Chapter 3

LAZINESS AND PARALLEL DATA STRUCTURES

3.1 Introduction

There are two routes into parallelism, one is through intra-processor parallelism (e.g. replicating functional units within a processor) and the other is through inter-processor parallelism (replicating processors). The advantage of intra-processor parallelism is that it does not require a shift in programming style: the computer still looks like a normal von Neumann machine. The processor uses internal concurrency to boost performance, e.g. multiple instructions can be started at each clock cycle, or at the bit level by operating on each bit of a word concurrently. To use this form of the parallelism only the compilers need be modified so that instructions in the sequential program are scheduled to avoid the processor stalling.

The alternative to intra-processor parallelism is inter-processor parallelism. Here, multiple processors are combined using some interconnection network to form a parallel computer. The combined processors either have local per-processor memories or one large shared memory. Either way, each processor can be viewed as a single computer. The big advantage of parallel computers is that the component complexity is kept low. Collections of simpler processors are easier and cheaper to build than one very complicated processor. Increasingly the two approaches are being combined as intra-
Chapter 3. Laziness and parallel Data Structures

In order to reason formally about parallel program behaviour, formal models of synchronisation and communication are needed. In the most general models, programs are described as a set of interacting sequential processes (with encapsulated control and state). The interfaces between the sequential processes are a well defined set of communication channels down the communication channels and receiving results in a programmed exchange. Each task can be viewed as a black box with input channels and output channels defining its interface.

3.1.2 Map function

The essence of data-parallelism is an $O(1)$ map function. A data-parallel interpretation of \((\text{map } f)\) applies \(f\) to every element of a parallel data structure all at the same time. This model is at odds with the conventional interpretation of map on lists. Although map \(f\) can be interpreted as applying \(f\) to every element of a list, in a non-strict language the function applications only occur to those elements required by a subsequent computation. To highlight this dichotomy we investigate the potential for data-parallelism on lazy lists and monolithic arrays.

The function map \((+1)\) in exList of figure 3.1 could be applied in a data-parallel manner to each element of the list \(xs\), because the surrounding sum consumes all of the resulting list. As a general rule, if a map expression is enclosed by a function that is both head and tail strict [75], then data-parallel evaluation of the map is feasible. If we aim to implement this model on a massively parallel SIMD machine, we would have to implement the list as a data-structure whose elements occupy consecutive memory.
locations. This is at odds with the conventional run-time representation of lists using pointers. If lists at first seem unsuitable for data-parallel evaluation, monolithic arrays [38] provide a more obviously practical framework. Figure 3.1 defines the Haskell array-mapping function a map. The function array creates a monolithic array. The first argument defines the bounds of the array, and the second defines a list of associations of the form ‘index:=value’ where the contents of the array at index are defined to be value. The array function is strict in both the bounds and the indices of the association list, but not in the indexed values. As was seen to be the case with map and lists, the non-strictness associated with the values of array elements interacts awkwardly with data-parallel evaluation. For example, because of the properties of take and amap, the reciprocal function (1/) is only applied to the first eight elements of arr. We can see from the definition of exArray that it is possible to determine at compile-time that only the first eight elements of arr are required; does this form a general rule?

```
> map f []  = []
> amap f a = array b [i:=f (a!i)|i<-range b]
>map f (x:xs) = f x: map f xs
>         > where b = bounds a
>
>exList xs
>exArray arr

>= sum (map (+1) xs)
> = sum (take 8 (elems (amap (1/) arr)))
```

Figure 3.1: List & Array Map
Chapter 3. Laziness and parallel Data Structures

We propose an evaluation mechanism that combines the desirable features of the lazy and strict evaluation of map. Whenever a map-like computation is forced, multiple elements of the parallel object being mapped evaluate their results synchronously. However, the mechanism retains non-strict semantics. In the discussion so far, non-strictness and data-parallelism have been mixed only if it is possible to determine at compile-time how much of the object resulting from a map-like computation is required. The model we propose delays this choice to run-time when we know exactly what needs to be evaluated. We maintain a run-time data structure called the aim describing those elements that are required to be evaluated. Whenever a map-like computation is forced the function applications of the map are evaluated in parallel at those elements defined by the aim. map can therefore be implemented as $O(1)$ whilst retaining all the benefits of non-strict evaluation.

3.2 An introduction to the `aim of evaluation'

3.2.1 The print-eval loop: aim = multiple points of interest

The concept of the aim of evaluation arose from the concern that the print-eval loop of a non-strict language unnecessarily serialises and throttles potential parallelism within a program. In its simplest form, the print-eval loop consumes a list of characters by outputting them one-by-one to the screen. In a non-strict language, the implementation
should do only just enough evaluation to generate each character in turn. Given the expression:

\[
\text{concat (map (integerToString . (+1)) [1..])}
\]

the print-eval loop will try to print the first character of the list denoted by the expression. This will force the production of the first element of the natural numbers, and then map will add one to the value forced. The resulting number 2 is converted into a string, and finally, the print-eval loop can print its first character.

The aim of evaluation provides the programmer with a mechanism of explicitly parallelising the print-eval loop. The idea is that instead of having a single point of interest at the head of the output stream, the programmer controls multiple points of interest. For example, in a sequential setting the expression take 8 ['a'..'z'] prints the characters abcdedefgh one after the other. In a parallel scenario, the same effect can be achieved by creating a parallel array, with a lower bound of 1, containing the characters 'a' through to 'z'. If the programmer sets the aim to identify array elements 1 through to 8, and evaluates the array of characters, then the first eight elements of the array can be evaluated in parallel.

### 3.2.2 The aim as an abstraction of a SIMD activity mask

When using a SIMD machine computation is expressed in terms of parallel operations on monolithic array-like data-structures. These array-like structures are often distributed across the processors of a machine. In a typical mapping each processor contains a single element of an array. When programming in a language for a SIMD machine, the
programmer is aware of the notion of an active set or activity mask which controls where computation occurs. For example, given two integer arrays to be divided element-wise, if any element of the divisor contains zero, then an error will be raised for the entire computation. Conventional imperative SIMD languages overcome this problem by setting the activity mask before troublesome statements are executed. In the context of division by zero, the activity mask would be set at those processors of the divisor that do not contain a zero, and then the division can be safely executed in parallel. From a machine perspective, the activity mask manifests itself in the hardware in terms of a register in each processor which controls whether the processor should be part of the lock-step evaluation of a SIMD program.

The activity mask used by the imperative SIMD programmer is therefore an abstraction of the activity registers of each of the processors of a SIMD machine, and the aim of evaluation is an abstraction of the activity mask.

3.2.3 An operational view of the aim of evaluation.

The aim forms an integral part of the data-parallel evaluation mechanism we propose. It is continually recalculated to identify just those elements of an array that are needed during each step of evaluation. We illustrate this recalculation by considering four typical scenarios where the aim plays an important role in data-parallel evaluation.
Chapter 3. Laziness and parallel Data Structures

In the discussion that follows, terms of the form expr represent arbitrary expression which when evaluated produce an array that is distributed throughout the processors of a parallel machine. The functions mapArray, zipWithArray, and filterArray are analogous to the standard list-processing functions with similar names. However, the marked difference is that these functions require parallel arrays as arguments, and produce arrays as results.

The aim stays the same The evaluation of:

\[
\text{mapArray integerToString expr}
\]

where expr evaluates to an array, produces an array in which each processor contains a textual representation of an integer. If evaluation only requires a subset of the array elements, then before the map can be applied, that same subset of elements from expr will need to be evaluated. Therefore given an aim ff for the entire computation, the lock-step evaluation will start by evaluating expr with the aim ff, and then the same set of processors will have their integer contents converted into a string. In general, map transmits the aim unchanged.

Splitting the aim of evaluation Where there are different control paths within a program, the aim may change, splitting itself to mirror the control paths. Considering a single route through the tree of all the different control paths, the aim will steadily narrow during each step of a map computation. For example, zipWithArray can be used to map the four argument choice function across two integer arrays:
Chapter 3. Laziness and parallel Data Structures

let choice :: (Int->Int) -> (Int->Int) -> Int -> Int -> Int

choice f g 0 y = g y

choice f g x y = f x

in zipWithArray (choice (+1) (+2)) exprA exprB

In a data-parallel setting, the pattern matching involved with the evaluation of choice can be performed in parallel on the entirety of both expr A and exprB. If the above let-expression is evaluated with aim α, then because of the pattern matching of 0 required by the first equation, those processors identified by the aim α from the array exprA will be evaluated in parallel. The lock-step evaluation of zipWithArray (choice (+1) (+2)) will then proceed by identifying those processors within α that matched the pattern 0—they make up a new aim β (a subset of α) which is used in the parallel evaluation of g y. As g is bound to the strict function (+2), all of the ys (namely the array exprB) within the aim β will need to be evaluated, and then the computations of (+2) y can all occur in parallel.

Once the first equation has completed evaluation, those elements of exprA that did not match the first equation can proceed to evaluate the second default equation. Again a narrowing of the aim will occur, where a new aim α - β is created that identifies those elements of α that did not match against the pattern 0. This aim will then be used to evaluate the expression f(x). As f is instantiated to the increment function (+1), then all the xs (namely the array exprA) will be evaluated with the aim α - β. Due to the initial pattern matching of choice, the array exprA will already be evaluated at a superset of the aim.
Chapter 3. Laziness and parallel Data Structures

required (\(\alpha - \beta \leq \alpha\)). Evaluation of exprA therefore finishes immediately, and then\((+1) \times \)
occurs in parallel at processors identified by \(\alpha - \beta\). Finally, the evaluation of choice is
completed by merging the arrays produced by the two pattern matching equations.

In summary, the distinguish feature of this map-like evaluation is that if the
function being mapped has \(n\) different pattern-matching equations (i.e., control paths),
then the aim used by the map is split into \(n\) pieces. Each of the new aims models the
processors which matched a particular pattern matching equation. All of the right-hand
sides of the equations are then evaluated sequentially, one after the other, with the newly
created aims (This model of evaluating pattern matching equations is aimed towards the
lock-step evaluation required by a SIMD machine. Other machine models would use a
scheme that is better utilised for their own architectural features. For example, an MIMD
machine could create the aim for each of the control paths, and then evaluate all the right-
hand sides concurrently). The resulting arrays are finally merged together to produce an
array that encapsulates the entire computation.

Growing the aim There are often circumstances in which the aim dramatically grows to
mirror the data-dependencies that often occur within function definitions. For example, if
the first ten elements of the array produced by the following expression are required:

\[
\text{filterArray even expr}
\]

then it would be incorrect to evaluate expr at the same ten processors. What will be
required is to evaluate expr just enough such that its first ten even elements are accounted
Chapter 3. Laziness and parallel Data Structures

for. Therefore interwoven with the filtering will be a "growing" of the aim such that it
encapsulates just enough of expr to provide the ten elements initially requested. The steps
involved in the evaluation of filterArray are: (1) the aim is set for the filter expression as a
whole; (2) filterArray grows the aim---not necessarily one element at a time---evaluating
expr until all the elements that are to be filtered are encapsulated; (3) the
evaluated elements of expr are finally collected into an array in which the filtered elements
occupy consecutive array elements. In the definition of a parallel filterArray, growing of
the aim is controlled by a send parallel data-communication primitive. Given an array to be
filtered, parallel filterArray calculates a destination for the contents of each
processor, such that the resulting array contains just the right elements from the original.
The send used to perform this communication implicitly controls steps 2 and 3 outlined
above.

Narrowing the aim The final example highlights that many processors can often share
evaluation of a common set of processors. Consider the array denoted by the sequence:

\[ B = [A 1; A 2; A 2; A 3; A 3; A \ldots] \]

in which the first element contains the first element from array A; the next two elements
are from the second element of A; the next three elements are from the third element of
A, etc. If the first 10 elements of B are to be evaluated, then only 4 elements of A will
be required. Similarly, 5151 elements from B will only require 100 elements from A. The
effect of each processor selecting the contents of other processors, where there is the
potential for many processors accessing a common processor is achieved using a fetching
Chapter 3. Laziness and parallel Data Structures

parallel data-communication primitive. Like the send communication briefly mentioned above, the implementation of fetch changes the aim such that only the processors which have their contents "fetched" are evaluated.

This thesis is concerned with the implementation of the aim such that it models the four features outlined above. Unfortunately the presentation of the material will be in a form that bears little similarity to that here. One reason for this is that so far we have considered the interaction of the aim with Haskell functions. In contrast, the approach adopted in the rest of the thesis is to take a layered approach that translates Haskell into a sugared form of the lambda-calculus. The benefit of this approach is that it is easier to reason about the aim of evaluation, and the mechanics of evaluation with the aim, in terms of transformations and ultimately the evaluation of the lambda calculus.

3.2.4 Data-parallel non-strict glue

Given that the aim is one way of achieving a non-strict semantics within a data-parallel setting, we propose that the aim enables existing techniques that utilise non-strictness to be carried through to a parallel environment. In a combinatorial search problem he showed how the generation of the search space can be decoupled from the process that searches the generated space. This decoupling is achieved by defining a function that generates a potentially infinite search space, and a different function that traverses the generated data-structure looking for solutions to the combinatorial search problem. When the two functions are composed, non-strictness ensures that just those parts of the search space
Chapter 3. Laziness and parallel Data Structures

required to find a solution are generated---non-strictness therefore provides a special kind of "glue" for combining sub-problems.

Using the aim of evaluation as a method of achieving data-parallelism with a non-strict semantics, we can utilise the same technique in a data-parallel setting. Because of a non-strict semantics it is possible for data-parallel arrays to be potentially infinite. A generation function for the search problem can therefore be defined to distribute the search space throughout the consecutive elements of a potentially infinite array. If a consumer function identifies those points within the search space that are valid combinatorial solutions, then the valid solutions can be filtered to occupy consecutive locations of a solution array. Just as Hughes observed, the generation and consumption of the search space can be achieved by composing the producer and consumer functions together. If only the first solution is required, then the aim would be set to the first element, and the evaluation would grow the aim, effectively searching for the single solution within the search space in parallel. However, the benefit of using the aim of evaluation is that if 50 solutions are required, the aim would be set from elements 1 through to 50 of the parallel solution array, and because of the multiple points of interest, the fifty solutions will be found in parallel. The expansion of the search space for each of the fifty solutions will therefore occur concurrently, whilst the entire computation retains a non-strict semantics.
3.3 PODS

All parallelism in Data Parallel Haskell (DPHaskell) is achieved by operations on parallel data structures called pods. A pod represents a collection of index/value pairs, where each index uniquely identifies a single element of a pod. As pods are an abstraction of the processing elements of a data-parallel machine, we choose to collect the index value pairs into a data type we call a "processor". For example, (|42;"DON'T PANIC"|) represents a single processor of a one-dimensional pod (a vector), where (| and |) are used as delimiters in the same way as brackets are used to delimit a tuple. The expression determines that the value of the pod at a position identified by 42 is "DON'T PANIC". At the beginning of our work, we envisaged higher-dimensional pods, characterised by multiple indices to identify a single element within the pod. The general form of a processor would therefore be (|e₁, . . . ,ek ; e|) where k >= 1, and the sequence of expressions e₁ to eₖ would uniquely identify an element in a k-dimensional parallel object; e would describe the data in that element. As a consequence of having multi-dimensional parallel objects the name pod is derived from the acronym Parallel. Objects with arbitrary Dimensions. However, because the use of pods is tightly coupled with the pod comprehension notation presented below, we have found that ambiguities arise in the semantics of higher-dimensional pod comprehensions, so we restrict pods to a single dimension. A pod contains a sequence of uniquely labelled processors, any of which may be missing. The empty pod that contains no processors does not necessarily consume no resources when an implementation maps a pod to the real processors of a data-parallel
Chapter 3. Laziness and parallel Data Structures

machine. At the language level we do not specify how pods should be implemented in terms of either sparse or dense parallel data-structures. Therefore the empty pod, or in keeping with our analogy, a mange-tout, may consume the same resources (i.e., space) as a pod in which every processor is defined. In summary, pods are different from ordinary monolithic arrays because they contain holes where processors are missing.

PODs also differ from arrays in a more fundamental way because their size is unbounded and potentially infinite. Because of the semantics of a non-strict language, where evaluation only occurs at the point that it is required, the evaluation of a finite portion of an expression that denotes an infinite object is possible. In an implementation, a dense finite object is created, with a size equal to the difference between the smallest and largest indices defined by the aim. If pods are re-evaluated with a different aim, then the representation of a pod grows accordingly.

3.4 POD comprehensions

Figure 3.2(a) defines a function that negates each element of a list using a Haskell list comprehension. This syntax provides a good starting point for a parallel notation because it decomposes a problem into the transformations that occur at each element of a list. Because there is no implied sequencing to the transformations that could result in dependencies between each of the elements of a parallel data-structure, then the transformations can be applied independently all at the same time. Figure 3.2(b) defines a pod comprehension corresponding to the list comprehension of figure 3.2(a). The desired
Chapter 3. Laziness and parallel Data Structures

reading of the negateV function is "For each defined processor in vec identified by y, that contains data x, create a one-dimensional pod such that each processor at y contains the data -x ". Examining the definition, we see that only '-' is specific to the negateV function. This means that the computation of the vector negate can be modularised by gluing together a general pod comprehension and a function that is applied to each defined element of a pod. By parameterising the definition of negateV on the function applied to each element of the pod, we derive \text{mapPod} defined

(a) List comprehension \hspace{1cm} \text{negateL} \hspace{0.5cm} xs = [ -x \mid x \leftarrow xs ]

(b) POD comprehension \hspace{1cm} \text{negateV} \hspace{0.5cm} vec = << (y; -x) \mid (y;x) \leftarrow vec >>

(c) Redefinition of (b) using \hspace{1cm} \text{mapPod} \hspace{0.5cm} f \hspace{0.5cm} vec = << (y;f x) \mid (y;x) \leftarrow vec >>

\hspace{2cm} \text{higher-order pod map} \hspace{1cm} \text{negateV} \hspace{0.5cm} vec = \text{mapPod} \hspace{0.5cm} (x \rightarrow -x) \hspace{0.5cm} vec

Fig. 3.2 : From lists to vectors .... mapping negate in parallel

in figure 3.2(c). As the motivation behind the aim evaluation mechanism was the efficient parallel implementation of a non-strict map, it is not surprising that mapPod has a $O(1)$ time complexity in relation to the size of the vector being mapped.

The semantics of list comprehensions is very much tied to set theory, in such a way that whenever multiple generators are used in a comprehension, all possible combinations of values are produced as a result. For example the expression
Chapter 3. Laziness and parallel Data Structures

\[
[ (x,y) \mid x \leftarrow [1..10], y \leftarrow [1..10] ]
\]
generates the cartesian product of two lists. More surprisingly

\[
[ x \mid x \leftarrow [1..10], y \leftarrow [1..10] ]
\]
generates a list with 100 elements---the computational interpretation of generators is very much tied to iteration. Since we believe that effective use of a parallel machine with thousands of processing elements can only be attained with pods containing thousands of elements, a strategy that relies upon a combinatorial explosion of values must somehow be avoided---the physical constraints of a machine's memory would soon be exhausted if pods contained millions of elements. The solution we adopt to this problem is two fold: (1) we ensure that for each element drawn from a generator, only one element will be drawn from subsequent generators---"zip like" drawing; (2) the drawing of elements from the generators of a pod comprehension is performed lazily such that values are drawn only as required.

Addressing the first of these issues, we exploit the fact that the index of a pod element is unique. If we draw an element \((i;v)\) from a pod, we can be sure that we will not be able to draw any other element with index \(i\). We may however wish to draw an element \((i;w)\) with the same index \(i\), from some other pod in a different generator. We therefore propose two versions of the generator: \((p;p_d;!) \leftarrow POD\) is read as drawn-from and names in the patterns \(p\) and \(p_d\) are bound by the generator; \((e;p_i;!) \leftarrow POD\) is read indexed-from and the value represented by the expression \(e\) determines which processor's contents should be matched against the pattern \(p_i\). Drawn-from generators provide a point of interest.
Chapter 3. Laziness and parallel Data Structures

Comprehensions contain at most one drawn-from generator as multiple points of interest make little sense for subsequent generators in a comprehension. Therefore the expression x in the index generator \((x;b) \ll= \text{vecB}\) of figure 3.3(a) is bound by the pattern x in the preceding drawn-from generator. The intended semantics are that the constraint of the generators ensures zip-like drawing; the comprehension therefore defines a vector addition operation. Comprehensions

(a) \(\text{addV vecA vecB} = \langle (x;a+b) | (x;a) \ll= \text{vecA}, (x;b) \ll= \text{vecB}\rangle\)

(b) \(\text{shiftAnd vec} = \langle (x;a\&\&b) | (x;a) \ll= \text{vec}, (x-1;b) \ll= \text{vec}\rangle\)

(c) \(\text{negateV' vec} = \langle (x;-y) | (x;y) \ll= \text{vec}, (x;z) \ll= \perp\rangle\)

Figure 3.3: The lazy semantics of pod comprehensions

are not just restricted to simple zip-like drawing. The index-generator in figure 3.3(b) defines a constraint \(x-1\) which results in communication between elements of the pods—we return to communication a little later.

The second issue, crucial to the semantics of generators, is lazy drawing. Using \(\ll=\) draws elements from processors in a strict manner—the processors have to be defined for a result to be defined. However \(\ll=\) draws processors and their contents from a pod lazily.
Chapter 3. Laziness and parallel Data Structures

depending upon the strictness characteristics of their subsequent use. An analogy can be
made between this model and refutable and irrefutable (strict and lazy) pattern matching in
Haskell. The definition of negateV shown in figure 3.3(c) has the same semantics as the
earlier definition of negateV in figure 3.2(b), since the variable z is not used in the
comprehension. In contrast a similar list comprehension below tries to draw elements from
?, resulting in the comprehension evaluating to ⊥.

\[-x | x \leftarrow [1..10], y \leftarrow \perp\]

Figure 3.3 (b) shows a more subtle example in which processors and their contents only
have to be defined depending upon the strictness of &&. Given a processor identified by x
that contains False, due to the laziness of &&, neither the processor nor its contents at a
position identified by x-1 need to be defined. We clarify such semantics in fig 3.6.
The constraint used in the function shiftAnd of figure 3.3(b) causes communication
between elements of the vector vec. We generalise such communication into two models:
fetching data from a remote processor to a local processor, and sending data from a local
processor to a remote one. The first expression shown in figure 3.4 uses the sending model
of communication. Data stored in a processor identified by x is sent to a processor at
location x+2. The effect is to shift every defined processor's contents two places to its
right. In contrast, the expression on the right of figure 3.4 uses the fetching model of data
communication. The value x bound by the drawn-from generator (\<-\) represents the
processor in which data will be placed. The data is fetched from the contents of a processor
determined by the constraint on the index-generator (\<=\). By drawing elements from the
Chapter 3. Laziness and parallel Data Structures

infinite pod in the drawn-from generator (every processor and index is defined), a binding occurrence for \( x \) is provided for subsequent generators. Although the use of \( \text{inf} \) looks clumsy, rarely is such trickery required in a fetch computation see for example, the fetch used in the definition of parallel scan in fig 4.1.2.

As we shall see in the denotational semantics, the clumsiness has

\[
\begin{align*}
\text{let } f &= \lambda z \rightarrow z + 2 \\
\text{let } f' &= \lambda z \rightarrow z - 2 \\
in &\text{ inf } = \langle\langle ..42.. \rangle\rangle \\
\langle\langle (f \ x; y) \mid (x; y) \langle\langle \text{vec} \rangle\rangle \rangle &\text{ in } \langle\langle (x; y) \mid (x ;\_\_\_ \_\_) \langle\langle \text{inf} \rangle\rangle \\
(f' \ x; y) &\langle\langle \text{vec} \rangle\rangle
\end{align*}
\]

Figure 3.4: The duality of send and fetch

some theoretical backing, as in \( f \) is used to create a binding occurrence for elements in \( Z \), resulting in a definition which looks similar to the denotational semantic definition of fetch. In these two examples, as the lambda expression used in the fetch is the inverse of that used in the send, the two expressions are equivalent. This equivalence between sending and fetching expressions does not always hold as the inverse to a function used in data communication may not exist. For example, with the sending expression
Chapter 3. Laziness and parallel Data Structures

\[
\langle \langle \langle f \ a, v \rangle \mid \langle a, v \rangle \rangle \rangle \langle \langle - \ vec \rangle \rangle
\]

vec may contain \( \langle x, vx \rangle \) and \( \langle y, vy \rangle \) where \( f x \in f y \). In this case, either \( \langle f x, vx \rangle \) or \( \langle f y, vy \rangle \) might appear in the solution, though not both since an index must identify a unique value. An implementation may choose either solution (see fig 3.5.5).

Constraints of a different nature can also be applied to pod comprehensions in the form of boolean guards. These guards act as filters that select those processors for which the guard is True from a more general stream created by the generators. For example the expression

\[
\langle \langle \langle x, X \rangle \mid \langle x; 42 \rangle \rangle \rangle \langle \langle - \ vec, even x \rangle \rangle
\]

defines a pod in which each processor contains its identifier as long as that processor originally contained the number 42 and its processor identifier is even. A collection of higher-order functions based upon the examples in this section are shown below. The mapPod function of figure 3.2, is generalised to form a zip-with computation by using a drawn-from generator in a pod comprehension. The sendPod and fetchPod functions are based upon the communication functions of figure 3.4. sendIPod is similar to sendPod, except that an index-pod (i.e., a pod containing elements that identify the processors in another pod) is used to specify the processors where communication occurs to and from.

\[
> \text{mapPod} :: \text{Pid} x \Rightarrow (a \rightarrow b) \rightarrow \langle \langle x; a \rangle \rangle \rightarrow \langle \langle x; b \rangle \rangle
\]

\[
> \text{mapPod} \ \text{fn vec} = \langle \langle x; \text{fn} y \rangle \mid \langle x; y \rangle \rangle \langle \langle - \ vec \rangle \rangle
\]

\[
> \text{zipWithPod} :: \text{Pid} x \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow \langle \langle x; a \rangle \rangle \rightarrow \langle \langle x; b \rangle \rangle \rightarrow \langle \langle x; c \rangle \rangle
\]
Chapter 3. Laziness and parallel Data Structures

> zipWithPod fn vecA vecB = << (lx; fn y zl) | (lx;yl) <= vecA
> (lx;zl) <= vecB >>
>
> fetchPod :: (Pid x, Pid y) => (x -> y) -> (x;yl) <= vecA
> fetchPod fn vec = << (lx;yl) | (lx;zl) <= vec ,
> (lx;yl) <= vec >>
>
> sendPod :: (Pid x, Pid y) => (x -> y) -> (x;yl) <= vecA
> sendPod fn vec = << (lx;yl) | (lx;yl) <= vec >>
>
> sendIPod :: (Pid x, Pid y) => (x;yl) -> (x;yl) <= ivec
> sendIPod ivec vec = << (lx;yl) | (lx;yl) <= ivec ,
> (lx;yl) <= vec >>
>
3.5 The semantics of POD comprehensions: primitive operations

The formal semantics of pods and pod comprehensions is given in terms of translation rules that produce Haskell enriched with the primitive parallel operations map n, indices, fetch and send. The rules provide an insight into the implementation of the data-parallel extensions by using a "concrete" representation of parallel objects, where all communication is performed by primitive data-rearrangement operations. The denotational
Chapter 3. Laziness and parallel Data Structures

semantics of the parallel data-structures and primitive operations is given in Chapter 3. Here we give an informal description of the primitives, along with definitions in terms of pod comprehensions.

3.5. 1 \perp and the representation of PODs

Before describing the primitive parallel operations, we set the scene by highlighting various problems and peculiarities associated with \perp (bottom) in data-parallel programs. Figure 3.5(a) defines an expression with a value equivalent to \perp. Evaluation at an undefined processor of a pod results in \perp; the expression in figure 3.5(b) is therefore interpreted as

(a) \begin{align*} & \text{let bot = bot in bot} \\
(b) & \langle\langle (1;2),(3;4) \rangle\rangle \end{align*}

(d) let f 0 = bot |

(b) \begin{align*} & \langle\langle (1;2),(3;4) \rangle\rangle \rangle \end{align*}

(f n = n)

(c) \begin{align*} & \langle\langle (1,2), (2; \text{bot}), (3;4) \rangle\rangle \end{align*}

\begin{align*} & \text{in} \langle\langle \text{if } x,y \rangle | (x;y) \langle\langle -\text{vec} \rangle\rangle \end{align*}

Figure 3.5: A selection of bottoms

only defining values for processors one and three. The semantics of (b) differs from the pod shown in figure 3.5(c), although indexing (b) and (c) at any index produces the same values. The difference between the two expressions is highlighted by mapping \lambda x \rightarrow 42
Chapter 3. Laziness and parallel Data Structures

over each pod. Indexing the resulting pods at processor two results in \( \perp \) for (b) but 42 for (c). Figure 3.5(d) shows a peculiar case of bottom. If the expression in the index position of a processor on the left-hand side of a comprehension results in bottom (such as processor \( f \: 0 \) in (d)), the effect is to erase that processor's contents from the resulting pod representation—a pod can never be indexed at processor \( \perp \).

Keeping such characteristics of pods in mind, we present a representation of a pod suitable for implementation on a SIMD machine. A conventional non-strict evaluation mechanism enables expressions such as (a) to be manipulated. An extended evaluator based upon the aim mechanism allows expressions such as (c) and (d) because only those processors defined by the aim are ever evaluated. Unfortunately, (b) poses problems. pods are implemented as extensible, dense, array-like structures—if processors \( x-1 \) and \( x+1 \) exist, then processor \( x \) exists. Such an “implementation”pod or vector has no representation for the semantics required in figure 3.5(b). Therefore when vectors are used to model pods, the vectors need to encode a representation of “not here” for any undefined processors. A solution to this problem is to represent pods by the product type:

\[
\text{Data}<<([\text{Int};\alpha]) \gg \equiv \text{MkPod} \ll \text{Bool} \ll \alpha
\]

where \( \ll \alpha \) represents an implementation vector of type \( \text{ff} \). Wherever the value of an entry in the boolean mask defined in the first part of the product type is True, the corresponding element in the second vector has a defined processor—“I'm here”. Although it seems that a lot of trouble has been expended on fulfilling the desired semantics of figure
Chapter 3. Laziness and parallel Data Structures

3.5(b), they ensure an important invariant that is required in the implementation of map \( n \) (see fig 5.1.2).

3.5.2 map

The primitive function map\( n \) is analogous to the family of list functions, map, zipWith, zipWith3, etc. The drawing of elements occurs as required in a zip-like manner by index. All the function applications of the map occur synchronously and in parallel at those processors defined by the aim of evaluation. Informally, the semantics of map1 and map2 are defined by the pod comprehensions below, where vec has a vector type, and not a pod type:

\[
\text{map1} :: (\alpha \rightarrow \beta) \rightarrow \langle \alpha \rangle \rightarrow \langle \alpha \rangle
\]

\[
\text{map1 } f \text{ vec } = \langle \langle (x;f y) \mid (x;y) \rangle \langle \alpha \rangle \rangle
\]

\[
\text{map2} :: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \langle \alpha \rangle \rightarrow \langle \beta \rangle \rightarrow \langle \gamma \rangle
\]

\[
\text{map2 } f \text{ vecA vecB } = \langle \langle (x;f y z) \mid (x;y) \rangle \langle \alpha \rangle \rangle ; \langle (x;z) \rangle \langle \beta \rangle \rangle
\]

3.5.3 indices

A pod comprehension manipulates the indices of a pod as integers. As with arrays, these indices are an abstraction of the contiguous nature of a machines memory. Given the index \( n \) of the first pod element, the \( (k + n) \)th pod element will be at a fixed offset \( k \) from the prior element. As the indices are an implicit part of the pod representation, the primitive function indices "recreates" the processor number by converting a vector of booleans into a
vector in which each processor contains an integer that represents the processor number
informally indices can be defined by the vector comprehension:

\[
\text{indices} :: \langle \text{Bool} \rangle \times \langle \text{Int} \rangle \\
\text{indices mask} = \langle (x; \text{if } m \text{ then } x \text{ else error "Bottom" } ) \rangle (|x; m|) \langle -\text{mask} \rangle
\]

### 3.5.4 fetching

An index vector is a vector of integers which is used to identify the processors in which
communication occurs to and from. Evaluation of fetch ivec data, where \( (x; i) \in \text{ivec} \)
fetches the contents of processor \( i \) from data, and places it in processor \( x \). The novel feature
of fetching communication is that it is inherently lazy. In the implementation, if a fetching
primitive is evaluated with an aim \( \alpha \) then only those processors of \( \text{ivec} \) defined by \( \alpha \) are
evaluated. Once evaluated, this vector is used to construct a new aim \( \beta \) which is used in the
evaluation of data. Once data is evaluated, communication occurs that transfers the
contents of each of the evaluated processors into a destination specified by the original
index vector \( \text{ivec} \). Informally the semantics can be expressed as the pod comprehension
below.

\[
\text{fetch} :: \langle \text{Int} \rangle \rightarrow \langle \text{Int} \rangle \rightarrow \langle \text{Int} \rangle \\
\text{fetch ivec data} = \langle (x; y) \rangle \langle (x; i) \rangle \langle -\text{ivec}, \langle i; y \rangle \rangle \langle -\text{data} \rangle
\]
Chapter 3. Laziness and parallel Data Structures

3.5.5 sending

The sending primitive that has the the informal semantics shown by the pod comprehension below has two problems associated with it: (1) sending does not fit into the 'aim' model of evaluation—a search of a potentially infinite pod is required; (2) multiple processors may send their contents to the same processor, it is not clear how this should be resolved.

\[
\text{send} :: \langle \text{Int} \rangle \rightarrow \langle \alpha \rangle \rightarrow \langle \alpha \rangle
\]

\[
\text{send ivec data} = \langle\langle (j; y) \mid (x; i) \rangle\rangle \langle\langle ivec, (x; y) \rangle\rangle \langle\langle data \rangle\rangle
\]

The first of these problems is a result of incorporating laziness into a data-parallel language. Aims are a technique of ensuring that map can be implemented in a synchronous manner on a data-parallel machine. In the implementation, a data-parallel evaluation mechanism threads the aim throughout a program, continually calculating which set of processors needs to be evaluated. The sending primitive throws a spanner into the works of the aim mechanism.

If the primitive send ivec data is evaluated with an aim ff, then the index vector ivec is inverted, and the primitive fetch ivec-1 data is performed; the essence of send is the runtime calculation of which processors are needed to convert the send into a fetch.

Unfortunately inverting the index vector is not simple as ivec is potentially infinite. Given
Chapter 3. Laziness and parallel Data Structures

the aim ff, ivec is evaluated in such a way that for each \((x; i) \in ivec\), then every processor identified by the aim is accounted for by each \(i\) of ivec. If no processor sends its data to a processor identified by the aim, then evaluation of the index vector may continue for a very long time. This is the motivation behind the "bottom" semantics for undefined processors-the search of the index vector in a sending communication may not terminate.

The second problem associated with sending has been addressed many times elsewhere \([65,66]\). The common technique used to resolve collisions in a send is to apply an associative binary operator to all colliding data. As this technique is potentially usefull, some parallel

\[
\begin{align*}
\text{aexp} & \perp \ll ((\text{exp};\text{exp})) | \text{pqual} \ 1 \ , \ : \ : \ , \text{pqualn} \gg \ (\text{POD comprehension}) \\
\text{pqual} & \perp \text{exp} \ (\text{filter}) \\
| \ (\text{var};\text{var}) & \ll - \exp \ (\text{drawn-from}) \\
| \ (\text{exp};\text{var}) & \ll - \exp \ (\text{indexed-from})
\end{align*}
\]

Figure 3.6: Syntax of POD comprehensions

computer manufactures have provided communication hardware for making collisions more efficient \([11]\). However, each list will not be nil (i.e \([\ ]\)) terminated---the tail of the list will always contain bottom as we cannot determine when a potentially infinite set of processors have stopped sending data. This means strict functions such as + cannot be used to resolve collisions. The solution we adopt is to choose a single value from the
colliding data. Although this implies a non-deterministic semantics, an implementation on a SIMD machine will always be deterministic.

3.6 The semantics of POD comprehensions: desugaring

The semantics for pod comprehensions (syntax shown in figure 3.6) is presented in-terms of a series of translation rules that "desugar" the comprehensions into the primitive parallel operations presented in the previous section. The motivation behind this desugaring is the conversion of the "microscopic" view of transformations applied to the elements of a pod, into a "macroscopic" view of applying monolithic operations to the pod as a whole. This translation defined by schema TPOD, provides the foundations for the vectorisation of functional programs described in Chapter 5. The initial state of the translation scheme is TPOD [[pod]] [ ] «...True ...», and to eliminate scoping problems the patterns in generators are assumed to be unique variables. The translation scheme uses the following information for book-keeping purposes:

1. a mapping from bindings in a source language comprehension that represent elements of a parallel object, to bindings in the translated language that represent entire parallel objects. For example, given the expression 'x + 2' where x represents an element of a parallel object, the translation produces expressions of the form map1 (λ x → x + 2) x in which x represents the entire parallel object that x is an element of; the mapping x → x is recorded during translation. When the notation binds[x → y] is used on
Chapter 3. Laziness and parallel Data Structures

the top line of a translation rule it defines the mapping $x \rightarrow y$ as a member of the
environment. When it is used on the right-hand side of the translation, it defines that
the environment is extended with the mapping $x \rightarrow y$.

2. a mask which defines the valid processors in the pod resulting from the
comprehension.

3.6.1 Drawn-from

Drawn-from generators provide a mechanism of anchoring a point of interest for
subsequent generators in a comprehension. Those processors defined in the pod being
drawn-from are used to define the processors in the pod that results from the
comprehension. Translation rule 1 below encapsulates such semantics. Case analysis is
first performed on the pod on the right-hand side of the generator to expose its
implementation structure. The exposed mask that represents the defined processors of vec
is then threaded through successive calls to the translation scheme, eventually re-emerging
in rule 4 to define the valid processors of the pod that results from the comprehension.
Notice how the $x$ and $y$ in the translated code represent all the indices and contents of vec,
whereas $x$ and $y$ in the original comprehension represented a single index and element of
the pod vec. The implementation structure of pods is reinforced by this rule, as a vector of
integers that represents vec's indices is created by the indices primitive.
Chapter 3. Laziness and parallel Data Structures

TPOD \([ [\ll e\| (| x ; y |) \ll vec , q ] ] \) binds mask junked

\[= \text{case vec of} \]

\[\text{MkPod mask } y \to \text{let } x = \text{indices mask} \]

\[\text{in TPOD } [[\ll e \to q ] ] \text{ binds } [x \to x ; y \to y] \text{ mask} \]

Rule-1

3.6.2 Indexed-from

Indexed-from generators provide a mechanism of expressing communication in DPHaskell. They have non-strict semantics because elements are drawn from the pod on the right-hand side of the generator as required. We achieve this in translation rule 2 by using the non-strict properties of let expressions and irrefutable pattern matching (the \(\sim\) symbol used in the case expression of rule 2). Elements are 'logically' drawn from vec only when \(y\) is evaluated in an inner scope of the comprehension. Unlike rule 1, the defined processors of vec are not used to define the processors of the resulting comprehension. In other data-parallel languages a zip-like computation results in a parallel object whose extent is defined by the intersection of the argument objects. In DPHaskell, the resulting pod will have processors defined wherever they are defined in the drawn-from generator. Any further restrictions caused by the index generators depend upon the strictness characteristics of the functions that force the index generator's vector. This is achieved in rule 2 by suspending the "processor exists" check within the let
Chapter 3. Laziness and parallel Data Structures

expression identified by y. When y is forced, a fetch communication occurs that evaluates
the index vector represented by the expression e f. This vector is then used to create the
aim for evaluation of data'. Only at this point in the whole computation is the processor
exists check performed, and data' evaluated.

\[
\text{TPOD} \left[ \left[ \langle\langle e | \langle\langle \text{ef}; y \rangle \rangle \langle\langle \text{vec}, q \rangle \rangle \right] \{b_1 \rightarrow b_1 \ldots b_n \rightarrow b_n\} \text{mask} \right] = \text{case vec of} \right.
\]
\[
\sim(\text{MkPod mask'} \text{data'}) \rightarrow
\]
\[
\text{let y = fetch (mapn ('b_1 \ldots b_n \rightarrow \text{ef}) b_1 \ldots b_n) (map2 ('m \rightarrow \text{if m then d else } \perp) \text{mask'} \text{data'})}
\]
\[
\text{in TPOD} \left[ \left[ \langle\langle e | q \rangle \rangle \right] \{y \rightarrow y; b_1 \rightarrow b_1 \ldots b_n \rightarrow b_n\} \text{mask} \right]
\]

Rule 2

3.6.3 Filtering

Translation rule 3 for filtering expressions is relatively straightforward. As a filter restricts
those processors of the pod resulting from the comprehension, we apply the logical 'and' of
the mask that represents the defined processors and the filtering expression.

\[
\text{TPOD} \left[ \left[ \langle\langle e | \langle\langle \text{ef}, q \rangle \rangle \rangle \langle\langle \text{vec}, q \rangle \rangle \right] \{b_1 \rightarrow b_1 \ldots b_n \rightarrow b_n\} \text{mask} \right] = \text{TPOD} \left[ \left[ \langle\langle e | q \rangle \rangle \right] \{b_1 \rightarrow b_1 \ldots b_n \rightarrow b_n\} \right]
\]
\[
\left(\text{mapn+1 ('m \rightarrow b_1 \ldots b_n \rightarrow m \& \& \text{ef}) mask b_1 \ldots b_n} \right)
\]

Rule 3
Chapter 3. Laziness and parallel Data Structures

3.6.4 Left-hand side

The base case for the translation scheme is shown in rule 4. An expression such an e1 in the index position of a processor represents a sending type communication. We apply such communication to the mask that represents the defined processors, and the expression e2 that represents the contents of those processors. This new mask and parallel object is finally used to recreate the user representation of a pod that encapsulates the meaning of the original pod comprehension.

TPOD [[ << ([e1 ; e2 ] | >> ) ] (b1 → b1 . . . bn → bn) mask

= let svec = mapn (\b1 . . . bn -> e1 ) b1 . . . bn

in MkPod (send svec mask)

(send svec ( mapn (\b1 . . . bn -> e2 ) b1 . . . bn ))

Rule 4

fig 3.2 hinted at the possibility of simplifying communications expressed in terms of send into semantically equivalent communications using fetch. Rule 4 of TPOD always introduces a sending communication which, with a little effort, can be eliminated from the translated code. For example, in the translation below, the index vector svec used in both send operations is the vector created by the indices function. As the send communicates each processor’s contents to itself, the send primitive can be removed from the translated program.
Chapter 3. Laziness and parallel Data Structures

TPOD [[ << (lx; a&&b) | (lx; a) << vec, (lx-1; b) << vec, f x >> ]] [ ] <<...True...>>

⇒ case vec of

  MkPod mask a ->
  let x = indices mask
  in case vec of

  ~(MkPod mask' data') ->
  let y = fetch (map2(x a -> x -1) x a)
  (map2(m d -> if m then d else >) mask' data'
  in let svec = map3(b x a -> b x a)
  in MkPod (send svec (map4(m b x a -> m && f x) mask b x a))
  (send svec (map3(b x a -> a && b) b x a))

simplify

⇒

case vec of

  MkPod mask a ->
  let x = indices mask
  in case vec of

  ~(MkPod mask' data') ->
  let y = fetch (map2(x a -> x -1) x a)
  (map2(m d -> if m then d else >) mask' data'
  in MkPod (map4(m b x a -> m && f x) mask b x a)
  (map3(b x a -> a && b) b x a)
Chapter 3. *Laziness and parallel Data Structures*

3.7 Conclusions

POD comprehensions provide a mechanism of expressing data-parallel computations. Using an evaluation strategy which maintains a record of the elements of a parallel object that need evaluating, data-parallelism and non-strictness can be incorporated in the same language. In practical terms this means we can ignore problems that arise from composing functions that perform computations over differing sized parallel data structures. By using infinite pods with functions such as mapPod, finite computations can be performed on the resulting 'glued' functions. Unlike existing data-parallel languages, pod comprehensions provide a single framework within which communication and parallelism can be expressed.