished from normal substitution by a subscript c on the right bracket of the mapping.

```

—→
let val f = future (case e of p => xs | _ => raise Bind)
                      (* where xs is V(p) *)
in
  [ x ↦ #x (touch f)] e' for x ∈ xs
end

```

2.5.3 Sequential Semantics

In order to define the sequential semantics for the parallel binding form, we first introduce the notion of an α trap for the proper treatment of exceptions. The constructors **Value** and **Exn** are introduction forms for traps, and **release** is the corresponding elimination form.

```

datatype  $\alpha$  trap = Value of  $\alpha$  | Exn of exn
fun release t = case t of Value v => v | Exn e => raise e

```

We use **traps** to sequentialize parallel bindings such that exceptions occurring in speculative computations are only raised in the event that the result of the speculative computation is demanded. Figure 2.6 includes the definition of the sequential semantics.

2.5.4 Remarks

The difference between **val** and **pval** can be characterized as follows. In the case of **val**, control “lingers” at the right hand side of the binding site until computation has

syntax	<code>let pval p = e_1 in e_2</code>
static semantics	$\frac{C \vdash p \Rightarrow (VE, \tau_1) \quad C \vdash e_1 \Rightarrow \tau_1 \quad C + VE \vdash e_2 \Rightarrow \tau_2}{C \vdash \text{let pval } p = e_1 \text{ in } e_2 \Rightarrow \tau_2}$
dynamic semantics	<pre> let val t = Value \hat{e}_1 (* t a fresh id *) handle ex => Exn ex in [let p = release t in x / x]^{$x \in V(p)$} \hat{e}_2 end </pre>

Figure 2.6: Parallel bindings. In the dynamic semantics, the superscript annotation on the substitution means “for all variables x that appear in the pattern p .”

completed. By contrast, with a parallel binding, the right hand side’s computation is spawned, and control immediately flows past it. If and when the result of that computation is needed at a later point, control will wait for the result before moving on.

The combination of parallel bindings and parallel tuples introduces a variety of similar but slightly different patterns of parallelism, as depicted in Figure 2.7.

In the expressions in the bottom row of the table in Figure 2.7, we do not expect to observe any non-trivial difference in execution time. On the lower left, we wait for the evaluations of `f 1` and `g 1` to complete at the binding site before moving on in the program, while on the lower right we move past the binding site and wait around the application of `+` for the two operands to finish their evaluations. In either case, we initiate two parallel evaluations and wait for both to complete, albeit at different points in the code.

The story is different when an expression is such that **not sure how to say this** parallel computations are effectively speculative and their results are sometimes can-

	<i>control lingers at binding</i>	<i>control flows past binding</i>
<i>sequen- tial</i>	<pre> let val (x,y) = (f 1, g 1) in foo x + bar y end </pre>	<pre> let pval (x,y) = (f 1, g 1) in foo x + bar y end </pre>
<i>paral- lel</i>	<pre> let val (x,y) = (f 1, g 1) in foo x + bar y end </pre>	<pre> let pval (x,y) = (f 1, g 1) in foo x + bar y end </pre>

Figure 2.7: The interactions of parallel and sequential bindings and tuples.

called. This is the case with expressions where parallel bindings appear in sequence. Let us consider

```

let pval x = f 1
    pval y = g 2
    pval z = h 3
in
    if (x > 0) then y else z
end

```

In this expression, the evaluations of **x**, **y** and **z** are launched, and control flows to the **if** expression. At the **if**, control waits for the value **x** in order to evaluate the condition **x > 0** can be evaluated. If the condition is true, **z** is cancelled and **y** is synchronized on and returned; the converse occurs if the condition is false.

We can characterize the time required to compute this expression as follows. If the evaluation of **x**, **y** and **z** takes time t_x , t_y and t_z , then the execution time can be bounded above by

$$\max(t_x, t_y, t_z) + \omega + k$$

where ω is some bound on the overhead incurred by parallelism and k is the sequential overhead associated with `let`, `if`, etc. Compare this quantity to the bound on the sequential counterpart to the expression above (with `vals` in place of `pvals`), which requires time

$$t_x + t_y + t_z + k$$

We now consider under what constraints the parallel expression executes in less time than its sequential counterpart. The inequality

$$\max(t_x, t_y, t_z) + \omega + k < t_x + t_y + t_z + k$$

holds when

$$\omega < t_x + t_y + t_z - \max(t_x, t_y, t_z)$$

which we expect to be true in many cases, especially when the t_i are expensive.

2.6 Parallel Choice

Parallel choice gives us a way to express nondeterminism in our programs. The parallel choice operator is denoted by infix `<?>`, as in $e <?> e'$. There are a variety of contexts in which this construct is useful. In certain situations, either of two possible results will do equally well:

```
val solution = solve thisWay <?> solve thatWay
```

The n -queens problem, for example, has this property. In such cases, having multiple possible solutions evaluating together in parallel might lead to significant performance gains.

Consider the expression `let x = e <?> e'`. The variable `x` is bound to either `e` or `e'`; the semantics says nothing about which. The dynamic behavior of this expression, on the other hand, can be characterized as follows: `x` is bound to whichever of `e` or `e'` finishes first, assuming one of them does. Furthermore, once one of the expressions

$$\begin{array}{c}
\frac{}{\text{amb}(v, e) \mapsto v} \qquad \frac{}{\text{amb}(e, v) \mapsto v} \\
\\
\frac{}{\text{amb}(\text{raise Undef}, e) \mapsto e} \qquad \frac{}{\text{amb}(e, \text{raise Undef}) \mapsto e} \\
\\
\frac{e_1 \mapsto e'_1}{\text{amb}(e_1, e_2) \mapsto \text{amb}(e'_1, e_2)} \qquad \frac{e_2 \mapsto e'_2}{\text{amb}(e_1, e_2) \mapsto \text{amb}(e_1, e'_2)}
\end{array}$$

Figure 2.8: Small-step evaluation rules for **amb**.

completes, the other is canceled so as not consume any more resources. As these are characterizations of parallel dynamic behavior, they do not manifest themselves in the static and sequential semantics given in Figure 2.9.

The sequential semantics of $\langle ? \rangle$ are given by the primitive **amb**, which we assume to be present in our extended-SML target language. McCarthy defines an ambiguity operator [21] as “a basic ambiguity operator $\text{amb}(x, y)$ whose possible values are x and y when both are defined: otherwise, whichever is defined.” We borrow the essence of this definition for **amb**, for which we give dynamic semantics in Figure 2.8. There is no order of evaluation imposed on the last pair of rules, so a parallel choice expression under evaluation takes a step on either of its subexpressions. Our semantics makes no assumption about the scheme used to determine which evaluation step is taken; it might or might not be random, or “fair.” In a sequential implementation, however, stepping would likely alternate between the left and right subexpressions to simulate equal probability. In a parallel implementation, the two subexpressions are evaluated in parallel and must know about one another’s progress such that they may be cancelled at appropriate moment. We defer the discussion of the implementation of this behavior in Maticore to future work.

syntax	$e_1 <?> e_2$
static semantics	$\frac{C \vdash e_1 \Rightarrow \tau \quad C \vdash e_2 \Rightarrow \tau}{C \vdash e_1 <?> e_2 \Rightarrow \tau}$
dynamic semantics	$\mathbf{amb}(\hat{e}_1, \hat{e}_2)$

Figure 2.9: Parallel choice.

syntax	$\mathbf{raise\ Undefined}$
static semantics	$\overline{C \vdash \mathbf{raise\ Undefined} \Rightarrow \alpha}$
dynamic semantics	$\mathbf{raise\ Undefined}$

Figure 2.10: Raising exceptions.

2.7 Exceptions

We present simple **raise** and **handle** forms in Figures 2.10 and 2.11. Exceptions and exception handlers are absent from NESL, Data Parallel Haskell, and pH. It is straightforward to imagine uses of these forms in parallel functional programs, but they have thus far been left out of the formalisms in the literature. In our inclusion of exceptions in Manticore, we have encountered some subtleties in their interaction with the flattening transformation (see Chapter 4); the relevant points are addressed in detail in Section 4.3.

syntax	$e_1 \text{ handle } e_2$
static semantics	$\frac{C \vdash e_1 \Rightarrow \tau, e_2 \Rightarrow \tau}{C \vdash e_1 \text{ handle } e_2 \Rightarrow \tau}$
dynamic semantics	$\hat{e}_1 \text{ handle } _ \Rightarrow \hat{e}_2$

Figure 2.11: Handling exceptions.

2.8 Remarks

List-based functional languages traditionally include the combinators `map` and `filter`, which build new lists in left-to-right order (in a strict language) while traversing the lists to which they are applied. Both `map` and `filter` have natural parallel analogues. `MapP` applies a function to each member of an array in parallel, whereas `filterP` discards those elements of an array that do not satisfy a given predicate. Both functions are succinctly defined in terms of comprehensions:

```
(* mapP : ( $\alpha \rightarrow \beta$ ) ->  $\alpha$  parray ->  $\beta$  parray *)
fun mapP f xs = [| f x | x in xs |]
```

```
(* filterP : ( $\alpha \rightarrow \text{bool}$ ) ->  $\alpha$  parray ->  $\alpha$  parray *)
fun filterP pred xs = [| x | x in xs where pred x |]
```

Manticore also includes a variety of primitives on its parallel constructs, some of whose meanings are given informally here:

- The operators `##`, `@@`, and `!` are length, concatenation and subscript operators for parallel arrays.
- The function `flatten` consumes a parallel array of parallel arrays and produces the concatenation of the nested parallel arrays.

- The function `indices` produces the range of integers matching the indices of a given array; the related function `withIndices` produces a list of index, value pairs.

```
(* indices :  $\alpha$  parray -> int parray *)
fun indices xs = [| 0 to (##xs - 1) |]

(* withIndices :  $\alpha$  parray -> (int *  $\alpha$ ) parray *)
fun withIndices xs = [| (i, x) | i in indices xs, x in xs |]
```

Many utility functions can be conveniently expressed in terms of comprehensions and ranges. The function that produces a parallel array of `n` copies of some constant `k` (known in the literature as `dist`) can be written as follows:

```
(* dist :  $\alpha$  * int ->  $\alpha$  parray *)
fun dist (k, n) = [| k | i in [| 1 to n |] |]
```

Along similar lines, the function that produces a parallel array of some function applied to successive integers (a similar function is called `tabulate` in SML) can be written

```
(* tabulateP : int * (int ->  $\alpha$ ) ->  $\alpha$  parray *)
fun tabulateP (n, f) = [| f i | i in [| 0 to (n-1) |] |]
```

We can also write a two-dimensional tabulation function to build matrix-like structures. Note the use of a nested comprehension.

```
(* tab2P : int * int * (int * int ->  $\alpha$ ) ->  $\alpha$  parray parray *)
fun tab2P (m, n, f) =
  [| [| f (i,j) | i in [| 0 to m-1 |] |] | j in [| 0 to n-1 |] |]
```

Functions for grabbing chunks of arrays are conveniently built out of the same parts.

```
(* slice :  $\alpha$  parray * int * int ->  $\alpha$  parray *)
```

```
(* produces the slice of an array on the interval [start, start+len) *)  
fun slice (a, start, len) = [| a ! i | i in [| start to (start+len-1) |] |]  
  
(* take, drop : int *  $\alpha$  parray ->  $\alpha$  parray *)  
fun take (n, a) = slice (a, 0, n)  
fun drop (n, a) = slice (a, n, ##a)
```

We will refer to these building-block level functions in the examples that follow.

CHAPTER 3

APPLICATIONS OF MANTICORE

This chapter develops a selection of applications in Manticore as a demonstration of its use in various familiar computational settings. These examples should give the reader a sense of the language from a programmer’s point of view.

3.1 Parallel Tuples for Parallel Searches

As an example of the use of the parallel let form, consider the following application. We assume we are writing an engine for an I/O-bound MapQuest-like application that can search for road maps for routes. We assume the existence of abstractions `roadmap`, `loc`, and `route`, which we employ freely in the following. The task at hand is to determine a scenic route or a fast route for a given driver, who may or may not have a taste for scenery. The driver has been asked to indicate her “scenery tolerance” as the time in minutes she is willing to sacrifice in the interest of scenery.

```
(* fastest, prettiest : roadmap -> loc * loc -> route *)
fun fastest    m (a, b) = (* implementation omitted *)
fun prettiest m (a, b) = (* implementation omitted *)

(* chooseRoute : int * route * route -> route *)
fun chooseRoute (t, fr, sr) = if time(fr) - time(sr) <= t then sr else fr

val r = let pval fr = fastest USA (Bos, Phil)
          pval sr = prettiest USA (Bos, Phil)
          val st (* scenery tolerance *) = getInt ()
```

```

in
  if st > 0 then chooseRoute (st, fr, sr)
  else fr
end

```

While the program is waiting for user to provide this number, the program begins computing both fast and scenic routes (**fr** and **sr** below) in parallel. In the event that the driver has no taste for scenery, that is, when her scenery tolerance is 0, the program need not expend any further resources on computation of the scenic route. As such, the scenic route computation is cancelled if it is still in progress (see Section 2.5 for the details of this mechanism). If, on the other hand, the driver has a positive scenery tolerance, that is, the driver is willing to consider the scenic route, the program completes computation of both the fast and scenic routes in order to choose the appropriate one.

3.2 Some Operations on Matrices

3.2.1 Dense Matrix Transposition

For this example, we assume a dense representation of matrices. The transposition A^T of an $m \times n$ matrix A is an $n \times m$ matrix where $A_{j,i}^T = A_{i,j}$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$. The Manticore program to compute the transposition of a matrix is a straightforward transcription of its mathematical definition.

```

type mat = real parray parray (* we assume mats are not "jagged" *)

(* transpose : mat -> mat *)
fun transpose A =
  let val is = indices A      (* row indices *)
      val js = indices (A!0) (* col indices *)
  in

```

```

    [| [| A!j!i | j in js |] | i in is |]
end

```

3.2.2 Sparse Vector Matrix Multiplication

Sparse vector matrix multiplication is a classic motivating example for nested data parallel programming. The following formulation of the algorithm was originally presented in Blelloch [3] and was recently formulated in Data Parallel Haskell in Chakravarty et al. [10].

```

type vec = real parray
type sparse_vec = (int * real) parray
type sparse_mat = sparse_vec parray

(* dotp: sparse_vec * vec -> real *)
fun dotp (sv, v) = sumP [| x * (v!i) | (i,x) in sv |]

(* smvm : sparse_mat * vec -> vec *)
fun smvm (sm, v) = [| dotp (row, x) | row in sm |]

```

3.3 Quicksort

Quicksort is a common example in the nested data parallelism literature. We implement quicksort in Manticore as the function `qs`. In `qs`, the argument array `ns` is partitioned into elements less than, equal to, and greater than some given pivot element in parallel comprehensions. Each comprehension is in effect a filter running in parallel on the input data. Those comprehensions are in turn evaluated—and quicksorted—in parallel inside a parallel tuple.

This parallel formulation of quicksort differs from its sequential formulation only by a few parallel delimiters. The example is therefore a demonstration of the principle

that divide-and-conquer algorithms are in general well-suited to parallelization.

```
(* qs : int parray -> int parray *)
fun qs ns =
  if ## ns < 2 then ns
  else
    let val piv = ns ! 0
        fun f relop = [| n | n in ns where relop (n, piv) |]
        val (slt, eq, sgt) = (| qs (f op<), f op=, qs (f op>) |)
    in
      slt @@ eq @@ sgt
    end
```

3.4 Selecting the Distinct Elements of an Array

In solving any number of interesting problems, it is useful to eliminate duplicates from a collection of elements. The Haskell Prelude provides the function `nub` for this purpose, and the Unix command `uniq` eliminates adjacent duplicates from an already-sorted collection. We follow the latter example in sorting a list, then filtering out the duplicates.

We assume the existence of a built-in function `reduceByP` to reduce parallel arrays with a given associative operator and a given identity element.

```
val reduceByP : ( $\alpha$  *  $\alpha$  ->  $\alpha$ ) ->  $\alpha$  ->  $\alpha$  parray ->  $\alpha$ 
```

One can think of `reduceByP` as an abstract parallel sum operator, and furthermore can be thought of as using a parallel tree-like reduction strategy to run in logarithmic time. There is more detail on logarithmic parallel sums in Section 4.6.

We (implicitly) define a polymorphic higher-order function `sortByP`, which consumes an abstract less-than-or-equal-to relation along with a parallel array of elements to be sorted. We omit the definition of `sortByP` but the reader can infer it to be `qs`

from Section 3.3 with appropriate modifications. The function `uniqP` is subsequently defined as an abstract parallel sum on lists of sorted distinct elements. As a final necessary step we appeal to a function `PArray.fromList` which we assume to be part of the language's basis.

```

(* sortByP : ( $\alpha$  *  $\alpha$  -> bool) ->  $\alpha$  parray ->  $\alpha$  parray *)
fun sortByP ... (* implementation omitted *)

(* last :  $\alpha$  list ->  $\alpha$  *)
fun last [] = raise Undef
  | last xs = hd (rev xs)

(* liaison : ( $\alpha$  *  $\alpha$  -> bool) ->  $\alpha$  list *  $\alpha$  list ->  $\alpha$  list *)
(* pre: both arguments are sorted and free of duplicates *)
(* pre: the last element of the first argument is not greater than
      the first element of the second argument *)
(* ex: liaison op= ([1,2], [2,3]) = [1,2,3] *)
fun liaison _ ([], ys) = ys
  | liaison _ (xs, []) = xs
  | liaison eq (xs, y::ys) = if eq (last x, y) then xs @ ys else xs @ y::ys

(* uniqP : ( $\alpha$  *  $\alpha$  -> bool) ->  $\alpha$  parray ->  $\alpha$  parray *)
fun uniqP le xs =
  let fun eq (a,b) = le(a,b) andalso le(b,a)
      val sortedSingletons = [| [x] | x in (sortByP ls xs) |]
  in
    PArray.fromList (reduceByP (liaison eq) [] sortedSingletons)
  end

```

3.5 Image Transformations

In the following examples, we assume images are represented as dense matrices of colors, where each color is a triple of integers.

```
type rgb = int * int * int
type img = rgb parray parray
```

Many interesting transformations of images can be done in parallel. An RGB-wise negation or an image can be computed fully in parallel, as each pixel may be negated independently of all others:

```
(* negImg : img -> img *)
fun negImg img = [| [| (255-r,255-g,255-b) | (r,g,b) in row |] | row in img |]
```

A function to extract only the red values of an image has the same shape:

```
(* reds : img -> img *)
fun reds img = [| [| r | (r,_,_) in row |] | row in img |]
```

We can express the entire family of such element-by-element transformations with the following higher-order function:

```
(* trImg : ( $\alpha \rightarrow \beta$ ) ->  $\alpha$  parray parray ->  $\beta$  parray parray *)
fun trImg tr img = [| [| tr pix | pix in row |] | row in img |]
```

Note the function’s principal type is sufficiently general to apply to any array-of-arrays structure; it is in fact a parallel `map` combinator for two-dimensional arrays.

3.5.1 Minification

We now consider an algorithm for minification. To *minify* (as opposed to “magnify”) an image is to shrink the image to a smaller dimension. By inspection, in reducing one

`img = rgb parray parray` to another, some information about the original pixels must be lost, since there are strictly fewer pixels in a smaller image than a larger one, and the encoding of colors is no more terse in the former than the latter.

Naïvely, one could simply discard pixels in the process of minification, but such methods leave blatant artifacts in many cases. We can improve minification as follows. We consider the smaller or target image as a map from its pixels (“target pixels”) to a set of weighted pixels from the source image (“source pixels”). The weights represent, loosely speaking, how much each source pixel should contribute to the color of the target pixel; it is based simply on the geometry of the images in question. We color each target pixel with the weighted average color of those source pixels in the set onto which it maps.

The definition of `minify` follows, along with the definition of a few necessary auxiliary functions. Any auxiliary functions whose definitions are not explicitly given should be clear from their names.

```
(* rgbPlus : rgb * rgb -> rgb *)
fun rgbPlus ((r1,g1,b1),(r2,g2,b2)) = (r1+r2,g1+g2,b1+b2)

(* scaleRGB : real * rgb -> rgb *)
fun scaleRGB (x,(r,g,b)) = (int(x*r),int(x*g),int(x*b))

(* minify : img * int * int -> img *)
fun minify (img, w', h') =
  let val rgbSumP = reduceByP rgbPlus (0,0,0)
      val (w, h)  = (width img, height img)
      fun newpix (s, t) =
        let val i = max (0, floor ((w*s) - 0.5))
            val j = max (0, floor ((h*t) - 0.5))
            val  $\alpha$  = fracPart ((w*s) - 0.5)
            val  $\beta$   = fracPart ((h*t) - 0.5)
        in
```

```

      rgbSumP [| scaleRGB ((1.0- $\alpha$ )*(1.0- $\beta$ ), img!i!j),
                  scaleRGB ( $\alpha$ *(1.0- $\beta$ ), img!i+1!j),
                  scaleRGB ((1.0- $\alpha$ )* $\beta$ , img!i!j+1),
                  scaleRGB ( $\alpha$ * $\beta$ , img!i+1!j+1) |]
    end
  in
    [| [| newpix (real(x) / real(w'), real(y) / real(h'))
        | x in [| 0 to w'-1 |] |] | y in [| 0 to h'-1 |] |]
  end

```

3.5.2 Image Compression

An `img` as described above is a literal enumeration of pixels. A GIF, by contrast, consists of a pair of structures, a color palette mapping indices to `rgb` values, and a two-dimensional map of indices into that palette. Because each element in the “index matrix” (as opposed to the “color matrix”) is a small value (perhaps one byte instead of three), and because images tend to consist of thousands or millions of pixels, this representation can be significantly smaller than its raw counterpart, especially if the image has low color diversity. A real GIF will restrict the size of its palette so as to guarantee a reduction in size; for simplicity’s sake we will ignore this restriction and call our representation a `gif'`. The following types encode the necessary structures:

```

type palette = (int * rgb) parray (* a bijection *)
type gif' = palette * int parray parray

```

We will write a program to convert an `img` to its corresponding `gif'`. We first gather the unique colors in an image, build a palette out of them, and construct an image matrix by mapping each color in the color matrix to its palette index. From Section 3.4 we use `reduceByP` and `uniqP` in our implementation.

```

(* rgbLE : rgb * rgb -> bool *)
fun rgbLE ((r,g,b), (r',g',b')) =
  let fun num (r,g,b) = (r*pow(2,16)) + (g*pow(2,8)) + b

```

```

    in
      (num (r,g,b)) <= (num (r',g',b'))
    end

(* rgbEQ : rgb * rgb -> bool *)
fun rgbEQ ((r,g,b), (r',g',b')) = (r=r') andalso (g=g') andalso (b=b')

(* distinctColors : img -> rgb parray *)
fun distinctColors img = uniqP rgbLE (flatten img)

(* mkGIF' : img -> gif' *)
fun mkGIF' img =
  let val pal = withIndices (distinctColors img)
      fun findIdx c = [| i | (i, c') in pal where rgbEQ (c, c') |] ! 0
  in
    (pal, trImg findIdx img)
  end

```

The corresponding decompression function is encoded as follows:

```

(* unGIF' : gif' -> img *)
fun unGIF' (pal, idxs) =
  let fun findRGB i = [| c | (i', c) in pal where i = i' |] ! 0
  in
    trImg findRGB idxs
  end

```

Note that both `findIdx` and `findRGB` depend on the fact that a palette is a bijection. We could check this property in the code (with, for example, dynamic checks), but here we trust the correctness of the behavior of `uniqP` to ensure it.

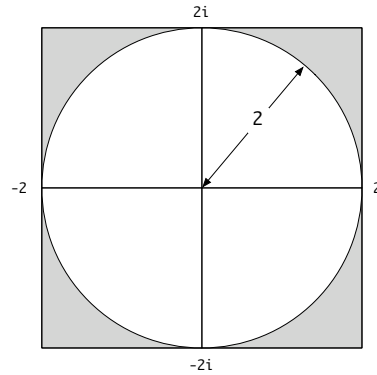


Figure 3.1: The slice of the complex plane extending two units from 0 on both axes. All points in the shaded regions have modulus > 2 .

3.5.3 The Mandelbrot Set

Parallel comprehensions allow us to build images derived from application of functions very concisely. The Mandelbrot set is an especially interesting image, and it can be constructed in embarrassingly parallel fashion.

We assume the existence of a function `color : int * real * real -> rgb` to compute the color of a pixel in a given location in the complex plane. Its integer parameter is an upper bound on the number of iterations the function is to compute; the reals are the two components of a complex number. The implementation of `color` is omitted for simplicity, but we assume it exploits the following property of the Mandelbrot set: for any $z \in \mathbb{C}$ where $|z| > 2$, z is not in the Mandelbrot set. Note that all points in the shaded region in Figure 3.1 have modulus > 2 . Values with sufficiently large moduli can be immediately designated not in the set; this corresponds to a short-circuit in any implementation. We note that some computations of `color` will take much longer than others, since many inputs will short-circuit the number of iterations.

For the purposes of this example we will compute on the slice of the complex plane centered on the origin and bounded by -2 and 2 in both directions, where each pixel will represent a square of side length `.02`; these choices are of course arbitrary and easily modified.

```

(* color : int * real * real -> rgb *) (* implementation omitted *)

(* axis : real parray *)
val axis = [| real(n) * 0.02 | n in [| ~100 to 100 |] |]

(* mset : int -> img *)
fun mset i = [| [| color(i,re,im) | re in axis |] | im in axis |]

```

This application is an important example since the amount of time required to compute `color` will vary significantly according to its argument. Therefore a naïve equal distribution of work—that is, one where each processing element is given the same number of points for which to compute `color`—may result in suboptimal performance. This program will benefit from both flattening and fusion (Chapter 4) in the compiler.

3.6 The Barnes-Hut force calculation algorithm

The Barnes-Hut force calculation algorithm [2] is an $O(n \log n)$ method of calculating the force enacted on one another by n bodies in space. The algorithm constructs a hierarchical tree of cells, each of which contains one massive body. This tree manifests itself as an “oct tree” in three dimensions or a “quad tree” in two dimensions; the latter is presented here for simplicity. Once the tree is constructed, the force on a given body is calculated individually for “near” bodies (where nearness is determined by a constant threshold) or from clusters of bodies if the cluster is sufficiently distant.

The following implementation is based on the similar implementation in Keller’s dissertation [16]. The construction of the spatial tree—a divide-and-conquer process reminiscent of quicksort—is highly amenable to parallelization: parallel comprehensions are freely employed in `bhTree` and its auxiliary `splitCs` below.

```

type area = {llx:real, lly:real, urx:real, ury:real}
type vec  = {x:real, y:real}
type cen  = {x:real, y:real, m:real}

```



```

datatype tree = T of cen * tree list

(* center : area -> vec *)      (* impl. omitted *)
(* cut : area -> area parray *) (* impl. omitted *)

(* splitCs : vec * cen parray -> cen parray *)
fun splitCs ({x=fx,y=fy}, cs) =
  let val (le,gt) = (op<=,op>)
      fun filt (xop, yop) =
          [| c | c as {m,x,y} in cs
             where xop(x,fx) andalso yop(y,fy) |]
      in
          [| filt(le,le), filt(gt,le), filt(le,gt), filt(gt,gt) |]
      end

(* bhTree : area * cen parray -> tree *)
fun bhTree (a, cs) =
  if ## cs = 1 then
    T (hd cs, [])
  else
    let val (a1, a2, a3, a4) = cut a
        val (cs1, cs2, cs3, cs4) = splitCs(center a, cs)
        val areaCens = [| (a,cs) | a in cut(a),
                           cs in splitCs(center(a), cs)
                           where ## cs > 0 |]
        val subtrees = [| bhTree(a,cs) | (a,cs) in areaCens |]
        val cd = centroid [| cs | Tree(cs, ts) in subtrees |]
      in
          T (cd, subtrees)
      end

(* accels : tree * real * cen parray -> vec parray *)
fun accels (tree as T (crd, subtrees), len, cs) =

```

```

if ## cs = 0 then
  [| {x=0.0, y=0.0} |]
else
  let val (farC, closeC, direct) = splitFarClose (cs, len, crd)
  val farAcs = [| accel(crd,mp) | mp in farC |]
  val closeAcss =
    [| accels(t, len/2.0, closeC | t in subtrees |]
  val closeAcs = [| superimp a | a in transpose(closeAcss) |]
in
  combine (farAcs, closeAcs, direct)
end

```

3.7 Google's MapReduce

Google's MapReduce [12] is an abstract functional program design for parallel distributed data processing. MapReduce programs operate on collections of key, value pairs. A particular program specifies two functions, a “map” and a “reduce.” The former function transforms a given pair (k, v) to some (k', v') ; the latter function aggregates batches of values v' whose keys k' are equal. Google uses this pattern on its own distributed system to map keywords to web pages, count the frequency of accesses of website, and compute various other interesting results.

Although MapReduce is used in a distributed setting, the design could just as well be utilized on an individual shared-memory machine. The following program demonstrates a realization of the pattern in Manticore.

Note the higher-order function `sortByP` is borrowed from Section 3.4.

```

type 'a ord = 'a * 'a -> bool
type 'a eq  = 'a * 'a -> bool

(* groupBy : 'k eq -> ('k * 'v) parray -> ('k * 'v list) parray *)
fun groupBy keyEq sorted =

```

```

let fun build ([], acc) = acc
    | build ((k,v)::more, []) = build (more, [(k,[v])])
    | build ((k,v)::more, (k',vs)::acc) =
        if keyEq (k, k') then
            build (more, (k',v::vs) :: acc)
        else
            build (more, (k,[v]) :: (k',vs) :: acc)
in
    PArray.fromList (build (rev (List.fromPArray sorted), []))
end

type map      = 'k0 * 'v0 -> 'k1 * 'v1
type reduce = 'k1 * 'v1 parray -> 'k1 * 'v2

(* MapReduce : map * reduce * ('k1 ord)
   -> ('k0 * 'v0) parray -> ('k1 * 'v2) parray *)
fun MapReduce (m,r,lt) input =
    let fun lt'((k,_),(k',_)) = lt(k,k')
        fun eq (p,p') = not(lt'(p,p')) andalso not(lt'(p',p))
        val intermed = [| m (k,v) | (k,v) in input |]
        val sorted   = sortByP lt' intermed
        val grouped  = groupBy eq sorted
    in
        [| r (k1, v1s) | (k1, v1s) in grouped |]
    end
end

```

CHAPTER 4

COMPILING MANTICORE BY PROGRAM TRANSFORMATION

The compilation of Manticore is comprised of many stages. This paper describes the first few stages near the front end of the compiler; later stages are described in other work. These “early compilation” stages are given as a set of translations from one language to another: the source language \mathbb{M} is a pared-down version of the source language as described in the foregoing chapters; the language \mathbb{FM} , for *Flat Manticore*, is a flattened version of \mathbb{M} , such that nested data parallel constructs have been rewritten away; \mathbb{DM} , for *Distributed Manticore*, introduces an abstract encoding of the distribution of computation among multiple processing elements into the language; and \mathbb{TM} , the target language, encodes parallel arrays explicitly as *ropes* and manages the distribution of computation directly.

4.1 The Flattening Transformation: An Overview

What is the flattening transformation, and why do it? The flattening transformation is, in short, a means of adjusting the representation of aggregate data such that it is easily amenable to fast parallel computation. In Manticore, we follow the precedent of NESL and its descendants by flattening nested arrays in the process of compilation. We also “flatten” nested parallel tuples in the compiler, although the flattening in question is of a different stripe; we discuss it in the Section 4.4.

To motivate the flattening transformation, consider the following example. New Yorker cartoons are generally simple line drawings, printed in grayscale. See the example appearing in Figure 4.1. Dot for dot, they are mostly their background

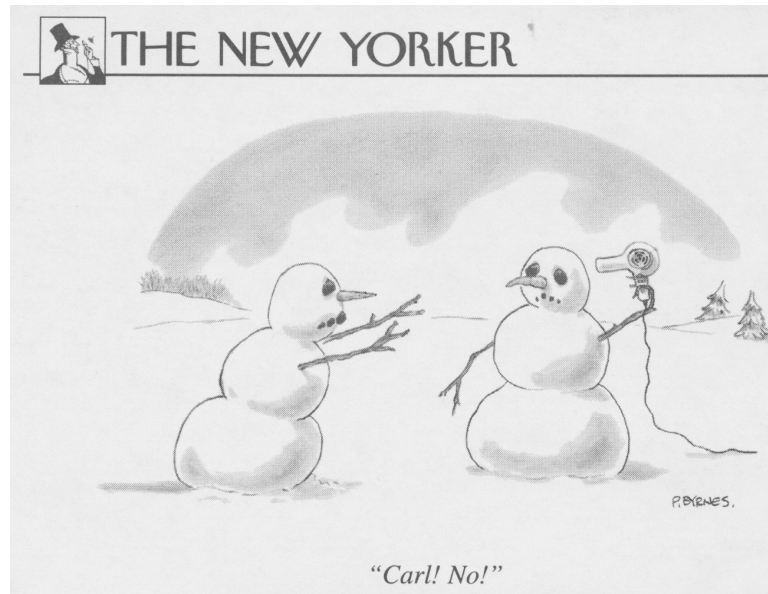


Figure 4.1: A New Yorker cartoon.

color: a glance will reveal they consist of many more background-color pixels than non-background-color-pixels. As such, let us assume a New Yorker cartoon is advantageously represented as a background color plus a sparse matrix of those pixels that deviate from that background color. Each row in such a representation will consist of an array of column, color pairs. The type `nyc` represents a New Yorker cartoon.

```
type rgb    = int * int * int
type nycRow = (int * rgb) parray
type nyc    = rgb * (nycRow parray)
```

Note the second component of a value of type `nyc` is a nested data parallel value, a parallel array of parallel arrays of `int * rgb` pairs.

Observe that if an image is truly grayscale, it is unnecessary to represent each shade of gray with an `rgb` triple, since gray values are precisely those whose red, green, and blue values are equal. Therefore we should be able to represent New Yorker cartoons in the following representation, which is more concise.

```
type gray    = int
```

```

type nycRow' = (int * gray) parray
type nyc'    = gray * (nycRow' parray)

```

Assume that we are now interested in writing a function to convert values of type `nyc` to values of type `nyc'`. We can express this as a nested comprehension as follows:

```

(* gr : rgb -> gray *)
fun gr (r,g,b) = (r+g+b) div 3

(* convertCartoon : nyc -> nyc' *)
fun convertCartoon (bg, sm) =
  (gr bg, [| [| (col, gr c) | (col, c)) in row |] | row in sm |])

```

This is an idiomatic Manticore program and we should expect it to exhibit good performance.

Now imagine this computation executing in parallel on a machine with four processors. We must formulate a strategy for dividing the work among them. As a first approximation, we will consider dividing the sparse matrix row-wise into four equal chunks, and delegating a chunk to each processor for transformation in parallel. The resulting horizontal partition is illustrated and labeled in Figure 4.2. This simple strategy has an obvious flaw. Given our sparse representation of these images, some rows will have more entries than others. Operationally this means that in a row-wise division of an image, some processors might end up with much less work to do than others. In Figure 4.2, it is visually apparent that chunk 2 would contain more data than chunk 4 in a sparse representation, as it contains many more pixels that deviate from the background color. Whichever processor is assigned chunk 4 has too little to do; chunk 2's processor is correspondingly too busy. Thus in this case using equal spatial regions is not a reliable way to divide computation equitably. If we can, we would prefer to divide the *work* into four equal parts, and, as we have seen in this example, the shape of the data structure may or may not help us do that in the most obvious way.

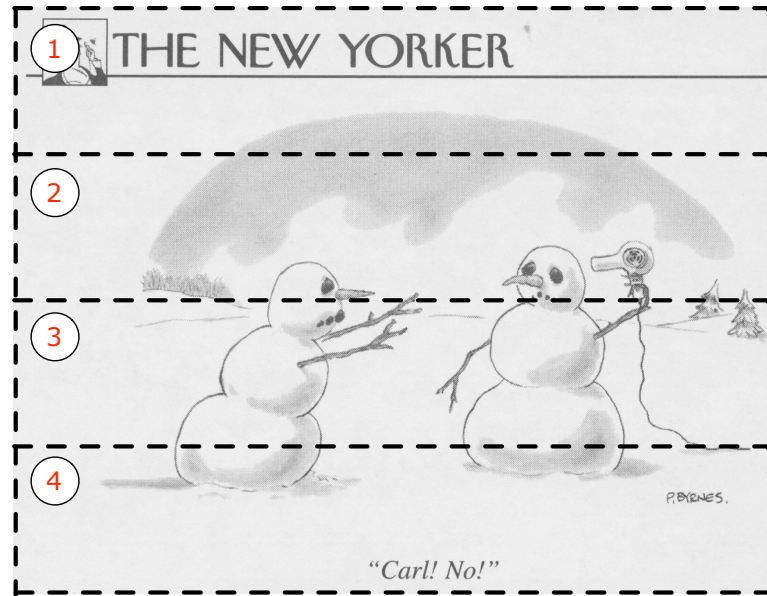


Figure 4.2: The cartoon from Figure 4.1, partitioned.

The flattening transformation provides a simple solution to this and similar problems. Under this transformation, we rearrange a nested structure into a pair of flat structures, one that contains the original data and one that encodes the original nesting structure. For example,

$[[[a, b], [c], [d, e, f]]]$

is transformed into

$([a, b, c, d, e, f], [2, 1, 3])$

where the first element of the pair contains the data a through f , and the second element of the pair contains the lengths of the segments of the original nested structure.

The flattening transformation helps in the cartoon example as follows. The cartoon's sparse matrix appears in the source program as an array of arrays of pairs. We transform the sparse matrix into an array of pairs¹ and an array of segment lengths.

1. In the interest of clarity, we are ignoring here and in the subsequent discussion the standard transformation of arrays of pairs into pairs of arrays.

We call this flat array of pairs `ps`. We then divide `ps` into n arrays of equal length and delegate each to one of n processors for computation. Each processor has, within some small margin, the same amount of work to do: if w is the amount of work required to process one pair, each processor's work is approximately $w \times (|\mathbf{ps}|/n)$.

An execution of `convertCartoon` can be illustrated as follows. A sparse matrix might appear as follows, where each `pn` is a column, color pair:

```
[ [ p1, p2 ], [ p3 ], [ p4, p5, p6, p7 ], [ p8 ] ]
```

After transforming the sparse matrix into an array of pairs and an array of segment lengths, we have

```
( [ p1, p2, p3, p4, p5, p6, p7, p8 ], [ 2, 1, 4, 1 ] )
```

The first component of this pair can easily be divided evenly among processing elements. This simple transformation, scaled up to realistic size, can be reasonably expected to improve performance significantly.

If we say that a flat array such as `[1,2]` has nesting depth 1, a nested array such as `[[[1]], [2]]` has nesting depth 2, and so on, we claim that the application of the flattening transformation removes one level of nesting depth, that is, transforms a nested structure of depth $d > 1$ into a pair of structures of depth $d - 1$ and 1 respectively. As such we will apply the flattening transformation as many times as we need to produce a depth 1 structure.

We may wish to compute not just with individual cartoons, but collections of cartoons, collections of collections of cartoons, and so on to any nesting depth. The flattening transformation will help make all such computations fast, even when there are multiple layers of nesting:

```
type mag = nyc parray
type vol = mag parray
fun convertMag m = [ convertCartoon c | c in m ]
fun convertVol v = [ convertMag m      | m in v ]
```


The flattening transformation is formalized in Section 4.6 below.

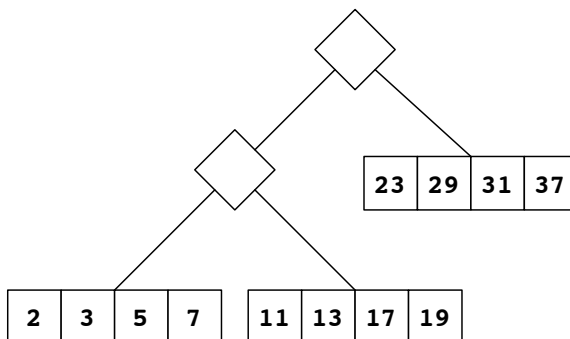
4.2 Ropes

Under the hood, Manticore’s parallel arrays are represented as ropes [6]. Ropes were originally proposed as an immutable alternative to strings supporting fast concatenation, among other things. In their original form, ropes are balanced binary trees with an array of characters at each leaf. The string encoded in such a structure is the sequence of characters at the leaves of the tree, read from left to right. Via this representation, concatenation of strings s_1 and s_2 is the construction, with balancing, of a fresh tree whose left and right subtrees are s_1 and s_2 .

For Manticore, we modify the original rope structure in two ways. First we define a polymorphic rope structure, whose data is not restricted to type `char`. Second, we include a special rope form to represent a caught exception; this is a technical necessity which will be discussed in Section 4.3. This data structure can be specified in Standard ML as follows:

```
datatype  $\alpha$  rope
  = Cat of  $\alpha$  rope *  $\alpha$  rope
  | Leaf of  $\alpha$  vector
  | Raise of exn
```

(In practice this structure will be more complicated; it might, for example, be implemented as a red-black tree to support fast balancing.) Exactly how many items to store in each leaf vector may vary from system to system, and we expect experience to help us learn to tune this parameter. For the purposes of presentation in the present work, we will assume the size of the vector at each leaf is four. This is an unrealistically small number, but it makes for tractable illustrations. For example, we can depict the parallel array of the first twelve prime numbers as a rope with three leaves:



Representing parallel arrays as ropes confers a variety of advantages. Appending arrays is fast; it is simply the generalization of string concatenation as described above. Furthermore, the gathering of data into vectors at rope leaves naturally encodes a favorable parallel execution of certain programs. For example, a computation that maps a function over each element of a parallel array can do so in parallel for all leaves, and in sequence at each individual leaf. This delegation of work is described formally in Section 4.6, but the intuition is as follows. Consider the following example, which uses the same parallel array of twelve primes pictured above.

```
let val primes = [| 2,3,5,7,11,13,15,17,23,29,31,37 |]
  fun sub1 n = n - 1
in
  [| sub1 p | p in primes |]
end
```

Let us assume it is inefficient to compute all twelve function applications in parallel with one another, that is, that such a parallel execution would be so fine-grained as to be inefficient [23]. The shape of the rope representation of `primes` itself suggests a simple solution: assign one processing element to each leaf, and map `sub1` in sequence over the values at each leaf. In this setting, adjusting the size of the vectors at leaves becomes a means of adjusting the granularity of the parallelism in a given execution.²

2. This implementation of parallel mapping bears a relationship to the `parmap-interval` in Mohr et al.’s lazy task creation paper [23]. Our solution encodes a hard decomposition of the computational work in the data structure itself, whereas their `parmap-interval`

4.3 Exceptions

In order to preserve Manticore’s sequential semantics, we need to give special consideration to exceptions. In the presence of parallelism, it might be the case that multiple exceptions are raised concurrently in a given parallel array, comprehension or tuple. To be faithful to our sequential semantics, we need to ensure that the “leftmost” exception—that is, the one that would be raised first in strict sequential evaluation—is in fact the one raised in Manticore. With respect to the handling of exceptions, certain troublesome cases can arise when working with flattened structures. It might also be the case that under fusion, what is semantically a nonterminating computation could terminate with an exception. Each of these scenarios is considered in turn.

4.3.1 Raising the Leftmost Exception

For the following examples, we assume we have Ackermann’s function, which takes a very long time to compute even on small inputs, and a function that raises an exception quickly.

```
fun ack(m,n) = if m = 0 then n+1
               else if m>0 andalso n=0 then ack(m-1, 1)
               else ack(m-1, ack(m,n-1))
```

```
fun ack'(m,n) = let val a = ack(m,n)
                 in  if a > 12 then raise TooBig
                 else a
                 end
```

```
fun intsqrt n = if n<0 then raise Negative
```

operates on a traditional array, saturating n processing elements with roughly $1/n$ th of the work each by computing with array indices.

```
else (* implementation *)
```

Both elements of the following parallel array will raise exceptions on evaluation.

```
[| ack'(4,3), intsqrt(~3) |]
```

The value of `ack(4,3)` is roughly a twenty thousand digit number, so it’s certainly bigger than 12 and will trigger the exception `TooBig`. In a sequential implementation, these expressions would evaluate left to right, and `TooBig` would be raised after a long computation. In a parallel setting, `Negative` will be raised much sooner than `TooBig`, but the computation as a whole must wait for the completion of the evaluation on the left to respect the sequential semantics.

Our compiler’s representation of ropes includes a special form `Raise` that exists to address this issue. Recall the simple rope datatype from Section 4.2, whose variants are `Cat`, `Leaf` and `Raise`. We define a smart constructor `mkCat` for ropes as follows:

```
fun mkCat (Raise e, _) = Raise e
  | mkCat (_, Raise e) = Raise e
  | mkCat (r1, r2) = compactAndBalance (r1, r2)
                        (* implementation unspecified *)
```

(This definition appears in context in Figure A.1 below.) This function encodes the semantic preference for “left exceptions” via pattern matching.

This approach has the advantage of being simple to implement and easy to reason about. On the other hand, it is necessary for all functions that operate on ropes to consider the `Raise` form. One of many effects of this design is that `mkCat` must always be used in favor of direct injection into the rope type with `Cat`. It is inefficient to account for exceptions in contexts where they will never be raised. We may design our type system to distinguish those computations that might raise exceptions from those that definitely will not, in which case we could construct lighter-weight rope structures—ropes with no `Raise` form—when possible. (The present work does not

include a design of such a type system.) For the time being, by including a **Raise** form in our ropes and using ropes to represent parallel arrays, we are able to compute with exceptions as specified by our semantics in a straightforward way.

4.3.2 Handling Flattened Structures

Consider the following expression.

```
let f(n) = if (n<0) then f(n) else (6 div n)
    nss = [[ [ 1, 2, 3 ], [ 0, ~1, 1 ], [ 1, 6 ] ]
in [[ [ f(n) | n in ns ] handle [ ] | ns in nss ]
```

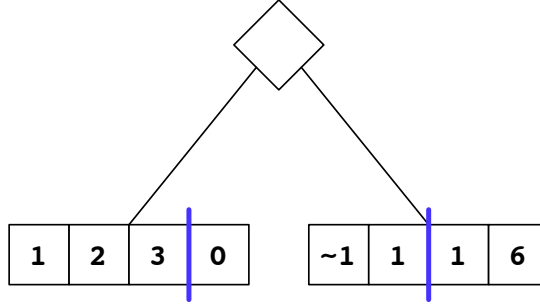
We assume any expression `n div 0` raises the exception **Undef**. Therefore the function `f` returns an integer on positive inputs, raises **Undef** on zero, and loops forever on negative inputs.

By inspection, we expect this expression to evaluate to

```
[[ [ 6, 3, 2 ], [ ], [ 6, 1 ] ]]
```

The point of interest here is that `f` will raise an exception when applied to 0, thereby preventing the computation from disappearing into an infinite loop. Thus the whole parallel comprehension containing 0 should defer to its handler and evaluate to `[[]]` as defined by the sequential semantics.

This expression illustrates a sensitive point in the interaction between exception handling and flattening. Our flattening transformation produces pairs of arrays, with the first pair a flat array of data represented as a rope (discussed above briefly in Section 4.2). The nested array `nss` above will flatten into a pair consisting of the rope illustrated here and the segment descriptor `[[3, 3, 2]]`:



The illustration shows segment boundaries as a visual aid only: these boundaries are not directly present in the rope structure. In other circumstances—namely when exceptions were not involved in the computation—we would simply map `f` over the leaves in parallel and over the leaf vectors in sequence, as outlined above. Let us consider what would happen at each leaf in the present case. On the left, an exception is raised upon encountering 0. On the right, the computation disappears into an infinite loop at `-1`. On the left, we are tempted to annihilate the whole leaf-level computation, but then we lose the results of computations we still care about, namely `6 div 1`, `2` and `3` respectively. On the right, we have a dual problem in that we never reach the computations to the right of the `-1`.

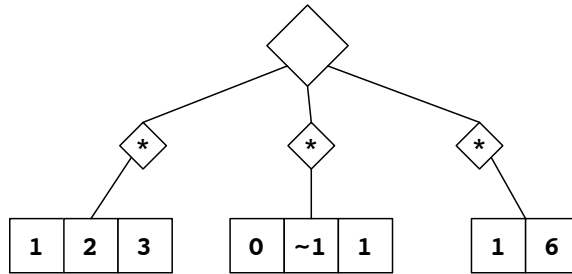
These problems arise because in flattening `nss` we have taken away information that is essential to the proper behavior of the handler. We still possess the segment information in the rope’s concomitant segment descriptor, but it is not available locally.

To describe our solution to this problem, let us first identify the set of parallel comprehensions that potentially exhibit this problematic behavior. Define *hard-to-handle* expressions to be those expressions

```
[| e handle e' | xs in xss |]
```

where `e` has type `τ array` for some `τ`. Our solution is based on an alternative representation of arrays hard-to-handle comprehensions. We can represent hard-to-handle arrays as ropes of ropes, which is to say not-fully-flattened structures. Such a repre-

sensation of `nss` is as follows:



We will call these structures *rumpled* as they have not been entirely flattened. These ropes of ropes are not susceptible to the problems above; whatever handling needs to take place can take place at the interior nodes (marked with asterisks in the picture).

Our strategy for evaluating hard-to-handle expressions is this. Attempt to perform the computation, without any exception handling, on the flattened structure. If no exceptions are raised, then we are finished. If an exception is raised, cancel the evaluation of the expression, build a rumpled structure with handlers at the appropriate interior nodes, and evaluate the expression again. We are guaranteed to handle exceptions properly under the rumpled representation, so when the second evaluation finishes we are done.

It is only when we already know that an exception is raised in an expression that we change over to a rumpled representation. This strategy is based on the assumption that the pattern captured in a hard-to-handle expression is infrequently, if ever, employed in practice, coupled with the observation that exceptions are raised less often than not in computations that might raise them. In other words, troublesome situations like the one presented here are both statically and dynamically rare. Therefore we are willing to employ an optimistic strategy that favors computations during which no exceptions are raised. As a result, we accept that the degree of parallelism offered by our implementation is sometimes compromised, relative to the rest of the language, in the case of hard-to-handle expressions.

Note that handlers that operate on atomic elements, as in

```
[| 10 div n handle 1 | n in [| 1, 0, 9, 10, 12 |] |]
```

do not require any special consideration. Handlers in such cases can simply work at the level of individual function applications at the elements at the leaves. The strategy we have outlined here is incorporated into the transformation defined in Figure 4.8 below.

4.3.3 Inadvertently Introducing Nontermination

The following example illustrates how a common fusion transformation might cause a terminating computation not to terminate.

```
fun divergeNeg n = if n < 0 then (divergeNeg n) else n
```

```
fun valOf (SOME x) = x
  | valOf NONE     = raise Option
```

```
val c =
  let val ys = [| SOME ~1, NONE |]
  in
    [| divergeNeg x | x in [| valOf y | y in ys |] |]
  end
```

The function `divergeNeg` is the identity function on nonnegative inputs and diverges on negative inputs. The function `valOf` returns the value of an option if there is one to be had, and raises the exception `Option` otherwise. As it stands, `c` will terminate with a raise `Option` when the inner `valOf` encounters `NONE`.

The compiler will be tempted to apply a common fusion transformation to simplify `c` by function composition.

```
val c' = [| (divergeNeg o valOf) x | x in [| SOME ~1, NONE |] |]
```

This is an apparent improvement, as two array traversals have been fused into one. The two programs, however, are observationally different. The subtlety in this case is

that c' will loop forever on $(\text{divergeNeg } o \text{ valOf}) \text{ (SOME } \sim 1)$. Therefore we have demonstrated that function composition has the potential to transform a terminating computation into a nontermination one. We will need to be mindful of this problem in designing our fusion mechanism. Tracking exceptions in our type system (a possibility mentioned above in Subsection 4.3.1) may help us avoid this particular issue. The details of fusion in Manticore are beyond the scope of the present work, but they will be addressed in future work.

4.4 Flattening Tuples

In the following discussion we sometimes use “tuple” to mean “parallel tuple” where the meaning is clear. It must be understood, however, that the only subjects of the tuple flattening transformation are parallel tuples.

The nesting of parallel tuples has the potential to introduce unwanted barriers in the parallel execution of a program. Consider the parallel tuple

$$(| \text{ f } \mathbf{x}, (| \text{ g } \mathbf{x}, \text{ f } (\text{ g } \mathbf{x}) |) |)$$

for some values \mathbf{f} , \mathbf{g} and \mathbf{x} . Each parallel tuple corresponds to a multi-way fork/join in execution. Consequently, nested tuples—tuples that themselves contain tuples—induce multiple forks and joins. In this example, we must join twice: once for the inner tuple, and once for the outer tuple. For efficiency’s sake we would rather have a single fork spawning the evaluation of each tuple component (in this case three) followed by a single join. We realize this in Manticore by extending the flattening transformation to include parallel tuples.

A *flat tuple* is, roughly speaking, a tuple whose only parentheses are the outermost ones. $(|1, 2.0, "3"|)$ is a flat tuple; $(|(|1, 2.0|), "3"|)$ is not. More precisely, a tuple of type $\tau_1 * \tau_2 * \dots * \tau_n$ is flat if for all $1 \leq i \leq n$, τ_i is not a product type. Tuples that are not flat are *nested*.

In the formalism that follows we define a relation \mathcal{T} that consumes a parallel tuple

and produces a *nester*, which is a function to build an appropriately-shaped nested tuple from a flat one, applied to a flat tuple of the subexpressions of the original tuple in their original left-to-right order.

4.4.1 Flattening Tuples, Formally

To formalize the flattening transformation on parallel tuples, we first define a minimal term language on which to operate. Terms in this language are ranged over by the metavariable t . This language consists only of non-empty parallel tuples, which may be nested, and a generic expression form that is *not* a parallel tuple, represented by the term e .

$$\begin{aligned} t &::= (|\bar{t}|) \mid e \\ \bar{t} &::= t, \bar{t} \mid t \end{aligned}$$

Correspondingly, we define a language of shapes, ranged over by σ , whose grammar is isomorphic to the term grammar just defined.

$$\begin{aligned} \sigma &::= (\bar{\sigma}) \mid \bullet \\ \bar{\sigma} &::= \sigma, \bar{\sigma} \mid \sigma \end{aligned}$$

The $\mathbf{s} : t \rightarrow \sigma$ and $\bar{\mathbf{s}} : \bar{t} \rightarrow \bar{\sigma}$ relations calculate shapes from terms. Informally, a shape is produced from a term by replacing all its expressions e with bullets (\bullet) and all parallel parentheses with regular parentheses.

$$\begin{aligned} \mathbf{s}[|\bar{t}|] &= (\bar{\mathbf{s}}[\bar{t}]) \\ \mathbf{s}[e] &= \bullet \\ \bar{\mathbf{s}}[t, \bar{t}] &= \mathbf{s}[t], \bar{\mathbf{s}}[\bar{t}] \\ \bar{\mathbf{s}}[t] &= \mathbf{s}[t] \end{aligned}$$

We now define an operator $\mathbf{f}[\cdot]$ that has the effect of removing all the “inner parentheses” from a parallel tuple. Here is an example of its application:

$$\mathbf{f}[|(1, 2|), (|3, 4, 5|)|] = (1, 2, 3, 4, 5|)$$

We must first define a concatenation operator on \bar{t} terms, which we represent with the symbol \oplus .

$$\begin{aligned} t \oplus \bar{t} &= t, \bar{t} \\ (t, \bar{t}_1) \oplus \bar{t}_2 &= t, (\bar{t}_1 \oplus \bar{t}_2) \end{aligned}$$

The definition of $\mathbf{f} : t \rightarrow t$ necessitates the definition of two auxiliary operations $\mathbf{p} : t \rightarrow \bar{t}$ and $\bar{\mathbf{p}} : \bar{t} \rightarrow \bar{t}$. \mathbf{f} and \mathbf{p} are mnemonics for “flattener” and “parenthesis remover” respectively.

$$\begin{aligned} \mathbf{f}[(|\bar{t}|)] &= (|\bar{\mathbf{p}}[\bar{t}]|) \\ \mathbf{f}[e] &= e \\ \bar{\mathbf{p}}[t, \bar{t}] &= \mathbf{p}[t] \oplus \bar{\mathbf{p}}[\bar{t}] \\ \bar{\mathbf{p}}[t] &= \mathbf{p}[t] \\ \mathbf{p}[(|\bar{t}|)] &= \bar{\mathbf{p}}[\bar{t}] \\ \mathbf{p}[e] &= e \end{aligned}$$

We define a family of shape-indexed operators $\wedge^\sigma(\cdot)$ that impose nesting structure on input tuples according to the indexing shapes. These operators can be written as simple polymorphic lambda terms, as in the following example:

$$\wedge^{(\bullet, (\bullet, \bullet))} \equiv \lambda(x_1, x_2, x_3). (x_1, (x_2, x_3))$$

We will call these operators *tuple nesters* or simply *nesters*.

The formal definition of nesters is as follows. We first introduce a language of terms ν to represent nested tuples of variables. Each variable in a ν term is x_i where i is a positive integer; this simplifies the bookkeeping of variable names.

$$\begin{aligned} \nu &::= (\bar{\nu}) \mid x_i \ (i \in \mathbb{Z}^+) \\ \bar{\nu} &::= \nu, \bar{\nu} \mid \nu \end{aligned}$$

Then we define nesters as follows by means of auxiliary relations $\mathbf{p}_\nu : \nu \rightarrow \nu$ and $\mathbf{v} : \mathbb{Z}^+ \times \sigma \rightarrow \nu$. \mathbf{p}_ν is a no-inner-parentheses relation on ν terms exactly analogous to

the similar relation \mathbf{p} on terms above and whose formal definition is not given here. The measure $|\cdot| : \sigma \rightarrow \mathbb{Z}^+$ gives the number of bullets in a shape; its simple inductive definition is also omitted. We also employ a concatenation operator \oplus_ν for terms of type ν ; its definition is omitted as well. The relations \mathbf{v} and $\bar{\mathbf{v}}$ replace every bullet in a shape with a fresh variable. Semicolons are used to separate the parameters of \mathbf{v} to avoid confusion with commas, which have syntactic significance.

$$\begin{aligned} \wedge^\sigma &= \lambda \mathbf{p}_\nu[x] . x \text{ where } x = \mathbf{v}[1, \sigma] \\ \mathbf{v}[n; (\bar{\sigma})] &= (\bar{\mathbf{v}}[n; \bar{\sigma}]) \\ \mathbf{v}[n; \bullet] &= x_n \\ \bar{\mathbf{v}}[n; \sigma, \bar{\sigma}] &= \mathbf{v}[n; \sigma] \oplus_\nu \bar{\mathbf{v}}[n + |\sigma|; \bar{\sigma}] \\ \bar{\mathbf{v}}[n; \sigma] &= \mathbf{v}[n; \sigma] \end{aligned}$$

We define the tuple flattening transformation operator \mathcal{T} as follows:

$$\mathcal{T}[t] \rightarrow \wedge^{\mathbf{s}[t]} \mathbf{f}[t]$$

where t is a parallel tuple. The following example shows an application of \mathcal{T} .

$$\mathcal{T}[(|e_1, (|e_2, e_3|), (|e_4, e_5|)|)] = \wedge^{(\bullet, (\bullet, \bullet), (\bullet, \bullet))}(|e_1, e_2, e_3, e_4, e_5|)$$

In Subsection 4.6.1 we define a language \mathbb{M} , which is a simple model of the full Manticore language. \mathbb{M} includes a set of realistic programming constructs such as conditional expressions, let bindings, and so on. The extension of the formalization of \mathcal{T} to cover \mathbb{M} is straightforward, entailing definitions that are longer but not essentially different than those given here, and as such the extension is left to the interested reader.

4.5 Fusion

We defer a proper discussion of fusion to future work. In the present work, assume the fusion rules from Keller's dissertation apply to Manticore.

4.6 The Compiler as a Formal System

We present the semantics of the front end of a Manticore compiler as a series of program transformations from one language to another. Specifically, we define four languages: \mathbb{M} , which consists of core Manticore constructs, \mathbb{FM} , which is flattened Manticore, that is, Manticore without nested parallel constructs, \mathbb{DM} , distributed Manticore, and finally \mathbb{TM} , which is our target language. \mathbb{M} , \mathbb{FM} and \mathbb{DM} are inspired by NKL, FKL and DKL from Keller’s dissertation [16], which she defined in order to present transformations on which those in the present work are based. The final step in our transformation describes implementations of Manticore’s parallel constructs in \mathbb{TM} .

In previous chapters, our Manticore examples freely made use of a variety of standard functional programming language features, including SML-style records, algebraic datatypes, and higher-order functions. In this chapter’s formalization, we treat only a subset of those features that will be present in any realistic Manticore implementation. Our core language \mathbb{M} includes only function definition and application, tuples, let bindings and conditionals, along with an expression form for each of the languages features discussed in Chapter 2. The formalization in Keller’s thesis treats a similarly spare feature set, roughly corresponding to what we present here. Her thesis work, and specifically her flattening transformation, were extended in later work to cover datatypes [8] and higher-order functions [19].

The transformation from \mathbb{M} to \mathbb{FM} is denoted $\xrightarrow{\mathbb{F}}$; the transformation from \mathbb{FM} to \mathbb{DM} is denoted $\xrightarrow{\mathbb{D}}$; the transformation from \mathbb{DM} to \mathbb{TM} is denoted $\xrightarrow{\mathbb{T}}$. Each transformation is a set of rules that describes how expressions in a source language L can be rewritten as expressions in a target language L' . These rules fall into three broad categories:

- identity transformations on values and simple expressions (for example, $n \xrightarrow{\mathbb{F}} n$ for all integer constants n),
- preexisting rules from Keller’s transformations, and

- new rules addressing constructs in our languages that are not present in Keller’s languages.

In the present paper, we present rules of the third kind, leaving the other rules implicit. Specifically, we present rules pertaining to exceptions, parallel arrays, parallel comprehensions, parallel tuples, and parallel bindings.

4.6.1 \mathbb{M}

The \mathbb{M} language consists of a small core of computational elements, namely function definition, tuples, let bindings, conditionals, raising and handling exceptions, and each of the parallel features from Chapter 2. We also include integer and boolean constants, as well as a small base of necessary basis functions like `zips` of all arities, projection operators, and so on. The implicit type system is standard Hindley-Milner polymorphic typing extended with the typing rules presented with each expression form in Chapter 2. The grammar of \mathbb{M} is given in Figure 4.3.

4.6.2 \mathbb{FM}

\mathbb{FM} is similar to \mathbb{M} but differs in the following respects:

- There is no parallel comprehension form. Parallel comprehensions are rewritten away in the transformation $\xrightarrow{\mathbb{F}}$.
- There is an anonymous function (λ term) form.
- \mathbb{FM} includes forms for the introduction and elimination of *futures* (see Subsection 2.5.1).
- Some expression forms have a corresponding *lifted* form (see below) designated by a superscript arrow (\cdot^\uparrow) .

D	\rightarrow	letrec $F_1 \dots F_n$ in E	(program)
F	\rightarrow	fun $V (V_1, \dots, V_n) = E$	(function definitions)
E	\rightarrow	C	(constants)
		V	(variables)
		(E_1, \dots, E_n)	(tuples)
		(E_1, \dots, E_n)	(parallel tuples)
		$E_1 E_2$	(function application)
		let $P = E_1$ in E_2	(let bindings)
		plet $P = E_1$ in E_2	(parallel bindings)
		if E_1 then E_2 else E_3	(conditionals)
		$[E_1, \dots, E_n]$	(parallel arrays)
		$[E \mid V_1$ in E_1, \dots, V_n in E_n where $E_t]$	(parallel comprehensions)
		$[E_1$ to E_2 by $E_3]$	(ranges)
		$E_1 <?> E_2$	(parallel choice)
		raise $Undef$	(raising exceptions)
		E_1 handle E_2	(handling exceptions)
P	\rightarrow	V	(variable patterns)
		(P_1, \dots, P_n)	(tuple patterns)
C	\rightarrow	$0 \mid 1 \mid -1 \mid \dots$	(integers)
		true false	(booleans)
		$()$	(unit)
		$\pi_1 \mid \pi_2 \mid \dots$	(projection functions)
		zip ₁ zip ₂ \dots	(zip functions)
		$nil \mid cons \mid hd \mid tl$	(list functions)
V	\rightarrow	<i>variable</i>	(variables)

Figure 4.3: The grammar of \mathbb{M} .

The rules for rewriting \mathbb{M} terms into \mathbb{FM} terms are given in Figure 4.5. The grammar of \mathbb{FM} is given in Figure 4.4.

4.6.3 \mathbb{DM}

The language \mathbb{DM} introduces a layer of abstraction to distinguish local sequential computations from computations that might be executed across processors in parallel. Its **Loop** and **Gen** forms correspond roughly to **foldr** and **build** from the deforestation literature [13], and they are amenable to similar optimizations. Keller’s thesis gives a wealth of fusion rewriting rules on various juxtapositions of **Loop** and **Gen** forms; we assume these fusion rules apply to \mathbb{DM} as well.

We assume the translation of those \mathbb{FM} expressions that correspond to FKL expressions—namely, those involving only parallel arrays and function applications—are translated in \mathbb{DM} just as they are translated into DKL. Thus we present in $\xrightarrow{\mathbb{D}}$ only those rules pertaining to expression forms in \mathbb{FM} that have no counterpart in FKL. The grammar of \mathbb{DM} is given in Figure 4.6. The rules for transforming \mathbb{FM} expressions into \mathbb{DM} expressions are presented in Figure 4.8.

4.6.4 \mathbb{TM}

Our target language \mathbb{TM} is Standard ML extended with **amb** (see Section 2.8 for its definition) and the future-related forms introduced in \mathbb{FM} . In \mathbb{TM} we give concrete implementations of heretofore abstract parallel operations. Parallel arrays are represented with ropes (see Section 4.2) and parallel map, parallel filter, et cetera are defined as operations on those ropes. We have taken care to preserve the sequential semantics with respect to exceptions; see Section 4.3 for a lengthier discussion.

The grammar of \mathbb{TM} is given in Figure 4.10, and the transformation $\xrightarrow{\mathbb{T}}$ is given in Figure 4.11.

We present a prototype implementation of the compiler in \mathbb{TM} in Appendix A. A sim-

D	\rightarrow	letrec $F_1 \dots F_n$ in E	(program)
F	\rightarrow	fun V (V_1, \dots, V_n) = E	(function definitions)
E	\rightarrow	C	(constants)
		V	(variables)
		V^\uparrow	(“lifted” variables)
		(E_1, \dots, E_n)	(tuples)
		(E_1, \dots, E_n)	(parallel tuples)
		$\lambda V. E$	(anonymous functions)
		$E_1 E_2$	(function application)
		let $P = E_1$ in E_2	(let bindings)
		plet $P = E_1$ in E_2	(parallel bindings)
		if E_1 then E_2 else E_3	(conditionals)
		$[[E_1, \dots, E_n]]$	(parallel arrays)
		$[[E_1 \text{ to } E_2 \text{ by } E_3]]$	(ranges)
		$E_1 <?> E_2$	(parallel choice)
		raise $Undef$	(raising exceptions)
		E_1 handle E_2	(handling exceptions)
		E_1 handle $^\uparrow$ E_2	(handling exceptions)
		fut E	(future introduction)
		touch E	(future touching)
		cancel E_1 in E_2	(future cancellation)
		fut $^\uparrow$ E	(lifted future introduction)
		touch $^\uparrow$ E	(lifted future touching)
		cancel $^\uparrow$ E_1 in E_2	(lifted future cancellation)
P	\rightarrow	V	(variable patterns)
		(P_1, \dots, P_n)	(tuple patterns)
C	\rightarrow	<i>as in \mathbb{M} above</i>	
V	\rightarrow	<i>variable</i>	(variables)

Figure 4.4: The grammar of \mathbb{FM} .

$\xrightarrow{\mathbb{F}}_0$	E where E is a nested parallel tuple of type τ $\mathcal{T}(E)$	(FM-ptup-flat)
$\xrightarrow{\mathbb{F}}_1$	(E_1, \dots, E_n) let $f_1 = \mathbf{fut}(\lambda().E_1) \dots f_n = \mathbf{fut}(\lambda().E_n)$ in (touch $f_1, \dots, \mathbf{touch} f_n$)	(FM-ptup)
$\xrightarrow{\mathbb{F}}_1$	FIXME let $ V_1, \dots, V_n = E_1, \dots, E_n $ in E let $V_1^f = \mathbf{fut}(\lambda().E_1) \dots V_n^f = \mathbf{fut}(\lambda().E_n)$ in $[V_1 \mapsto \mathbf{touch} V_1^f, \dots, V_n \mapsto \mathbf{touch} V_n^f]_{tcs} E$	(FM-plet)
$\xrightarrow{\mathbb{F}}_2$	$[\mathbf{fut} e \mid x \text{ in } xs]$ $\mathbf{fut}^\uparrow(e, xs)$	(FM-fut)
$\xrightarrow{\mathbb{F}}_2$	$[\mathbf{touch} e \mid x \text{ in } xs]$ let $es = [e \mid x \text{ in } xs]$ in $\mathbf{touch}^\uparrow es$	(FM-touch)
$\xrightarrow{\mathbb{F}}_2$	$[\mathbf{cancel} e_1 \text{ in } e_2 \mid x \text{ in } xs]$ let $es_1 = [e_1 \mid x \text{ in } xs], es_2 = [e_2 \mid x \text{ in } xs]$ in $\mathbf{cancel}^\uparrow(es_1, es_2)$	(FM-cancel)
$\xrightarrow{\mathbb{F}}_2$	$[\mathbf{raise} \text{ Undef} \mid x \text{ in } xs]$ if $xs = [] $ then $[]$ else raise Undef	(FM-r)
$\xrightarrow{\mathbb{F}}_2$	$[e_1 \text{ handle } e_2 \mid x \text{ in } xs]$ $(\lambda x.e_1 \text{ handle}^\uparrow \lambda x.e_2) xs'$	(FM-h)

Figure 4.5: The $\xrightarrow{\mathbb{F}}$ transformation, which proceeds in three stages. The first stage flattens nested parallel tuples; the operator \mathcal{T} is defined in Section 4.4. The second stage desugars parallel tuples and parallel assignments, and the third stage flattens nested parallel comprehensions.

D	\rightarrow	letrec $F_1 \dots F_n$ in E	(program)
F	\rightarrow	fun $V (V_1, \dots, V_n) = E$	(function definitions)
E	\rightarrow	C	(constants)
		V	(variables)
		V^\uparrow	(“lifted” variables)
		(E_1, \dots, E_n)	(tuples)
		(E_1, \dots, E_n)	(parallel tuples)
		$[E_1, \dots, E_n]$	(parallel arrays)
		$E_1 <?> E_2$	(parallel choice)
		$\lambda V. E$	(anonymous functions)
		$A (E_1, \dots, E_n)$	(function application)
		let $P = E_1$ in E_2	(let bindings)
		plet $P = E_1$ in E_2	(parallel bindings)
		if E_1 then E_2 else E_3	(conditionals)
		raise $Undef$	(raising exceptions)
		E_1 handle E_2	(handling exceptions)
		E_1 handle $^\uparrow$ E_2	(handling exceptions)
P	\rightarrow	V	(variable patterns)
		(P_1, \dots, P_n)	(tuple patterns)
C	\rightarrow	<i>as in \mathbb{M} above</i>	
A	\rightarrow	V	
		G	
G	\rightarrow	Loop (G_1, G_2, G_3)	
		Loop $^\uparrow(G_1, G_2, G_3)$	
		Gen (G_1, G_2, G_3)	
		Gen $^\uparrow(G_1, G_2, G_3)$	
		Fut $^\uparrow(G)$	
		Touch $^\uparrow$	
		Cancel $^\uparrow$	
		$G_1 \times G_2$	
		$G_1 \Delta G_2$	
		$G_1 \circ G_2$	
		$G_1 \triangleleft G_2 \triangleright G_3$	
		$\langle G \rangle$	
		V	
V	\rightarrow	<i>variable</i>	(variables)

Figure 4.6: The grammar of \mathbb{DM} .

Map f	$\rightarrow \pi_1 \circ \mathbf{Loop}(f \circ \pi_1, \text{trivAcc}, \text{trueC}) \times id$	(simple maps)
Filter p	$\rightarrow \pi_1 \circ \mathbf{Loop}(id, \text{trivAcc}, p) \times id$	(simple filters)
Dist	$\rightarrow \pi_1 \circ \mathbf{Gen}(id, id, \text{trueC}_1)$	(value distribution)

Figure 4.7: Syntactic sugar for \mathbb{DM} .

$\xrightarrow{\mathbb{D}}$	raise $Undef$	(\mathbb{DM} -r)
$\xrightarrow{\mathbb{D}}$	raise $Undef$	
$\xrightarrow{\mathbb{D}}$	$(\lambda x.e_1 \mathbf{handle}^\uparrow \lambda x.e_2) xs$	(\mathbb{DM} -h-not-par)
$\xrightarrow{\mathbb{D}}$	$join \circ \langle \mathbf{Map}(\lambda x.(e_1 \mathbf{handle} e_2)) \rangle (split(xs), ())$	
$\xrightarrow{\mathbb{D}}$	$(\lambda x.e_1 \mathbf{handle}^\uparrow \lambda x.e_2) xss$	(\mathbb{DM} -h-par)
$\xrightarrow{\mathbb{D}}$	$join \circ \langle \mathbf{Map}(\lambda x.e_1) \rangle (split(xss), ())$	
	handle	
	$joinN \circ \langle \mathbf{Map}(\lambda x.(e_1 \mathbf{handle} e_2)) \rangle (splitN(xss), ())$	
$\xrightarrow{\mathbb{D}}$	$[e_1 \dots e_2 \mathbf{by} e_3]$	(\mathbb{DM} -range)
$\xrightarrow{\mathbb{D}}$	let $(a, b, c) = (e_1, e_2, e_3), n = ((b - a)/c) + 1$	
	in $join \circ \langle \pi_1 \circ \mathbf{Gen}(id, \lambda m.m + c, \text{trueC}_1) \rangle (splitLen\ n, splitSc\ a)$	
$\xrightarrow{\mathbb{D}}$	$\mathbf{fut}^\uparrow(e, xs)$	(\mathbb{DM} -fut)
$\xrightarrow{\mathbb{D}}$	Fut $^\uparrow(\lambda x.e) xs$	
$\xrightarrow{\mathbb{D}}$	$\mathbf{touch}^\uparrow xs$	(\mathbb{DM} -touch)
$\xrightarrow{\mathbb{D}}$	Touch $^\uparrow xs$	
$\xrightarrow{\mathbb{D}}$	$\mathbf{cancel}^\uparrow(e_1, e_2)$	(\mathbb{DM} -cancel)
$\xrightarrow{\mathbb{D}}$	Cancel $^\uparrow(e_1, e_2)$	

Figure 4.8: The $\xrightarrow{\mathbb{D}}$ transformation. The functions $joinN$ and $splitN$ create the “rumpled” structures discussed in Subsection 4.3.2.

```

datatype  $\alpha$  rope
  = Cat    of  $\alpha$  rope *  $\alpha$  rope
  | Leaf   of  $\alpha$  vector
  | Raise  of exn

type seg_desc = int list

datatype  $\alpha$  flat_rope = FlatRope of  $\alpha$  rope * seg_desc

val mkCat :  $\alpha$  rope *  $\alpha$  rope ->  $\alpha$  rope (* smart constructor *)
val tabulateD : (int ->  $\alpha$ ) -> int * int ->  $\alpha$  rope
val mapD : ( $\alpha$  ->  $\beta$ ) ->  $\alpha$  rope ->  $\beta$  rope
val filterD : ( $\alpha$  -> bool) ->  $\alpha$  rope ->  $\alpha$  rope
val reduceD : ( $\alpha$  *  $\alpha$  ->  $\alpha$ ) ->  $\alpha$  ->  $\alpha$  rope ->  $\alpha$ 

```

Figure 4.9: Ropes and rope operations in TML. See Appendix A for more details.

ple rope structure is encoded as a datatype in Figure A.1, along with some auxiliary functions defining common operations on them (See Figure 4.9 for relevant types).

The functions that compute on these ropes in parallel do so in terms of futures. Rope traversals are parallelized by computing on right subropes as futures and computing on left subropes immediately. Recursive appeals to this strategy generate a high degree of parallelism. The implementation of a parallel tabulating function—that is, a function that builds a parallel array by applying a function of type $\text{int} \rightarrow \alpha$ to a given range of integers—is given in Figure A.2. Parallel implementations of `map` and `filter` are presented in Figure A.3.

A `trap` datatype, for trapping exceptions, is defined in Figure A.4; `Traps` are used in the implementation of `reduceD` (Figure A.5), an abstract parallel sum operator. `ReduceD` consumes an associative operator and an identity element for that operator, and performs a parallel reduction with that operator on a given rope. A parallel sum function, for example, is easily expressed as `reduceD op+ 0`. Finally, three implementations of “lifted future” combinators are given in Figure A.6; these are targets of rules in the $\xrightarrow{\mathbb{T}}$ transformation given in Figure 4.11.

D	\rightarrow	<code>letrec $F_1 \dots F_n$ in E</code>	(program)
F	\rightarrow	<code>fun $V (V_1, \dots, V_n) = E$</code>	(function definitions)
E	\rightarrow	<code>C</code>	(constants)
		<code>V</code>	(variables)
		<code>(E_1, \dots, E_n)</code>	(tuples)
		<code>fn $P \Rightarrow E$</code>	(anonymous functions)
		<code>amb(E_1, E_2)</code>	(nondeterministic choice)
		<code>$E_1 E_2$</code>	(function application)
		<code>let val $P = E_1$ in E_2 end</code>	(let bindings)
		<code>if E_1 then E_2 else E_3</code>	(conditionals)
		<code>raise Undef</code>	(raising exceptions)
		<code>E_1 handle $E_2 \Rightarrow E_3$</code>	(handling exceptions)
		<code>fut E</code>	(future introduction)
		<code>touch E</code>	(future touching)
		<code>cancel E_1 in E_2</code>	(future cancellation)
P	\rightarrow	<code>V</code>	(variable patterns)
		<code>(P_1, \dots, P_n)</code>	(tuple patterns)
C	\rightarrow	<i>as in \mathbb{M} above</i>	
V	\rightarrow	<i>variable</i>	(variables)

Figure 4.10: The grammar of TM.

$\xrightarrow{\mathbb{T}}$	$join \circ \langle \mathbf{Map} \ f \rangle \ (split \ xs, ())$	(TM-mapD)
$\xrightarrow{\mathbb{T}}$	$(propX \circ mapD \ f) \ (mkFlatRope \ xs)$	
$\xrightarrow{\mathbb{T}}$	$join \circ \langle \mathbf{Filter} \ p \rangle \ (split \ xs, ())$	(TM-filtD)
$\xrightarrow{\mathbb{T}}$	$(propX \circ filterD \ p) \ (mkFlatRope \ xs)$	
$\xrightarrow{\mathbb{T}}$	$(joinW \oplus) \circ \langle \pi_2 \circ \mathbf{Loop} \ (id, \oplus, falseC_2) \rangle \ (split \ xs, splitSc \ 0_{\oplus})$	(TM-red)
$\xrightarrow{\mathbb{T}}$	$reduceD \ circlePlus \ zero \ (mkFlatRope \ xs)$	
$\xrightarrow{\mathbb{T}}$	$join \circ \langle \mathbf{Dist} \rangle \ (splitSc \ k, splitLen \ n)$	(TM-dist)
$\xrightarrow{\mathbb{T}}$	$(propX \circ tabulatedD \ (fn \ anything \ => \ k)) \ (1, \ n)$	
$\xrightarrow{\mathbb{T}}$	$join \circ \langle \pi_1 \circ \mathbf{Gen} \ (id, \lambda m.m + c, trueC_1) \rangle \ (splitLen \ n, splitSc \ a)$	(TM-range)
$\xrightarrow{\mathbb{T}}$	$(propX \circ tabulatedD \ (fn \ m \ => \ a+m*c)) \ (0, \ n)$	
$\xrightarrow{\mathbb{T}}$	$\mathbf{Fut}^{\uparrow} \ f \ xs$	(TM-fut)
$\xrightarrow{\mathbb{T}}$	$futureMapS \ f \ xs$	
$\xrightarrow{\mathbb{T}}$	$\mathbf{Touch}^{\uparrow} \ xs$	(TM-touch)
$\xrightarrow{\mathbb{T}}$	$touchS \ xs$	
$\xrightarrow{\mathbb{T}}$	$\mathbf{Cancel}^{\uparrow} \ (xs, ys)$	(TM-cancel)
$\xrightarrow{\mathbb{T}}$	$cancelS \ (xs, ys)$	
$\xrightarrow{\mathbb{T}}$	$join \circ \langle \pi_1 \circ \mathbf{Loop} \ (f, g, b) \rangle \ (split \ xs, a)$	(TM-loop-seq)
$\xrightarrow{\mathbb{T}}$	$loopS \ (f, \ g, \ b) \ (xs, \ a)$	
$\xrightarrow{\mathbb{T}}$	$join \circ \langle \pi_1 \circ \mathbf{Gen} \ (f, g, b) \rangle \ (ns, ks)$	(TM-gen-seq)
$\xrightarrow{\mathbb{T}}$	$genS \ (f, \ g, \ b) \ (ns, \ ks)$	

Figure 4.11: $\xrightarrow{\mathbb{T}}$

CHAPTER 5

RELATED WORK

Parallel programming languages are a hot topic. Many other parallel programming languages are under active development both in industry and in academia. This section surveys a selection of related projects contemporary with our own.

Our most direct sources of inspiration are the data-parallel functional languages NESL, Nepal, and Data Parallel Haskell, with a focus on the relevant contributions in Keller’s dissertation. The connections between these languages and Manticore are numerous and have been detailed in the preceding chapters.

5.1 StreamIt

The StreamIt programming language [31] is built around the notion of composable units called **filters**. A **filter** is an abstraction containing a function **work** of type $\alpha \rightarrow \beta$ that, when attached to an input stream of type α , yields a stream of type β whose values are the resulting of applying **work** to the values of the input stream.

Filters can be aggregated into three different kinds of structures: **pipelines**, **splitjoins** and **feedbackloops**. A **pipeline** is a linear composition of **filters** such that the component **filters** are applied in succession. A **splitjoin** splits the values of its input stream into multiple values according to a specified means, processes each of them, and feeds the results to its output stream according to another specified means. A **feedbackloop** is a filter that both yields its outputs to a stream and feeds them back into its input.

We show here an encoding of these StreamIt filters in Manticore using explicit threading. We thereby demonstrate to a coarse approximation that StreamIt programs can

encoded in Manticore.

```
type ( $\alpha$ ,  $\beta$ ) filter = ( $\alpha$  chan,  $\beta$  chan,  $\alpha \rightarrow \beta$ , unit  $\rightarrow$  unit)
```

```
fun mkFilter (a:  $\alpha$  chan, work:  $\alpha \rightarrow \beta$ ) : ( $\alpha$ ,  $\beta$ ) filter =
  let val b = channel ()
    fun pop () = recv a
    fun push(x) = send (b, x)
    fun go () = (push (work (pop ())), go ()); go ()
  in
    (a, b, work, go)
  end
```

```
fun mkPipeline (ab: ( $\alpha$ ,  $\beta$ ) filter, bg: ( $\beta$ ,  $\gamma$ ) filter) : ( $\alpha$ ,  $\gamma$ ) filter =
  let val (a, _, work1, _) = ab
    val (_, g, work2, _) = bg
    fun pop () = recv a
    fun push(x) = send (g, x)
    fun work' = work2 o work1
    fun go () = (push (work' (pop ())), go ()); go ()
  in
    (a, g, work', go)
  end
```

```
datatype sj = Duplicate | Roundrobin of int list
```

```
fun mkSplitjoin (a:  $\alpha$  chan, ab1: ( $\alpha$ ,  $\beta$ ) filter, ab2: ( $\alpha$ ,  $\beta$ ) filter,
                Duplicate, Roundrobin []) : ( $\alpha$ ,  $\beta$ ) filter =
```

```
(* note: only one sj combination presented *)
```

```
let val b = channel ()
    val (_, _, work1, _) = ab1
    val (_, _, work2, _) = ab2
    fun pop () = recv a
    fun push x = send (b, x)
    fun go () = let val x = pop ()
                pval w2 = work2 x
                in
                    push (work1 x); push w2; go ()
                end
    val switch = cell () (* assume CML implementation *)
    fun work x = let val b = get switch
                in put (switch, not b);
                  (if b then work1 else work2) x
                end
    in
        put (switch, true);
        (a, b, work, go)
    end
```

```
fun mkFeedbackloop (body: ( $\alpha$ ,  $\alpha$ ) filter,
                   loop: ( $\alpha$ ,  $\alpha$ ) filter,
                   Duplicate, Roundrobin []) =
    (* note: only one sj combination presented *)
    let val (bIn, bOut, bWork, _) = body
        val (lIn, lOut, lWork, _) = loop
        fun pop () = recv lOut
        fun push(x) = (send (bOut, x); send (lIn, x))
```

```

    fun work = lWork o bWork
    fun go () = (push (work (pop ()))); go ()
in
    (bIn, bOut, work, go)
end

val (intChan, floatChan) = (channel (), channel ())
val intToFloatFilter  = mkFilter (intChan, intToFloat)
val negateFloatFilter = mkFilter (floatChan, neg)
val p = mkPipeline (intToFloatFilter, negateFloatFilter)

// StreamIt program
int->int filter RunningTotal {
    int t;
    init {t = 0;}
    work {t += pop(); push(t);}
}

(* Manticore program *)
val runningTotal =
    let val t = cell ()
        fun rsum x = (put (t, t+x); t + x)
    in
        mkFilter (channel (), rsum)
    end
end

```

5.2 Cilk

The Cilk programming language [5, 14] is an extension of C with additional constructs for expressing parallelism. Cilk has demonstrated success in various domains. For example, three world-class chess programs have been implemented in it: *★Tech* [18],

★Socrates [15], and Cilkchess [1]. Cilk is an imperative language, and, as such, its semantics is different from Manticore's in some obvious ways.

Superficially, Cilk appears to be annotated C. Cilk extends C with the following keywords: `cilk`, `spawn`, `sync`, `inlet`, `abort`, and `SYNCHED`. Each one is considered in relation to Manticore in turn.

The keyword `cilk` is a modifier designating a Cilk procedure, that is, one that may make use of parallelism. Any C procedure can be trivially preceded by this modifier to be transformed into a Cilk procedure, as in

```
cilk int f(int x) {return x;}
```

Any given Cilk program consists of a collection of C procedures and Cilk procedures. Cilk procedures can call C procedures and Cilk procedures, whereas C procedures can only call C procedures. In Manticore, there is no distinction in the language between those functions that might execute in parallel and those that will not, so we have no mechanism analagous to `cilk`. Another obvious difference between Cilk and Manticore is the presence of side effects in the former and their absence (except those pertaining to exceptions and communication) in the latter.

Cilk procedures call other Cilk procedures with the use of the `spawn`. A spawned procedure starts running in parallel, and its parent procedure continues execution. In this way, spawned Cilk procedures are similar to Manticore expressions bound with `pval`. For example, the following Cilk procedure computes $\binom{n}{k}$ by means of parallel recursive calls.

```
cilk int choose(int n, int k) {
    if (k == 1) {
        return n;
    } else if ((k == n) || (k == 0)) {
        return 1;
    } else {
        int a, b;
```

```

    a = spawn choose(n-1, k-1);
    b = spawn choose(n-1, k);
    return (a + b);
}

```

Spawned procedure statements are not expressions. Specifically, the `return` statement in `choose` cannot be expressed in the following way:

```
return ((spawn choose(n-1, k-1)) + (spawn choose(n-1, k)));
```

This limitation is overcome by binding each sub-computation to its own variable, then computing with those variables, as in `choose` above.

In Manticore, a similar function can be written using parallel bindings:

```

(* mchoose: int * int -> int *)
fun mchoose (n, k) =
  if (k=1) then n
  else if ((k=n) orelse (k=0)) then 1
  else let pval a = mchoose (n-1, k-1)
        pval b = mchoose (n-1, k)
        in (a + b) end

```

The behavior of this Manticore function is essentially the same as that of the Cilk function. Both are pure functions, and their control flow is identical.

One can employ parallel tuples in Manticore to express the same function. The following expression is the behavioral equivalent of the parallel bindings in `mchoose` above:

```
op+ (| mchoose (n-1, k-1), mchoose (n-1, k) |)
```

This form happens to be more concise than the parallel binding form, but there is no urgent reason to prefer one to the other. The behavioral differences between parallel

tuples and parallel bindings are only made manifest when the speculative aspect of parallel bindings is in play (see Section 2.5).

The Cilk statement **sync** suspends execution until all pending parallel procedure calls are complete. We can insert a **sync** statement into the **return** branch of the procedure **choose** as follows:

```
a = spawn choose(n-1, k-1);
b = spawn choose(n-1, k);
sync;
return (a + b);
```

The presence of the **sync** statement guarantees that the two spawned procedures are finished before their values are added together. We were able to omit a **sync** statement in the definition of **choose** above because Cilk inserts a **sync** statement before **return** statements if no **sync** is present. A similar synchronization occurs in **mchoose** whether we use the parallel binding or the parallel tuple form, but that synchronization is implicit in Manticore. In fact, it can only be implicit, as Manticore provides no direct way to express what **sync** expresses.

Cilk provides a pseudo-variable **SYNCHED** to allow programs to test dynamically whether or not all pending processes are complete in a given context. In the Cilk manual [14], **SYNCHED** is presented as a means of conserving space, as in the following example:

```
state1 = alloca(state_size);
spawn foo(state1);
if (SYNCHED) state2 = state1;
else state2 = alloca(state_size);
spawn bar(state2);
```

In this example memory is allocated for **state2** only if the memory allocated for **state1** is still in use. Since Manticore has no shared state, this use of **SYNCHED** is not applicable to the present work. One can imagine using **SYNCHED** to throttle parallelism

to keep live data under control. At present, Manticore provides no similar mechanism for dynamic performance tuning.

A Cilk `inlet` is a nested procedure with special properties. Its first “argument” must be a `spawn` statement, and all of its (zero or more) subsequent arguments must be expressions. The execution of an `inlet` is guaranteed to be atomic with respect to other `inlets` that are children of the same parent procedure. The following implementation of `fib` presents an `inlet` in context.¹

```
cilk int fib(int n) {
    int x = 0;
    inlet void add(int r) {
        x += r;
        return;
    }
    if (n<2) return n;
    else {
        add(spawn fib(n-1));
        add(spawn fib(n-2));
        return x;
    }
}
```

Note the atomicity guarantee of inlets with respect to its siblings ensures the correctness of `x` when it is returned; in other words, atomicity prevents races to increment `x`. This particular inlet is essentially unrelated to Manticore, which has no assignment. Functional inlets, that is, inlets free of side effects, are easily simulated in Manticore, since pure Manticore functions can be defined in any scope.

The most interesting point of consideration about inlets has to do with their interaction with `abort` statements. Any inlet can include an `abort`. An `abort` cancels all pending computations of a common parent process. It provides a straightforward

1. This is a paraphrase of an example in the Cilk manual [14].

means of expressing speculative computation, whose evaluation semantics are different from any programs that can be written in Manticore. We explore these differences below.

We claim that while a large set of parallel programs can be expressed in both Cilk and Manticore, there are some programs which can be expressed in one but not the other. That is, the languages are not merely imperative and functional versions of a common essential language. Two examples demonstrate this claim.

First, we consider a Manticore program that cannot be written down (at least not in any obvious way) in Cilk.

```
val a = let pval x = foo ()
          pval y = bar ()
        in
          case e (* e does not use x or y *)
            of true => x
             | false => y
        end
```

This Manticore program computes x , y , and e in parallel. When e has been fully evaluated, the program follows the appropriate arm of the case expression, where one of x or y is cancelled mid-flight.

We can attempt to write the same program in Cilk as follows:

```
int x, y, c;
x = spawn foo();
y = spawn bar();
c = e;
sync;
if (c) then return x;
else return y;
```

This program also evaluates x , y , and e in parallel, but the `sync` statement suspends

control until all three are finished evaluating, even though either the value of `x` or the value of `y` is ultimately not needed. We cannot leave out the `sync` statement and hope for the best: if we leave out `sync`, the behavior of the program is unspecified. Note that while we (implicitly) cancel one arm of the conditional in the Manticore program, we cannot cancel only one of `x` and `y` in the Cilk program. In summary, the Manticore program has the potential to exhibit greater and finer-grained parallelism than its Cilk counterpart.

On the other hand, Cilk’s `abort` statement makes it possible to control the cancellation of parallel computations according to any logic the programmer requires. Consider the following Cilk inlet. We assume that some integer `s` is under consideration as a possible solution to some unspecified problem. The program is charged with gathering solutions until some `quota` is met. When the `quota` has been attained, the program cancels all pending computations, having gathered the solutions found into a `stackOfSolutions`.

```
inlet int f(int s) {
  if (isASolution(s))
    push(stackOfSolutions, s);
  if (stackSize(stackOfSolutions) == quota)
    abort;
}
```

We cannot easily write a similar program in Manticore. We can compute precisely one of a number of possible solutions in parallel by means of the parallel choice operator:

```
val sol = e1 <?> e2 <?> ... <?> en
```

When one of the expressions in this chain of parallel choices is done evaluating, all others will be cancelled. There is, however, no obvious way to write the program that computes two (or more) solutions such that all pending computations are cancelled. One can construct an entirely unreasonable program to return two solutions as follows:

```
val sols = (e1, e2) <?> (e1, e3) <?> ... <?> (e1, en) <?>
```

```
(e2, e3) <?> ...
... <?> (enminus1, en)
```

This program evaluates quadratically more expressions than the original. In general, this technique results in exponentially bigger code size, and terrible performance; it is clearly not a realistic practice.

These examples demonstrate that neither Manticore nor Cilk provides finer-grained control than the other in general: either language provides finer control depending on the circumstances.

5.3 Other Contemporary Parallel Languages

5.3.1 *pH*

pH [25], a descendant of *Id* [24], is an implicitly parallel functional language with shared mutable state and an explicit sequencing operator ($>>>$). *pH* closely resembles Haskell. In *pH*, the programmer has almost no say about how a computation is divided into parallel subcomputations. *pH*'s arrays are like Manticore's parallel arrays, its comprehensions are like Data Parallel Haskell's parallel comprehensions, and its tuples are like Manticore's parallel tuples. The designers of *pH* believe explicitly managing control in a given program can be left to expert programmers in those special circumstances where fine distinctions in performance are important, and they have designed a language that essentially precludes the possibility of improving performance by manipulating control explicitly. Our design allows the programmer more control through its mixture of sequential, concurrent and parallel mechanisms.

pH provides mutable state in two forms, *I-structures* and *M-structures*. I-structures are updateable memory cells that are initially empty and can be written to exactly once in their lifetime; they can be read from arbitrarily many times. If a computation tries to read the contents of a I-structure and that structure is empty, it suspends computation until a value has been written to that structure. M-structures, by con-

trast, can not only be read from arbitrarily many times, they can also be written to arbitrarily many times. Reading from an M-structure will suspend computation until a value is present. There are two operations for reading M-structures, *fetch* and *examine*. Fetching from an M-structure empties its location; the M-structure will need to be written to again to satisfy any subsequent attempts to read it. Examining an M-structure yields its value, but leaves its value intact. I-structures and M-structures can both be encoded in CML [26], so they can correspondingly be encoded in Manticore. We are currently considering more basic language support for *pH*-style shared mutable data in Manticore, but we have yet to reach a conclusion about this aspect of our design.

pH does include an explicit sequencing operator, written `>>>`. The presence of this operator makes it possible to write sequential programs in *pH*, i.e., one can employ it to write programs that blatantly violate the spirit *pH* altogether. Nevertheless, there are situations where the use of this operator simplifies certain tasks considerably. Programs with output effects, for example, might require their effects to occur in a predictable order. Such programs can be written clearly and simply with judicious use of explicit sequencing. The issue of having a separate operator for explicit sequencing does not arise in Manticore, given the abundance of sequential constructs in our language.

In a given *pH* program, every tuple is parallel tuple; in Manticore, we can write similar programs by using parallel tuples only. The situation is similar with arrays, comprehensions, and bindings. Where *pH*'s sequencing operator `>>>` is used to suspend evaluation until values `x1`, `x2`, ..., `xn` have finished evaluation, a Manticore program can induce similar behavior by binding all pending values in an assignment to a (sequential) tuple:

```
val >>> = (x1, x2, ..., xn)
```

In summary, we believe that *pH* programs can be written in Manticore by providing the appropriate parallel annotations and using “synchronizing bindings” like the one above where necessary.

5.3.2 Eden

Eden [20] is a parallel functional language closely related to Haskell. Eden inherits many of Haskell’s distinguishing characteristics, including type classes and lazy evaluation. Unlike Manticore, Eden is specifically designed for use on distributed parallel machines; thus the focus of their implementation is fundamentally different from ours.

Eden programs are based on skeletons. Skeletons are essentially higher-order functions that capture common program structures: they are similar to `map` and `filter` but there are more of them and (traditionally) there is a fixed set of them. The original work on skeletons [11] focuses on the suitability of particular skeletons for certain parallel systems; users make informed choices about which skeletons to employ based on knowledge of their performance on a given target system. Eden is distinguished from its predecessors in that it allows programmers to create and develop their own skeletons in the surface language.

The higher-level concerns of the design of Eden are in many ways close to our own. It appears that Eden programs can be translated into Manticore programs, appealing to CML-style constructs where data parallel constructs are insufficient. Nevertheless, Manticore’s single-machine focus will continue to differentiate it from Eden as both projects move forward.

5.3.3 The DARPA HPCS Languages

The DARPA HPCS is the Defense Advanced Research Projects Agency’s High Productivity Computing Systems project. Their initiatives include funding development of three major parallel language projects: X10, Fortress, and Chapel. All three are imperative languages designed for use on machines with very large numbers of processing elements, thus, they differ from Manticore both in paradigm and target architectures.

X10 [28], Chapel [7] and Fortress [30] are imperative languages with high-level parallel

constructs. X10 is, by design, close to Java, superficially and otherwise. Some of X10's features are similar to corresponding features in Manticore: X10 supports "parallel collective operations" on arrays, provides (explicit) futures and has an exception-catching feature that passes exceptions up tree structures. Chapel has `forall` loops and `forall` expressions, and Fortress has aggregates and comprehensions, both of which are similar to our parallel comprehensions. Arrays in Chapel and Fortress are operated on in parallel by default, similar to our parallel arrays. The main differences between these languages and ours seem to be their imperative nature and their focus on supporting object-oriented programming. Because all three languages have shared mutable state, their language designs are largely focused on notions of atomicity, a topic our design does not currently address.

APPENDIX A

IMPLEMENTATION SKETCHES

This appendix contains outlines of our implementations of various Manticore constructs and combinators. These sketches are largely concerned with ropes (see Section 4.2) and traversals on them.

```

datatype  $\alpha$  rope
  = Cat    of  $\alpha$  rope *  $\alpha$  rope
  | Leaf   of  $\alpha$  vector
  | Raise  of exn

type seg_desc = int list

datatype  $\alpha$  flat_rope
  = FlatRope of  $\alpha$  rope * seg_desc

(* mkCat :  $\alpha$  rope *  $\alpha$  rope ->  $\alpha$  rope *)
(* smart constructor for ropes *)
fun mkCat (Raise e, _) = Raise e
  | mkCat (_, Raise e) = Raise e
  | mkCat (r1, r2) = compactAndBalance (r1, r2)
                        (* implementation unspecified *)

(* propX :  $\alpha$  rope ->  $\alpha$  rope *)
(* propagate exceptions out of ropes *)
fun propX (Raise e) = raise e
  | propX r = r

(* cancelOrCat :  $\alpha$  rope *  $\alpha$  rope future ->  $\alpha$  rope *)
fun cancelOrCat (Raise e, f) = (cancel f in Raise e)
  | cancelOrCat (r, f) = mkCat (r, touch f)

```

Figure A.1: Ropes in TML.

```

(* tabulateS : (int ->  $\alpha$ ) -> int * int ->  $\alpha$  vector *)
(* sequential tabulation *)
fun tabulateS f (lo, hi) =
  let fun build (n, acc) =
        if n > hi then rev acc
        else build (n+1, f(n)::acc)
      in
        Vector.fromList (build (lo, []))
      end

(* tabulateD : (int ->  $\alpha$ ) -> int * int ->  $\alpha$  rope *)
(* distributed tabulation *)
(* assumes the existence of the value "leafsize" *)
fun tabulateD f (lo, hi) =
  let val tabS = tabulateS f
      val nLeaves = ceil (real(hi - lo + 1) / real(leafsize))
      fun tabD (loLeaf, hiLeaf) =
          if (loLeaf = hiLeaf) then
            let val l = lo + (loLeaf * leafsize)
            in
              Leaf (tabS (l, min(l+leafsize-1, hi)))
            handle e => Raise e
            end
          else
            let val piv = (loLeaf + hiLeaf) div 2
                val f2 = fut (fn () => tabD (piv+1, hiLeaf))
                val r1 = tabD (loLeaf, piv)
            in
              cancelOrCat (r1, f2)
            end
          end
    in
      tabD (0, nLeaves-1)
    end
end

```

Figure A.2: Rope construction functions in TM.

```

(* mapD : ( $\alpha \rightarrow \beta$ )  $\rightarrow \alpha$  rope  $\rightarrow \beta$  rope *)
fun mapD f xs =
  let fun m (Cat (r1, r2)) =
        let val f2 = fut (fn () => m r2)
        in
          cancelOrCat (m r1, f2)
        end
      | m (Leaf v) = (Leaf (Vector.map f v))
                    handle e => Raise e
      | m (Raise e) = Raise e
  in
    m xs
  end

(* vfilter : ( $\alpha \rightarrow \text{bool}$ )  $\rightarrow \alpha$  vector  $\rightarrow \alpha$  vector *)
fun vfilter pred v =
  let val vlist = Vector.foldr op:: [] v
  in
    Vector.fromList (List.filter pred vlist)
  end

(* filterD : ( $\alpha \rightarrow \text{bool}$ )  $\rightarrow \alpha$  rope  $\rightarrow \alpha$  rope *)
fun filterD p xs =
  let fun f (Cat (r1, r2)) =
        let val f2 = fut (fn () => f r2)
        in
          cancelOrCat (f r1, f2)
        end
      | f (Leaf v) = (Leaf (vfilter pred v))
                    handle e => Raise e
      | f (Raise e) = Raise e
  in
    f xs
  end

```

Figure A.3: Distributed map and filter in TML.


```

datatype  $\alpha$  trap
  = Value of  $\alpha$ 
  | Exn of exn

(* release :  $\alpha$  trap ->  $\alpha$  *)
fun release (Value v) = v
  | release (Exn e)   = raise e

```

Figure A.4: The trap abstraction in TML.

```

(* reduceD : ( $\alpha$  *  $\alpha$  ->  $\alpha$ ) ->  $\alpha$  ->  $\alpha$  rope ->  $\alpha$  *)
(* assumptions: oper is associative, zero is its identity *)
fun reduceD oper zero xs =
  let fun r (Cat (r1, r2)) =
        let val f2 = fut (fn () => r r2)
            val r1' = r r1
        in
          case r1'
          of Exn e => (cancel f2; Exn e)
            | _ => let val r2' = touch f2
                  in
                    case r2'
                    of Exn e => Exn e
                     | _ => Value (oper (release r1',
                                          release r2'))
                  end
          end
        end
      | r (Leaf v) = (Vector.foldr oper zero v)
                    handle e => Exn e
      | r (Raise e) = Exn e
  in
    release (r xs)
  end

```

Figure A.5: Distributed reduction in TML.

```

(* futureMapS : ( $\alpha \rightarrow \beta$ )  $\rightarrow \alpha$  list  $\rightarrow \beta$  future list *)
fun futureMapS f xs =
  let fun m [] = []
      | m (x::xs) =
          let val ffx = fut (fn () => f x)
          in
            ffx::(m xs)
          end
      in
        m xs
      end

(* touchS :  $\alpha$  future list  $\rightarrow \alpha$  list *)
fun touchS fs = map (fn f => touch f) fs

(* cancelS :  $\alpha$  future list *  $\beta$  list  $\rightarrow \beta$  list *)
fun cancelS (as, bs) =
  ListPair.map (fn (a, b) => cancel a in b) (as, bs)

```

Figure A.6: Futuristic traversals in TML.

REFERENCES

- [1] The cilkchess website (<http://supertech.csail.mit.edu/chess/>).
- [2] Josh Barnes and Piet Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, December 1986.
- [3] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.
- [4] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.
- [5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.
- [6] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software—Practice & Experience*, 25(12):1315–1330, 1995.
- [7] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '04)*, pages 52–60, Los Alamitos, CA, April 2004. IEEE Computer Society Press.
- [8] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 94–105, New York, NY, September 2000. ACM.
- [9] Manuel M. T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Wolf Pfannenstiel. Nepal – Nested Data Parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference on Parallel Computing*, volume 2150 of *Lecture Notes in Computer Science*, pages 524–534, New York, NY, August 2001. Springer-Verlag.
- [10] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. New York, NY, USA, 2007. ACM Press.

- [11] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.
- [12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.
- [13] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM Press.
- [14] Supercomputing Technologies Group. Cilk 5.4.3 (rev 3379) Reference Manual. 2007.
- [15] C. Joerg and B. Kuszmaul. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge Workshop*, Rutgers University, 1994.
- [16] Gabriele Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. PhD thesis, Technische Universität Berlin, 1999.
- [17] Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In José Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.
- [18] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1994.
- [19] Roman Leshchinskiy, Manuel M. T. Chakravarty, and Gabriele Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2006)*, number 3992 in LNCS, pages 920–928, New York, NY, May 2006. Springer-Verlag.
- [20] Rita Loogen, Yolanda Ortega-mallén, and na-mari Ricardo Pe. Parallel functional programming in eden. *Journal of Functional Programming*, 15(3):431–475, 2005.
- [21] John McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

- [22] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.
- [23] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. 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*, pages 185–197, New York, NY, June 1990. ACM.
- [24] Rishiyur S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.
- [25] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
- [26] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.
- [27] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
- [28] Vijay Saraswat. Report on the experimental language x10. Technical report, IBM, February 2006.
- [29] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
- [30] Guy L. Steele Jr. Parallel programming and code selection in Fortress. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, page 1, New York, NY, March 2006. ACM.
- [31] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.