Chapter 4

The HOPP Model

The Bird-Meertens Formalism includes a set of functions which have useful data-parallel properties. These functions form the basis for the set of recognised functions in the HOPP model. The *extended* set of recognised functions contains additional functions which are commonly encountered. In this chapter, definitions are given for the recognised functions in the basic set, and the additional functions in the extended set. The choice between introducing additional functions with new implementations or as compositions of existing functions, is guided by formal as well as practical considerations. In theory, a function can be made a recognised one, if the following attributes can be provided:

- A definition for the function in terms of one or more of the existing functions.
- A parallel implementation for the function on each of the processor interconnect topologies catered for by HOPP.
- A cost estimate for each parallel implementation of the function.

In practice, the decision to include a function as a recognised one is additionally based on its usefulness. Also, if the cost associated with the parallel implementation of the newly-coined recognised function is much less than that of its composing functions, then it is probably justified to include it as a recognised function in its own right. In some cases the definition of the function in terms of existing functions may be so contrived that it may be sensible to include it as a recognised function in its own right.

4.1 The Basic Set of Functions

The recognised functions which have been borrowed from the Bird-Meertens Formalism are defined below. The ML-style [MHT89, Tof89] notation is used in all the definitions. Many of the functions are defined informally, but some of the functions are clearer if formally defined. However, either notation has no implications for the implementation, which is sequential.

1. **map** - applies some function f to each element of the argument list.

   fun map f [ ] = [ ]

   map f (x::xs) = (f x) :: map f xs

2. **fold** - combines the elements of a list using a binary operator.

   fold $\oplus$ a $[x_1, x_2, \ldots, x_n] = (\ldots((\mathrm{a} \oplus x_1) \oplus x_2) \ldots) \oplus x_n$

   When a call is made to **fold**, the value of a must be the identity of the f operator. The HOPP model assumes that the argument function f, in the **fold** definition is always associative. If this is not the case, two functions, **foldl** and **foldr**, can be defined, corresponding to operators that are left-associative and right-associative respectively. However, only if the argument function f is associative, can **fold** be implemented in parallel. Different parts of the list are reduced on different processors in parallel. The partial results on the different processors are then combined to produce the final result. If f is not associative, parallel evaluation will produce incorrect results. The assumption of associativity could be removed by making two versions of **fold** available. The onus would then be on the programmer to use the correct version that does not make any assumptions about the binary operator if it is not associative, in which case a sequential implementation would be chosen by the analyser.

The HOPP model provides two versions of **fold** for quite another reason. The following example illustrates the difference between the two versions.

(a) The first example defines a simple function that calculates the sum of a list of integers.

    fun sum xs = fold plus 0 xs

    where, fun plus x y = x + y

    If xs = $[1, 2, 3]$ then a step-by-step sequential execution of **fold** results in the following:

    i. fold plus 0 $[1, 2, 3]$
    ii. fold plus (plus 0 1) $[2, 3]$
    iii. fold plus (plus 1 2) $[3]$
    iv. fold plus (plus 3 3) [ ]
    v. 6

    It should be noted here that the size of the emerging result remains constant in every step of the **fold** operation.

(b) The second example defines a function that flattens a list.

    fun flat xs = fold app [ ] xs

    where, fun app xs ys = xs @ ys

    Let xs = $[[1, 2], [3, 4]]$.

i. fold app [ ] $[[1,2],[3,4]]$

ii. fold app (app [ ]$[1,2]$) $[[3,4]]$

iii. fold app (app $[1,2]$ $[3,4]$) [ ]

iv. $[1,2,3,4]$

In this case, it is clear that the size of the emerging result *grows* by an amount equal to the size of each list element, after every step of the **fold** operation.

The function **s_fold** (*static*) is introduced to express computations typified by the example in 2a, and similarly the function **g_fold** (*growing*) expresses computations of the type in example 2b. Syntactically, there is no need to distinguish between the two types of computations. However, the execution cost for a parallel implementation of **fold** also includes communication costs and this would depend on the size of the data being communicated. Given a list with n elements, each of size m, then computations in 2b result in a list of size nm elements. At each step of the **fold** operation, the size of the result grows by m. Example 2a results in a list of size m. It may be noted that in the case of **g_fold**, the argument data structure is at least a list of lists.

In a parallel implementation of **fold**, partial results are communicated to neighbouring processors. The size of the data being communicated at each step has a significant effect on the execution cost and is crucial in determining the choice of parallel implementation. By distinguishing between the two versions of the **fold** function, the programmer is invited to provide further cost information to the analyser which enables a more accurate prediction of execution costs. If the version of the **fold** function is not indicated, then the default assumption is **s_fold**.

3. **scan** - similar to **fold**, but the partial results after each application of the binary operator are also retained in the resulting list.

   scan f a [ ] = [a]

   scan f a (x::xs) = [a] @ scan f (f a x) xs

   Again, f is assumed to be associative. Otherwise two functions, *scanl* and *scanr*, must be defined. A **scan** can be implemented in parallel only if the argument operator is associative. As in the case of **fold**, two versions of the **scan** function are introduced - **s_scan** and **g_scan**.

4. **filter** - removes the elements that do not satisfy a property p, from its argument list.

   filter p [ ] = [ ]

   filter p (x::xs) = if (p x) then x :: filter p xs

   else filter p xs

   Only those elements that satisfy the property p are retained in the result list. The result of **filter** is, possibly, a shrunken list.

5. **inits** - results in a list of lists, in which each sublist contains the initial segments of its argument list.

   inits [ ] = [[ ]]

   inits $[x_1, x_2, \ldots, x_n] = [[\ ], [x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]]$

6. **tails** - results in a list of lists, in which each sublist contains the final segments of the list.

   tails [ ] = [[ ]]

   tails $[x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n], [x_2, \ldots, x_n], \ldots, [x_n], [\ ]]$

7. **cross_product**

   Two functions are defined for performing the cross-product operation, based on their order of evaluation. The basic operation is the same in both the cases, but the evaluation determines the different ordering of elements in the resulting list.

   - **r_cross_product** - specifies that the cross-product is to be performed in a row-major order.

     $$\begin{aligned}\text{r\_cross\_product f } [a_1,a_2,\ldots,a_m]\ [b_1,b_2,\ \ldots,b_n] \\ = [[(\text{f } a_1\ b_1),\ (\text{f } a_1\ b_2),\ldots,\ (\text{f } a_1\ b_n)], \\ [(\text{f } a_2\ b_1),(\text{f } a_2\ b_2),\ldots,\ (\text{f } a_2\ b_n)], \\ \vdots \\ [(\text{f } a_m\ b_1),\ (\text{f } a_m\ b_2),\ldots,\ (\text{f } a_m\ b_n)]]\end{aligned}$$

   - **c_cross_product** - specifies that the cross-product is to be performed in a column-major order.

     $$\begin{aligned}\text{c\_cross\_product f } [a_1,a_2,\ldots,a_m]\ [b_1,b_2,\ \ldots,b_n] \\ = [[(\text{f } a_1\ b_1),\ (\text{f } a_2\ b_1),\ldots,\ (\text{f } a_m\ b_1)], \\ [(\text{f } a_1\ b_2),(\text{f } a_2\ b_2),\ldots,\ (\text{f } a_m\ b_2)], \\ \vdots \\ [(\text{f } a_1\ b_n),\ (\text{f } a_2\ b_n),\ldots,\ (\text{f } a_m\ b_n)]]\end{aligned}$$

   Both these functions apply a binary operator f to pairs of elements taken from each of the two argument lists.

8. **composition** - the composition operation is represented by $\circ$.

   (f $\circ$ g) x = f (g x)

4.2 The Extended Set of Recognised Functions

The functions in the set defined by BMF operate on the *list* type. In the context of the HOPP model, it is sometimes difficult to express certain operations using just these functions. The resulting expressions appear to be rather contrived. This is probably due to two reasons. Firstly, the operations that need to be expressed may not be ones that would typically be performed on list types, but rather on arrays. Secondly, some of these operations may have been devised in the context of parallel programming. The Bird-Meertens Formalism is based on lists and in the context of sequential programming. These difficulties arise when the HOPP model attempts to use the BMF functions to solve array-based problems in the context of parallel programming.

The set of functions in BMF has been extended to incorporate some additional functions which provide *off-the-shelf* recognised functions that can be used effectively for a larger class of problems. These additional functions can be expressed in terms of one or more of the existing recognised functions. This imposes some restrictions on the kind of functions that can be made *recognised functions*. More importantly, it is hoped that the property of amenability to transformations will also be retained by the recognised functions. The possibility of new transformations on the extended set of functions can be investigated and more efficient programs can be derived in the context of parallel programming and the HOPP model. In most of these cases, the newly-coined function has a more cost-effective parallel implementation when compared to the naive implementation corresponding to its composing functions.

The extended set of recognised functions is now described. Each function is first defined using informal ML-style notation, which is followed by its definition in terms of the functions in the basic set of recognised functions. It is to be emphasised that the latter definitions merely serve as illustrations, and there may be other, possibly better ways of expressing the same.

The functions in the extended set are divided into two categories - those which are intuitive and for which their inclusion can be motivated, and some which are not so intuitive, but could be useful in writing programs.

The more important functions are as follows:

1. **map2** - similar to **map**. This function is sometimes referred to as *zipwith*.

   map2 f [ ] [ ] = [ ]

   map2 f (x::xs) (y::ys) = (f x y) :: map2 f xs ys

   This is just an extension of **map** to cater for two argument lists. The argument lists must be of the same size.

2. **zip** - pairs up corresponding elements from two input lists, resulting in a list of pairs.

   zip [ ] [ ] = [ ]

   zip (x::xs) (y::ys) = (x,y) :: zip xs ys

   zip can be defined in terms of **map2**.

   zip xs ys = map2 pairup xs ys

   where,

   pairup x y = (x,y)

3. The iterative functions - this is a set of functions that apply a function f to a list, for a specified number of times. After each stage of the iteration, f is applied to the result of the previous stage. The number of times that f is applied could either be predetermined or conditional. It may be

necessary in some cases, to perform operations on the resulting list before passing it on to the next stage of the iteration, or before the final result is output. This operation may also involve the original list. The function g in the following definitions allows for any transformation on the resulting list. If no such transformation is necessary, then g can be defined to be the identity function. There are three iterative functions which are available as recognised functions.

- **iterate_up** - an iterative function that applies a function f to a list for a specified number of times. After each stage of the iteration, f is applied to the result of the previous stage. A *start* index and a *finish* index are specified, and it iterates until *start* is greater than *finish*. At each iteration, the value of *start* is incremented by one, so that the number of iterations is (*finish - start + 1*).

```
iterate_up finish start f g xs = if (start > finish) then (g xs)
                                  else
                                     iterate_up finish (start + 1) f g
                                              (f finish start (g xs) )
```

- **iterate_down** - similar to **iterate_up**, but the iteration counter is a down-counter. After each iteration, the value of the start index *start* is decremented by one. The iteration stops when *start* is less than *finish*. The number of iterations is (*start - finish + 1*).

```
iterate_down finish start f g xs = if (start < finish) then (g xs)
                                    else
                                       iterate_down finish (start - 1) f g
                                       (f finish start (g xs))
```

- **iterate_cond** - provides for conditional iteration. The function f is applied to the input list. The resulting list is tested to check whether it satisfies some condition defined by the function *cond*. The iteration is stopped if the condition is satisfied, otherwise f is applied to the result. Sometimes it may be necessary to compare the resulting list with the original list in order to check whether the condition is satisfied. Therefore, the original list is a parameter to the function *cond* which operates on two input lists.

```
iterate_cond cond f g xs = let val result = f xs
                           in if (cond result xs) then (g result xs)
                              else
                                 iterate_cond cond f g (g result xs)
                           end
```

The three iterative functions are not themselves implemented in parallel, although they belong to the set of recognised functions. This is because iteration is inherently sequential - step i uses the results of step $(i-1)$. The functions f and g in the definitions of the iterative functions may be implemented in parallel if they are composed of recognised functions. However, if f and g happen to be sequential functions, then the scheme would only select a sequential implementation for the iterative functions. The details of the parallel implementations are given in Chapter 6.

Iteration on a list can be expressed in terms of function **composition**, which belongs to the set of functions in BMF. An illustration is given for the case of **iterate_up** and the definitions for the other iterative functions are similar.

$$\text{iterate\_up finish start f g xs} \equiv \underbrace{f1 \circ f1 \circ \ldots \circ f1}_{n} \text{ (f, g, finish, start, xs)}$$

where,

$$n = \text{finish - start} + 1$$

$$f_1 \text{ (f, g, finish, start, xs)} = \text{let val result} = \text{f finish start (g xs)}$$

in

(f, g, finish, start+1, result)

end

This is a rather informal definition, but it demonstrates that the iterative functions can be expressed in terms of an existing BMF function. The function **iterate_cond** cannot be expressed in the same way as the other two iterative functions since the exact number of iterations is not known. However, a similar idea is applicable.

4. **split** - splits a given list into a specified number of sublists.

$$\text{split } 1\ [x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_n]]$$
$$\text{split } k\ [x_1, x_2, \ldots, x_n] = [[x_1, x_2, \ldots, x_{\lceil \frac{n}{k} \rceil}], \ldots, [x_{(k-1)\lceil \frac{n}{k} \rceil + 1}, \ldots, x_n]]$$

The introduction of **split** as a recognised function was motivated by the need to express divide-and-conquer in applications. The function **split** takes two arguments, an integer $0 < k \leq n$ and a list. The result of applying the function to the list is to split the list into k sublists. A list of size n is rearranged to be a list of lists, with $(k-1)$ sublists of size $\lceil \frac{n}{k} \rceil$ and the last sublist being of length $n - (k-1)\lceil \frac{n}{k} \rceil$.

split can be expressed in terms of the basic set of recognised functions. The resulting composition of functions is quite complex.

```
split k xs = let val part = (len xs)/k
             in
                 (fold append [ ] ∘ map (filter (eq part)) ∘ map tails ∘
                  filter (multiple part) ∘ inits) xs
             end
```

where,

```
append xs ys = xs @ ys
eq n xs = (len xs) = n
multiple n xs = (len xs > 0) and (len xs) mod part = 0
```

The code for **split** in terms of the existing functions is quite contrived. The cost of implementing it as a composition of functions would be much higher than the cost of the version given in Chapter 6. Given that it is a useful function which is applicable in a variety of problems, the incorporation of **split** in the extended set of recognised functions can be justified.

5. $\mathcal{R}^k$ - A **composition** of functions allows for an input list to be piped through several phases of operation. In each phase, some operation (defined by the function(s) in that phase) is performed on the list. The resulting list is passed on to the next phase of the **composition**. This appears to be the only form of control flow in programs that can be expressed using the functions in BMF.

   It is clear, however, that not all the recognised functions operate on a single input list. Functions such as **cross_product** operate on two input lists. It may be the case that two copies of the input list arriving from the previous phase are its arguments. More generally, it may be necessary to perform some operation on each of the copies before they serve as inputs to the recognised function. The combinator $\mathcal{R}^k$ caters for precisely these situations.

The definition is provided for $\mathcal{R}^2$, the case where a recognised function operates on two input lists.

$\mathcal{R}^2$ F f_1 f_2 ... f_n g h xs = F f_1 f_2 ... f_k (g xs) (h xs)

where,

F is a recognised function with 2 input lists

f_1, f_2, ..., f_n are parameters to F

g and h are functions that operate on lists

Depending on which of the input lists is to be passed unchanged to F, g or h or both would correspond to the identity function. In general, g and h could be any complex functions including other recognised functions.

In general, $\mathcal{R}^k$ is defined for a recognised function operating on k input lists. Obviously, a maximum value must be defined for k and this would depend on the maximum number of input lists to a recognised function in the set.

It is difficult to provide a *formal* justification for including $\mathcal{R}^k$ in the extended set. An intuitive justification is based on the nature of control flow that may be required for some of the functions in BMF. **Composition** only provides for one form of control flow in the program, in which a *single* list is passed from one phase to the next. However, the nature of certain recognised functions suggests the need for a second form of control flow which is provided by $\mathcal{R}^k$.

6. **get_neigh** - obtains the left and right neighbours for every element in the argument list. The result is a list of lists, with each sublist containing three elements - the particular list element, and its left and right neighbours. An informal definition for **get_neigh** is as follows.

get_neigh l-∞ r-∞ $[x_1, x_2, x_3, \ldots, x_{n-1}, x_n] =$

$$[[x_1, \text{l-}\infty, x_2], [x_2, x_1, x_3], \ldots, [x_n, x_{n-1}, \text{r-}\infty]]$$

In the definition, ∞ indicates no neighbour in that particular direction, implying an end element of the list. l-∞ and r-∞ represent the left and right neighbours for the first and last elements of the list respectively. The first and last elements do not have neighbours, so the definition allows for some boundary values to be specified. In order to define **get_neigh** in terms of the existing set of functions, a function *sequence* which applies a set of functions to its input list is first defined. *sequence* can be defined in terms of **map2**.

sequence $[f_1, f_2, \ldots, f_n]$ xs = [(f_1 xs), (f_2 xs),...,(f_n xs)]
sequence $[f_1, f_2, \ldots, f_k]$ xs = map2 g $[f_1, f_2, \ldots, f_k]$ [xs, xs,..., k copies]
where, g f_i xs = f_i xs

Then, **get_neigh** can be defined in the following manner.

get_neigh l-∞ r-∞ xs = (stage_3 $\circ$ stage_2 $\circ$ stage_1) xs
where ,

stage_1 xs = sequence[($\mathcal{R}^2$ zip id (scan pick_sec l-∞)),
(reverse $\circ$ $\mathcal{R}^2$ id (zip $\circ$ reverse)
((scan pick_sec r-∞) $\circ$ reverse))] xs

pick_sec x y = y

id xs = xs

stage_2 xs = ($\mathcal{R}^2$ id (zip $\circ$ (fold append [ ])) (select 1)) xs

stage_3 xs = map left_right xs

left_right ((x,y),(w,z)) = [x,y,z]

Essentially, *stage_1* performs one left shift and one right shift on the original list. This is achieved by the use of the **scan** function and it obtains the left and right neighbours for each element. The shifted lists are then *zipped* up with the original list. *stage_2* and *stage_3* just produce the output in the required format. The function **select** is defined below.

The code for **get_neigh** in terms of the basic set of functions is rather contrived. This again, is due to the linear nature of operations on the list structure. **get_neigh** is a function that requires an operation which is not typical of list structures - going backwards rather than forwards.

The function **get_neigh** has been introduced in the list of recognised functions both for expressiveness and efficient implementation.

The less intuitive functions are discussed next. These functions also serve to illustrate that in some of the cases, there is no natural way of expressing them in the BMF-style of programming.

1. **len** - returns the length of a list.

   len [ ] = 0

   len (x::xs) = 1 + len xs

   len can be defined in terms of both **map** and **fold**.

   len xs = fold plus 0 ∘ map subst_1

   where,

   subst_1 x = 1

   plus x y = x + y

2. **select** - selects the jth element in a list.

   select j [ ] = error

   select j xs = if (j <= 0) then error
   else if (j > len xs) then error
   else get j xs

   where,

   get 1 (x::xs) = x

   get j (x::xs) = get ($j - 1$) xs

   Assuming the absence of error conditions, **select** can be defined using a **fold** in the following manner.

   f (k, j,x) x_j = if ($j = k$) then $(k + 1, j, x_j)$
   else ($k + 1, j$, x)

   select i xs = third $\circ$ foldl f (1,i,y) xs

   where,

   third (x,y,z) = z

   select is defined in terms of *foldl* which is not associative. However, a suitable parallel implementation for **select** has been defined. This is possible because the behaviour of **select** is predetermined.

3. **apply_select** - applies a function f to a specified set of elements in the input list. The list of indices of the elements to which f is to be applied must be sorted in *ascending* order.

   apply_select f $[i_j, i_k, \ldots, i_m]$ $[x_1, x_2, \ldots, x_j, x_k, \ldots, x_m, \ldots, x_n]$ =
   $[x_1, x_2, \ldots,$ (f x_j), (f x_k), $\ldots$, (f x_m), $\ldots, x_n]$

apply_select can be defined in terms of **map**.

apply_select f $[i_j, i_k, \ldots, i_n]$ xs = map (f 1 $[i_j, i_k, \ldots, i_n]$) xs

where,

```
f m (ind::inds) [ ] = [ ]
f m (ind::inds) (x::xs) = if (m = ind) then
                              f x :: f (m+1) inds xs
                          else
                              x :: f (m+1) (ind::inds) xs
```

4. **copy** - distributes a particular element of the list to the other elements in the list.

   copy i $[x_1, x_2, \ldots, x_n] = [(x_i, x_1), (x_i, x_2), \ldots, (x_i, x_n)]$

   copy can be expressed in terms of **select** and **map**.

   copy i xs = let val x = select i $[x_1, x_2, \ldots, x_n]$
   in
   map (pairup x) $[x_1, x_2, \ldots, x_n]$
   end

   where,

   pairup x y = (x,y)

   The index of the element which is to be distributed is an argument to the function **copy**. The result of **copy** is a list of pairs with the first of the pair being the distributed element.

5. **reverse** - reverses a list, as the name suggests.

   reverse $[x_1, x_2, x_3, \ldots, x_n] = [x_n, \ldots, x_3, x_2, x_1]$

reverse can be expressed in terms of **fold**.

reverse xs = g_fold rev [ ] xs

where,

rev xs x = [x] @ xs

4.3 Example Programs

In this section, some well-known problems are expressed in the style advocated by the HOPP model, using the functions from the extended set of recognised functions. The same examples will be later used to test the accuracy of the HOPP model, comparing the execution on a parallel machine with the prediction made by the analyser. Details on the parallel implementations and the results can be found in Chapter 7. The examples have been coded in ML.

4.3.1 Matrix Multiplication

The problem of multiplying two matrices $A_{m \times n}$ and $B_{n \times k}$, which results in the matrix $C_{m \times k}$, is considered here.

The problem can be expressed as a composition of two phases. The recognised functions are depicted in bold face and an informal ML-style notation is used.

fun mat_mult times plus A B = **map** (**map** (**s_fold** plus 0)) $\circ$
r_cross_product (**map2** times) A B^T;

fun plus a b = a + b;

fun times a b = a $*$ b;

B^T represents the transpose of B

Each matrix is represented as a list of lists. Each sublist represents a row/column of the matrix. *Phase one* - multiplies the corresponding pairs of elements from pairs of sublists. *Phase two* - performs the addition to obtain the inner products.

The function **s_fold** is used because the size of the emerging result after each stage of the *fold* operation is a constant, and is equal to the size of the base type of the matrix. The transpose of the second matrix is performed sequentially in this case, and therefore increases the costs of all the implementations by the same amount.

4.3.2 Merge Sort

The *merge sort* example is considered next. A list of integers is to be sorted in ascending order. The strategy adopted is to sub-divide the list into a list of lists, with each sublist containing two elements. These two elements are sorted and the sorted sublists are then merged, two at a time, to obtain a fully sorted list. This is similar to the *divide-and-conquer* approach, except that *divide* is not performed recursively. The list is transformed into a list of lists in a single step, using the **split** function. Each sublist then represents the base case which can be solved. For the sake of simplicity, the number of elements in the input list is assumed to be a power of 2.

The problem can be expressed as a sequence of three phases.

fun sort n xs = **g_fold** (merge [ ]) $\circ$ **map** msort $\circ$
split (n/2) xs;

where, n is the length of the list

fun msort [x,y] = if(x $\geq$ y) then [y,x] else [x,y];

fun merge [ ] [ ] = [ ]

merge xs [ ] = xs

merge [ ] ys = ys

merge (x::xs) (y::ys) = if (x $\leq$ y) then
x :: merge xs (y::ys)
else y :: merge (x::xs) ys;

Phase one performs the *divide* step, in which the list is split into sublists, each of size two; *Phase two* sorts each sublist; and *Phase three* performs the merge operation. The function **g_fold** is used since at each step of the **fold** operation, the size of the resulting list is the sum of the sizes of the two lists that were merged. It may be noted that *phase two* may be removed by splitting the list into singleton sublists. It is retained in this example for illustrative purposes only.

4.3.3 Solving Partial Differential Equations

Elliptic partial differential equations are commonly encountered in equilibrium or steady-state problems. One of the best-known elliptic equations is Poisson's equation, represented by Equation 4.1

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{4.1}$$

where, f is the source term. The domain of integration of such a two-dimensional elliptic equation is always an area S bounded by a closed curve C. Only a limited number of special types of elliptic equations have been solved analytically. There are several numerical approximation methods available for solving differential equations, of which those employing finite-differences are more frequently used because of their wider applicability. In these methods, the area of integration S bounded by the closed curve C, is overlayed by a system of rectangular meshes. The meshes are formed by two sets of equally spaced lines, each set parallel to the X-axis and Y-axis respectively. An approximate solution is found at the n points of intersection which are called mesh points. The approximation consists of replacing each derivative of the partial differential equation at a mesh point, $P_{i,j}$, by a finite-difference approximation in terms of the values of u at $P_{i,j}$ and the neighbouring mesh points and boundary points. For each of the n mesh points, an algebraic equation approximating the differential equation is written, giving a set of n algebraic equations in n unknowns. Sets of linear algebraic equations can be solved by *direct* or *iterative* methods. Direct methods are based on Gaussian elimination with pivoting or triangular decomposition of the matrix of coefficients. Descriptions and C programs for these methods can be found in [P+88]. The method considered in this section is an iterative method due to Jacobi.

The area S is assumed to be rectangular, with sides of length ph and qh, and to have known values b on the perimeter of S. If S is sub-divided into a network of squares of side h, then the mesh points are defined by:

$$\begin{aligned} x &= ih,\ (i = 0, 1, \ldots, p) \\ y &= jh,\ (j = 0, 1, \ldots, q). \end{aligned} \tag{4.2}$$

Approximating the equation by the five-point difference scheme [Smi65, Ame69] yields:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} - h^2 f_{i,j} = 0. \tag{4.3}$$

$u_{i,j}$ is then given by:

$$u_{i,j} = \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j}). \tag{4.4}$$

If the nth iterative value of $u_{i,j}$ is denoted by $u^n_{i,j}$, then an iterative procedure for solving Equation 4.4 is defined by:

$$u^{(n+1)}_{i,j} = \begin{cases} b_{i,j}; & \text{if } (i = 0, p),\ (j = 0, q) \\ \frac{1}{4}(u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1} - h^2 f_{i,j}); & \text{otherwise} \end{cases} \tag{4.5}$$

Successive iteration causes the approximate solution to converge to the exact solution.

The program for implementing the Jacobi method is given below. The region S is assumed to be a square, with $p = q =$ JMAX, where JMAX is some predefined constant, $h = 1$ and $b_{i,j} = 0$; $(i = 0, p;\ j = 0, q)$.

```
fun gen_zeros 0 = [ ]
  | gen_zeros n = 0.0 :: gen_zeros (n-1);

val inf1 = gen_zeros JMAX;

val inf2 = gen_zeros JMAX;


fun get_rest [ ] [ ] [ ] = [ ]
  | get_rest [ ] ts [ ] = [ ]
  | get_rest [ ] [ ] bs = [ ]
  | get_rest es ts [ ] = [ ]
  | get_rest es [ ] bs = [ ]
  | get_rest [ ] ts bs = [ ]
  | get_rest (es::ess) (t::ts) (b::bs) =
           (es @ [t,b]) :: get_rest ess ts bs;


fun rearrange [xs, ys, zs] =
           let val res = get_neigh 0.0 0.0 xs
           in get_rest res ys zs
           end;
fun less_than (eps:real) (value:real) = (value < eps);
fun maxi (x:real) y = if (x > y) then x else y;
fun diff (x:real) (y:real) = abs (x-y);
fun update f [u0,u1,u2,u3,u4] = 0.25 * (u4 + u3 + u2 + u1) - f/4.0;
fun id x y = x;


fun test (eps:real) xs ys = ((less_than eps) o (s_fold maxi 0.0) o
                             (map (s_fold maxi 0.0)) o
                             (map2 (map2 diff) xs) ) ys;
```

```
fun jac fss uss = ((map2 (map2 update) fss) ∘
                   (map rearrange) ∘
                   (get_neigh inf1 inf2) ) uss;


fun pde (eps:real) fss uss =
                    iterate_cond (test eps) (jac fss) id uss;
```

The function, *pde*, is the main function that iteratively applies the function, *jac*, to the input matrix *uss*. The source term is represented by *fss*. Both *uss* and *fss* are represented as lists of lists. The function *jac* obtains the neighbouring values for each mesh point and updates its value according to Equation 4.5. The function *test* checks whether convergence has been achieved. After each iteration, the value of a mesh point is subtracted from the corresponding value at the previous iteration. If the absolute value of the maximum of such differences is smaller than some predefined constant, *eps*, then iteration is stopped and the result is the required solution.

4.4 Conclusion

This chapter introduced the set of recognised functions and illustrated their use with three example programs. The programs for matrix multiplication and merge sort were more natural to express in the BMF-style, compared to the program for jacobi iteration. In all the three cases the difficulty of natural expression was caused by the use of the list data structure. The programs are best-suited to array data structures, but the recognised functions only work on lists. A *theory of arrays* [Mil93] with data-parallel operations defined on it would probably result in a more natural data-parallel model of programming.