Part 5
Concurrency issues

Functions written in a functional program have the advantage that they can be evaluated in any order while the outcome of the computation remains the same. This property makes functional programs *conceptually* very suited for *concurrent* (i.e. *interleaved* or *parallel*) evaluation.

However, the exploitation of concurrency in functional programs is still in its infancy. Automatic creation of parallelism such that a significant speed-up is obtained is not easy to realize. It is also not clear yet what kind of primitives should be given to a programmer for a high-level specification of the required process structure and behaviour. So this part of the book is of a rather experimental nature, due to the many open research questions and the rapid developments in this interesting and promising area. Nevertheless, we hope that we can give a good impression of the conceptual and practical possibilities of the concurrent evaluation of functional programs.

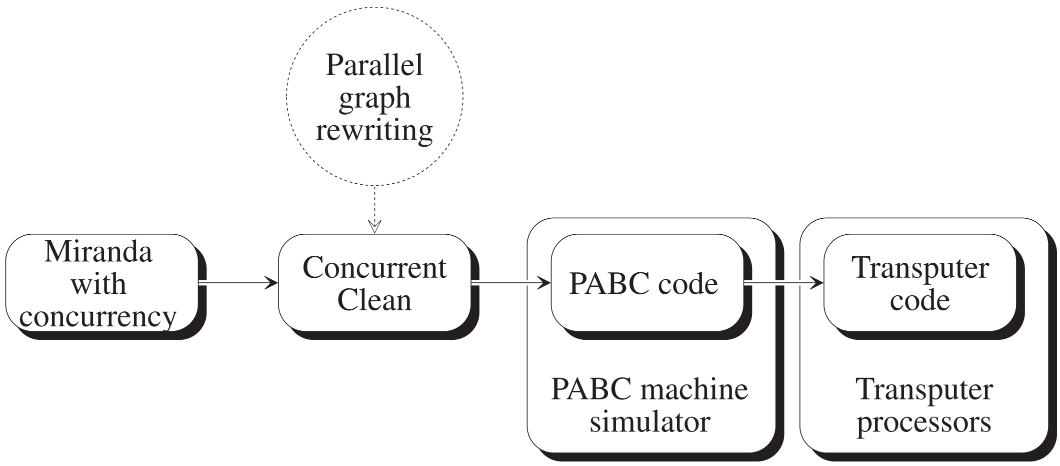


Figure P5.1 Implementing concurrent functional programming.

For simplicity we shall assume that concurrency is specified *explicitly* by the programmer via special *process annotations*. The presented annotations are especially designed to indicate the desired evaluation order for a multi-processor architecture where each processor is equipped with its own local memory. So functions (work) and constructors (data) have to be shipped from one local memory to another. Hence graphs have to be copied from one processor to another. This is modelled by the concept of *lazy copying* which extends the standard graph rewriting model. The implementation of concurrent functional programs on parallel architectures will be discussed via the same translation steps as were used for the sequential implementation (Figure P5.1).

Chapter 13
Basic language concepts

In a *concurrent* program the program is split up into parts (tasks) that are executed *concurrently*, i.e. *interleaved* or in *parallel* with each other. Each task is evaluated by a separate process. *Parallel* processes run on *different* processors. *Interleaved* processes run on the *same* processor. The result of a functional program is known to be independent of the chosen evaluation order (although one has to be a bit careful not to change the termination behaviour of a program). So functional programs seem to be very suited for concurrent evaluation. Conceptually it is indeed possible to evaluate functions interleaved or even in parallel with each other. By using mutual recursion arbitrary dependencies between these functions can be specified, thus creating a way to define *arbitrary networks of processes* (reducers). C*ommunication* and *synchronization* between the processes are realized *automatically* using the *lazy evaluation* principle. Communication takes place when a process demands a value that is being calculated by another process. No additional communication primitives are needed.

When concurrent evaluation is used to gain efficiency one actually would like to have an analyser that *automatically* marks expressions that can safely be executed in parallel. A strictness analyser can be used for this purpose. But one often also needs to know whether parallel evaluation of expressions is *worthwhile*. With the creation of each task a certain amount of *overhead* is involved, depending on the number of processes created and the amount of communication that takes place between them. One can only *gain* efficiency when a *sufficiently large amount of work* is assigned to a process which involves *limited inter-process communication* for the exchange of information. The amount of work performed by a process and the amount of inter-process communication that is needed to do the work is of course undecidable. The actual overhead will also depend on the concrete machine architecture the program is running on. How to split up work efficiently is therefore very problem- and machine-dependent and often difficult to solve, even for a human being. To make things easier, we assume from now on that the programmer has to define the concurrent behaviour of the functional program *explicitly*, either in order to achieve a certain desired concurrent structure or to achieve a faster program. Furthermore, we assume that concurrent functional programs at least have to run conveniently on a widely available class of parallel machine architectures: multiple instruction–multiple data machines with a distributed memory architecture. Such a machine can consist of hundreds or even thousands of more or less conventional processors that are connected via a communication network.

In spite of the conceptual possibilities, *concurrent functional programming* is still in its infancy. At the moment, none of the commercially available functional languages supports concurrent programming. However, in several experimental languages concurrency primitives have been proposed in the form of annotations or special functions (Kluge, 1983; Goguen *et al.*, 1986; Hudak and Smith, 1986; Burton, 1987; Glauert *et al.*, 1987; Vree and Hartel, 1988; Van Eekelen *et al.*, 1991; Darlington *et al.*, 1991). With these primitives the default evaluation order of an ordinary functional program can be changed such that a concurrent functional program is obtained. However, there is not yet a common view on which kinds of primitives are handy or definitely needed for concurrent functional programming.

This chapter therefore does not reflect *the* way to achieve concurrency, but it presents *one* of the methods to define concurrency. In the presented method the concurrent behaviour of a program is defined by means of special high-level concurrency *annotations* with an associated *type system*. By combining the use of higher order functions with the type system it is possible to specify a very large class of process structures in an elegant manner.

This chapter first gives a brief introduction on concurrency in general and concurrency in the context of functional languages (Section 13.1). Then the annotations and the type system are introduced in a Miranda-like syntax (Section 13.2). Section 13.3 shows some more elaborate examples of concurrent functional programs. Section 13.4 discusses the advantages and the disadvantages of the functional concurrency primitives that are used in this chapter.

13.1 Concurrency and functional programming

A **concurrent program** is a program in which parts of the program (called **processes**) are running **concurrently**, i.e. *interleaved* or in *parallel* with each other. An algorithm that is to be executed is called a **task**. Each task involves a certain amount of work. In a concurrent program the task to be performed is split up into *subtasks***.** Each subtask is assigned to a process. **Parallel processes** are processes that perform their task at the *same* time. **Interleaved processes** perform their task *merged* in some unknown *sequential* order on a time-sharing basis.

13.1.1 Why concurrent programming?

Concurrent programming is mainly used for the following reasons:

(1) To *obtain efficiency*: concurrent programs are used to decrease the time needed to accomplish the complete program task by assigning subtasks to *parallel* executing processes. Of course, one needs a multi-processor machine architecture for the execution of these processes to actually gain efficiency.
(2) To *structure software*: some programs are logically composed of components that perform subtasks concurrently (take for instance the components of an operating system). These subtasks can run interleaved on a single processor machine or in parallel on a multi-processor machine.
(3) To *program parallel hardware* such as a system that consists of autonomous components that interact with each other, e.g. a multi-processor architecture or a system configuration consisting of a processor, a printer and a tape unit.

The design and implementation of concurrent programs are a difficult job for a human being. One not only has to decide *which* subtasks are to be assigned to processes, but one also has to decide *how* and *when* the corresponding processes have to communicate with each other. The interaction between the processes must be programmed in such a way that at any moment any communication with another process is handled in the intended manner. In general, special language facilities or system calls are available that define the creation of processes with their tasks. Also, primitives are needed to enable the control over communication and synchronization between the processes.

When concurrent programming is used to *structure software* or to *program parallel hardware* the design of the concurrent program is substantially determined by the subtasks that need to be performed by the separate components in the system. Such a program can become complex when there are many tasks created that heavily depend on each other. For a human being it is very difficult to think of all the possibilities that can arise in such a complex concurrent environment.

Gaining speed with concurrent programming

When concurrent programming is used to obtain a more *efficient* program the subtasks performed by a program must be assigned to parallel executing processes in such a way that they indeed increase the efficiency. But with each process created and each communication that takes place some overhead is involved. An increase in efficiency can actually only be obtained when the subtask that has been assigned to a parallel process represents a sufficiently large amount of work. The additional overhead needed for process creation and inter-process communication should be low. The problem is now how to tune the number of processes and the kind and size of the tasks such that an optimal gain in efficiency is obtained on a certain concrete architecture.

Fine-grain versus coarse-grain parallelism

One can split up a program into *tiny* subtasks in which a relatively small amount of work is performed. In general, a lot of this **fine-grain parallelism** can be found in a program. It is even often possible to find many of these fine grains automatically. For example, the strict arguments of a function can safely be evaluated in parallel. Efficiency gain can only be achieved with tiny tasks if there is almost no overhead penalty for the creation of these tasks on the underlying architecture. However, in reality it may well be the case that the additional overhead needed to create these tiny subtasks becomes larger than the speed gained by evaluating them in parallel. The reason for this is that the overhead costs are generally dependent on the *number of tasks* and the *amount of communication* between them, but not on the amount of work performed by a task.

An efficiency gain can be obtained when it is possible to split up a program into subtasks that consist of a relatively large amount of work with only limited communication between the tasks. But, in general, only a *limited* amount of this **coarse-grain parallelism** can be found. Besides the knowledge that it is safe to fork off a task one also has to take the complexity of a task and the amount of communication between tasks into account. These last two entities are in general undecidable. Furthermore, it is even for a human being often rather difficult to find a suitable coarse-grain partitioning of a program. It generally means that one has to redesign the algorithm. When not enough coarse-grain parallelism can be found or created the capacity of a parallel machine cannot be used in an optimal manner.

Importance of the underlying concrete parallel architecture

The optimal tuning of a program will not only depend on the problem one has to solve but will also depend heavily on the concrete architecture of the parallel machine that is actually being used. There are single instruction–multiple data (SIMD) machines and multiple instruction–multiple data (MIMD) machines. In an **SIMD** machine a very large number (say, thousands) of tiny processors all execute the *same* instruction at the *same time*, albeit on *different data*. They are especially suited to performing the same type of calculation on a large data structure (array or vector processing). In an **MIMD** machine there are fewer (say, hundreds) 'ordinary' processors working independently. These architectures are generally more suited for executing concurrent functional programs. There are MIMD architectures with *shared memory* and with *distributed memory*.

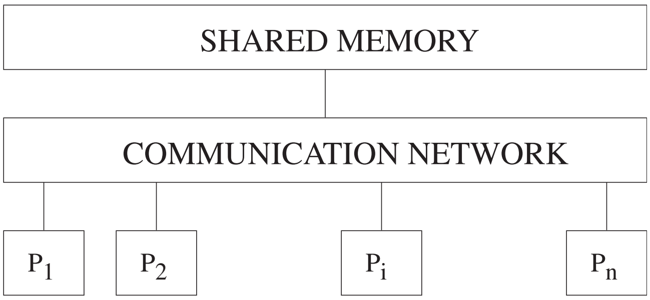


Figure 13.1 Parallel architecture with shared memory.

In a **shared memory architecture** (Figure 13.1) all processes share the same memory via a communication device such as a bus or a special network switch. Shared memory architectures have the advantage that all processes can communicate with each other via global data. This makes creation of processes and inter-process communication cheap. The status of each processor can also be inspected relatively easily by the operating system of the machine. As a consequence, the distribution of work between the processors can also be controlled relatively easily. A shared memory architecture is therefore best suited for automatic creation of parallelism. A disadvantage of such an architecture is that there are two bottlenecks, one formed by the communication network and one formed by the shared memory: generally only one memory access is possible at the same time. So only a certain limited number of processors (say, tens of processors with current technology) can effectively contribute in a shared memory configuration.

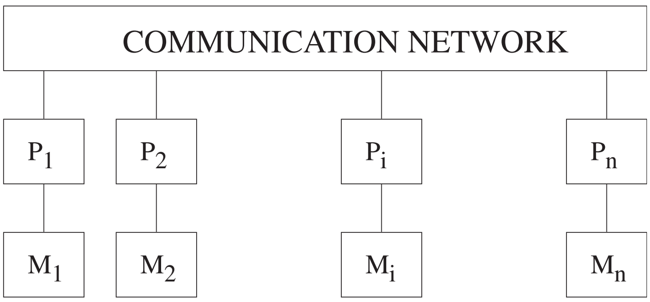


Figure 13.2 Parallel architecture with distributed memory.

In a **distributed memory architecture** (Figure 13.2) each processor has its own local memory. Processors are (sparsely) connected via some special communication network. In a distributed architecture generally not all processors are directly connected to each other. A fully connected network would be too complex and too expensive. An important property of distributed architectures is that for each processor it is more efficient to access objects located in its own local memory than to use the communication medium to access remote objects.

Distributed memory architectures have the disadvantage that all processors must communicate with each other via a message-passing mechanism that uses the communication network. Information has to be copied from one processor memory to another. This makes process creation and inter-process communication relatively expensive. Generally, it is also very expensive to get an overall overview of the status of such a machine. As a consequence, the distribution of work between the processors is harder to control. Furthermore, garbage collection is difficult to realize. However, the advantage of a distributed architecture is that the network is only used for global memory access but not for local memory access. So the bottleneck formed by the communication network is not so serious when local memory access happens frequently compared with global memory access. With a suitable network topology generally hundreds of processors can be connected to each other in such a way that they effectively work with each other.

Assumptions

When concurrency is used to speed up execution the actual speed-up one can obtain is highly dependent on the algorithm being used, the number and the kind of subtasks that are created and the actual multiprocessor architecture the program is running on. How to split up work efficiently is an undecidable problem. It is often even difficult for a human being to find an acceptable solution. Even when concurrency is used to structure software it is often very hard for a human being to come up with a good design and specification for the desired structure.

So an automatic partitioning of a program in concurrently executable parts seems not to be realistic yet. Therefore, we assume that the programmer *explicitly* has to define the concurrent behaviour of the functional program, either to achieve a certain desired concurrent structure (which cannot be achieved automatically anyway) or to achieve a faster program (which can only be achieved automatically under certain conditions). For this purpose one needs special language constructs for expressing the desired concurrent behaviour. One basic approach to express concurrence in imperative languages can be found in Hoare (1978). Furthermore, we assume that concurrent functional programs at least have to run conveniently on MIMD machines with a distributed memory architecture. They are widely available and generally contain more processors than shared memory architectures.

13.1.2 Why concurrent *functional* programming?

Imperative programming languages have the disadvantage that one cannot always assign an *arbitrary* subtask (such as a procedure call) to a process. A procedure call can have side-effects via access to global variables (see Chapter 1). When such a procedure is executed concurrently, the correctness of the program is no longer guaranteed. Furthermore, inter-process communication has to be defined *explicitly*. All possible communication situations have to be handled. Programs tend to become very complex.

Advantages of concurrent functional programming

In a concurrent functional program a task assigned to a process consists of the evaluation of a function. *Any* function (redex) can be assigned to a process. Since there are no side-effects, the outcome of the computation, the normal form, is independent of the chosen evaluation order. Interleaved as well as parallel evaluation of redexes is allowed. Communication between the processes takes place *implicitly*, simply when one process (function) needs the result calculated by another. No additional primitives for process communication are needed. Reasoning about the concurrent algorithm is for most properties similar to reasoning about any other functional program.

The fact that the evaluation order cannot influence the outcome of a computation also gives additional *flexibility* and *reliability* for the evaluation of functional programs on parallel architectures. When a processor becomes defective or when it is overloaded with work it is in principle possible to change the evaluation order and the number of tasks created in optimal response to the actual run-time situation.

Besides the advantages mentioned above, the programmer of concurrent functional programs has the full power of a functional language at his or her disposal. This means that the general advantages of functional programming are also applicable in a concurrent context.

Disadvantages of concurrent functional programming

Concurrent functional programming means that the programmer explicitly defines how the concurrent evaluation of the program must take place. As a consequence, a program in a functional language can no longer be regarded as an executable specification. How expressions are being evaluated is now of importance.

Actually, already for ordinary lazy functional programs there are situations in which a programmer cannot be totally unaware of the evaluation order of his or her program. For instance, patterns specified on a left-hand side of a function definition force evaluation. Whether or not a function can be called with an argument representing an infinite computation will depend on how the function is defined. Furthermore, most functional languages have facilities to influence the default evaluation order: LML (Augustsson, 1984), Miranda (Turner, 1985), Haskell (Hudak *et al.*, 1992) and Clean (Brus *et al.*, 1987). When these facilities are used one also has to be aware of how and when certain expressions are evaluated. So although functional languages focus on a declarative style of programming there are some points where this goal is not totally met.

In concurrent functional programming the programmer has to be even more aware of the evaluation order. This actually means that the evaluation order becomes part of the specification. However, when a functional program is compiled many transformations are needed to map the high-level specification efficiently onto the low-level instruction set of the machine. As a consequence, it is very hard to predict in which order arbitrary expressions are being evaluated. So the tools given to a programmer should make it possible to specify the reduction order in such a way that the desired concurrent behaviour becomes clear without detailed knowledge of a particular implementation.

13.2 Annotations for concurrency

13.2.1 Creating parallel processes

Suppose that one would like to increase the execution speed of the following function by introducing parallelism into the computation.

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  fib (n – 1) + fib (n – 2),      if n > 2
```

Since both arguments of the addition have to be calculated before the addition can take place one could try to optimize the performance by calculating the two recursive calls of fib in parallel, each on a different processor. This is a typical example of **divide-and-conquer** parallelism. We assume that one can create a parallel process for the reduction of a function application by prefixing the application with a special annotation: {Par}. A process created with a **{Par}** has as task to evaluate the annotated function in *parallel* to *normal form*. With this annotation the desired parallel divide-and-conquer variant of fib is specified as follows:

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  {Par} fib (n – 1) + {Par} fib (n – 2),      if n > 2
```

The use of {Par} in such a recursive function definition creates a new process for each annotated function application in each call of the function. In this way a tree of processes is created. The bottom of the tree consists of processes that execute the non-recursive alternative of the function definition (Figure 13.3).

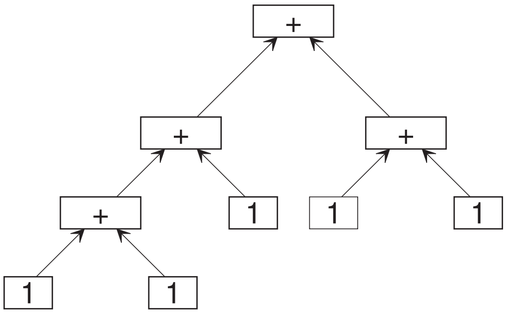


Figure 13.3 Snapshot of a tree of processes computing fib 5; the arrows indicate the direction of the flow of information between the processes.

It will often not be worthwhile to evaluate fib n in parallel in such a way. To turn the *fine-grain* parallelism into *coarse-grain* parallelism, a threshold is introduced that ensures that the processes at the bottom of the tree have a substantial amount of work to do (Figure 15.4).

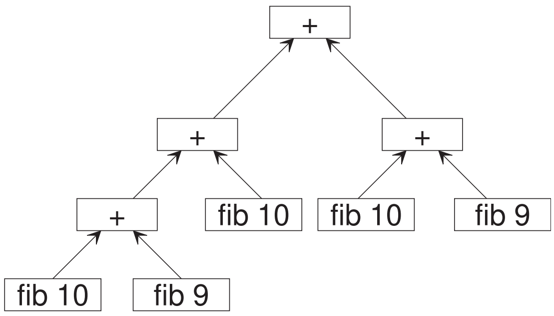


Figure 13.4 Tree of processes with threshold computing fib 13.

Divide-and-conquer Fibonacci with threshold:

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  {Par} fib (n – 1) + {Par} fib (n – 2),      if n > threshold
       =  fib (n – 1) + fib (n – 2),                  if n > 2

threshold:: num
threshold   =  10
```

The {Par} annotation can be used in the body (the right-hand side) of any function definition. In order to keep the semantic description of the {Par} annotation as simple as possible, {Par} is only defined for an argument that is a function application of the form f a_1 a_2 ... a_n. The {Par} annotation is not defined on *ZF*-expressions and the like. This is of course not a fundamental restriction.

When a function with a {Par} annotation in its body is evaluated by a process (the **parent** process) the following action is taken. For each **Par-annotated function application** {Par} f a_1 a_2 ... a_n specified in the body of the function a new process is created (a **child** process). This child process *preferably* runs in *parallel* on a *different* processor, with as task the evaluation of f a_1 a_2 ... a_n to *normal form*. If the child process for one reason or another cannot be created on another processor it is allowed to create it on the same processor as the parent process. Parallel processes are usually created to perform a substantial task. Therefore, a task performs the reduction of the indicated expression to *normal form* and not just to *head normal form*. When the child processes have been created the parent process continues as usual with the regular evaluation of the function body. Consequently, the creation of parallel processes will not influence the termination behaviour of a program.

13.2.2 Creating interleaved processes

In the solutions presented so far processes have not been used in an optimal way. The parent process did not do much useful work because it had to wait for the results of its child processes.

Consider again the fib example. A better processor utilization may be achieved when one of the two arguments is reduced by the parent process.

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  fib (n – 1) + {Par} fib (n – 2),     if n > threshold
       =  fib (n – 1) + fib (n – 2),           if n > 2
```

The intention in the fib example above is that the first argument is reduced on the parent processor in parallel with the evaluation of the second argument. As specified, the first argument *will* be calculated by the parent process and the second one by another parallel process. However, the solution assumes that the parent process directly continues with the evaluation of the first argument. In reality, this may not be the case. If the parent process happens to start with the calculation of the second argument first, it waits until the child process has communicated the result, after which the parent process can evaluate the first argument. So although the specification is fulfilled, the desired parallel effect may not be obtained.

The problem illustrated in the example arose because the programmer had made some (possibly wrong) assumptions on the precise evaluation order of the program (in the case above this concerned in particular the order of evaluation of the strict arguments of δ-rules). It is generally very dangerous to make any assumptions on the evaluation order. Therefore, we have chosen to be very *explicit* about the evaluation order when processes are created. A new annotation is introduced that creates an *interleaved* executing child process on the same processor as the parent process: the *{Self} annotation* (Figure 13.5).

Divide-and-conquer Fibonacci with parent processor (not the parent process) participating in the work:

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  {Self} fib (n – 1) + {Par} fib (n – 2),     if n > threshold
       =  fib (n – 1) + fib (n – 2),                  if n > 2
```

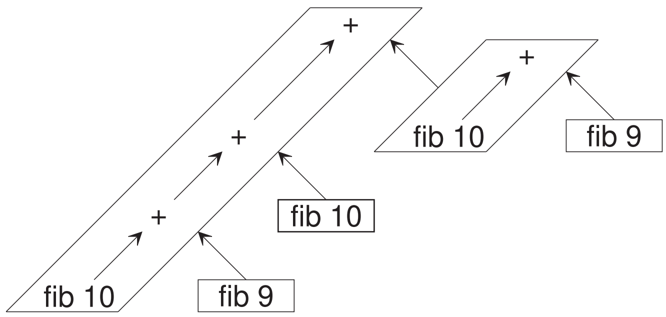


Figure 13.5 Tree of processes for fib; the left branch is computed on the same processor by different interleaved running processes.

The {Self} annotation can be used like a {Par}: in the body (the right-hand side) of any function definition. When a function with {Self} annotations in its body is evaluated by the parent process the following action is performed. For each **self-annotated function application** {Self} f a_1 a_2 ... a_n a new *child* process is created as a subprocess. This child process runs *interleaved* on the *same* processor as the parent process with as task the evaluation of f a_1 a_2 ... a_n to *normal form*.

In many cases a better utilization of the machine capacity can be achieved when a subprocess is created on the same processor as the parent process. Furthermore, the {Self} annotation is in particular handy to create **channel processes**. When several processes demand information from a particular process, it is useful to create a (channel) subprocess for each demanding process to serve the communication demands.

13.2.3 Communication and lazy evaluation

In the sequential case, when a value is needed, it is evaluated by the process itself. In the concurrent case such a value can be evaluated by another process, possibly on another processor. Clearly, the only sensible thing to do for a demanding process is to *wait* until the needed value becomes available. However, for efficiency it is important that a demanding process can continue as soon as possible. Such a process should not wait until the *whole* normal form has been calculated. As soon as part of it is known it can be communicated. This prompt communication of information can be compared with the way the result of an ordinary program is printed. So communication can take place as soon as a head normal form is reached. That part of the result which is still under evaluation will be communicated later when it becomes needed as well. So *communication* is defined *implicitly* by making use of the lazy evaluation order. This way of communication can be used to create *information streams* between processes.

A communication stream between processes (Figure 13.6). The function generator is also used later on in several other examples in this chapter.

```
generator:: num -> [num]
generator n = [n..100]

map (* 2) ({Par} generator 3)
```

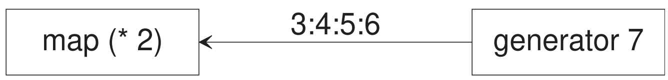


Figure 13.6 Simple pipeline between the processes map and generator; in the snapshot the values 3:4:5:6 are being communicated.

When the parent process needs a value of the child process and the child process has produced a head normal form, the requested information is communicated to the parent process. That part which is still under evaluation will be communicated later when there is a demand for it. In this way the generator and the map process are effectively operating in a (very small) pipeline.

13.2.4 Expressive power of concurrent functional programs

An advantage of functional languages is that it is relatively easy to define general tools for the creation of parallelism by using annotations like {Par} in combination with the ordinary expressive power of higher order functions in these languages.

Divide-and-conquer parallelism can be expressed in a general way using higher order functions:

```
divconq:: (* -> **) -> * -> (* -> bool) -> (** -> ** -> **) -> (* -> (*,*)) -> **
divconq f arg threshold conquer divide
          =  f arg,                                            if threshold arg
          =  conquer ({Self} divconq f left threshold conquer divide)
                      ({Par} divconq f right threshold conquer divide),otherwise
             where    (left, right)  =  divide arg

pfib:: num -> num
pfib n    =  divconq fib n threshold (+) divide
             where    threshold n    =  n <= 10
                      divide n       =  (n – 1, n – 2)
```

Function composition can be used to create pipelines of processes.

Static pipeline of processes (Figure 13.7):

```
stat_pipe:: * -> (* -> **) -> (** -> ***) -> (*** -> ****) -> ****
stat_pipe i f1 f2 f3 = f3 ({Par} f2 ({Par} f1 ({Par} i)))

stat_pipe (generator 3) (map fib) (map fac) (map (* 2))
```

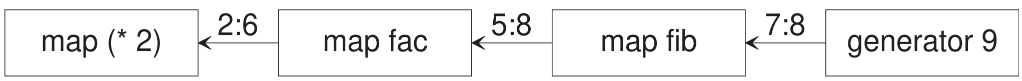


Figure 13.7 Snapshot of a static pipeline of four processes.

With higher order functions general skeletons can be defined to create frequently occurring process structures (this is essentially different from the process skeletons in Darlington *et al.* (1991), which are inherently predefined). Often parallel variants of the basic building blocks given in Chapter 2 can be used.

A general pipeline defined with the {Self} annotation:

```
parfoldr:: (* -> ** -> **) -> ** -> [*] -> **
parfoldr f i [ ]         =  i
parfoldr f i (x : xs)    =  {Par} f x in      where   in  =  {Self} parfoldr f i xs

parfoldr map (generator 3) [(* 2), fac, fib]
```

A parallel version of map implementing vector-like processing: it creates a parallel process for each element in a given list.

```
parmap:: (* -> **) -> [*] -> [**]
parmap f (x : xs) =  {Par} f x : {Self} parmap f xs
parmap f [ ]      =  [ ]
```

then

```
parmap (twice fac) [0, 1, 2, 3]
→ {Par} twice fac 0 : {Self} parmap (twice fac) [1,2,3]
→ {Par} twice fac 0 : {Par} twice fac 1 : {Self} parmap (twice fac) [2,3]
...   →   [1, 1, 2, 720]
```

A process can create one or more subprocesses with the {Self} construct. These subprocesses (running interleaved on the same processor) can be used to serve communication channels with other processes. Each communication link of a process has to be served by a separate subprocess that reduces the demanded information to normal form. A process with its subprocesses in a functional language acts more or less like a process with its channels in a message-passing language like CSP (Hoare, 1978). Serving subprocesses is like sending information over a channel to any process requesting that information.

In the following parallel version of the quicksort algorithm (see also Chapter 2) two child processes are created when the list to be sorted contains more than threshold elements (this is checked by the predicate too_few_elements that avoids walking down the complete list). Each child process sorts a sublist. The parent process will supply the appropriate sublist to each of its child processes. The parent process can perform both these tasks 'simultaneously' with the help of two subprocesses running interleaved with each other.

```
sorter:: [num] -> [num]
sorter list  =  quick_sort list,       if too_few_elements list threshold
             =  par_quick_sort list,  otherwise

threshold:: num
threshold  =  7

quick_sort:: [num] -> [num]
quick_sort [ ]       =  [ ]
quick_sort (x : xs)  =  quick_sort [b | b <- xs ; b <= x]
                        ++ [x] ++ quick_sort [b | b <- xs ; b > x]

par_quick_sort:: [num] -> [num]
par_quick_sort (x : xs)    =  {Par} sorter ({Self} smalleq x xs)
                              ++ [x] ++ {Par} sorter ({Self} larger x xs)
                              where   smalleq x xs  =  [b | b <- xs ; b <= x]
                                      larger x xs   =  [b | b <- xs ; b > x]
```

```
too_few_elements:: [num] -> num -> bool
too_few_elements  [ ]       n  =  True
too_few_elements  xs        0  =  False
too_few_elements  (x : xs)  n  =  too_few_elements xs (n – 1)

sorter [6,3,1,4,2,7,3,12,5,1,4,97,3,2,17,6,93,114]
```

13.2.5 Specifying process types

In the examples shown so far, processes only *locally* appeared in the function body. Complex process topologies can be specified more elegantly if processes can be passed as arguments to functions or returned as function results. The consistent use of processes can be checked by the compiler. Furthermore, for code generation it can be important to know that an expression will be (or has been) evaluated by a process. So a special type attribute {proc} is introduced. This type attribute indicates that an expression is known to be in process normal form.

When an expression is in **process normal form** (**PNF**) it will, *when its evaluation is demanded*, either be *in* normal form or it will be *reduced to* normal form by one or more *processes*. The PNF property is in general undecidable. However, it is possible to introduce a type system to achieve a decidable approximation of the PNF property (*known to be* in PNF). Clearly, an expression annotated with a {Par} annotation or a {Self} annotation is known to be in PNF since a process is created to reduce it to normal form. There are more cases in which expressions are known to be in PNF. A complete survey of these cases is given below. In the following we shall use *in PNF* instead of *known to be in PNF* when there can be no confusion.

An expression has type **{proc} T** when it is of type T and in PNF. This type can be used in the type definition of a function. An expression is said to have a **process type** when its type has the type attribute {proc}.

A tool to create a dynamic pipeline of processes of arbitrary length:

```
pipeline:: * -> [* -> *] -> {proc} *
pipeline gen filters  =  npipe ({Par} gen) filters

npipe:: {proc} * -> [* -> *] -> {proc} *
npipe in [ ]        =  in
npipe in (x : xs)   =  npipe ({Par} x in) xs

pipeline (generator 3) [map fib, map fac, map (* 2)]
```

In the function npipe the newly created parallel process will be evaluating a function x applied on an argument evaluated by another process. Due to the recursive call a pipeline of processes is created.

A type inferencer can derive that an argument of a function has a process type. However, in many cases the programmer wants to be more restrictive, indicating process types *explicitly*. A type *checker* can then check the consistency of the type attributes and assign process types to subexpressions of function definitions accordingly. For reasons of simplicity it is assumed that these actions are performed after the normal type inferencing/checking.

The following expressions are known to be in PNF and therefore a process type can be assigned to them:

- expressions of the form {Par} f e_1 … e_n or {Self} f e_1 … e_n for $n \geq 0$;
- an argument of a function if on the corresponding position in the type definition a process type is specified and a result of a function if on the corresponding position in the type definition a process type is specified;
- expressions of the form C a_1 a_2 ... a_n, where C is a constructor of which all the arguments a_i have a process type (*composition*);
- arguments a_i of an expression that has a process type and that is of the form C a_1 a_2 ... a_n, where C is a constructor (*decomposition*);
- expressions statically known to be in normal form, e.g. expressions not containing any function applications.

The decomposition case reflects the property that when a process returns a complex value the information that a process is evaluating this value should not be lost when an object contained in this complex value is selected. This property is called the **decomposition property**.

Assume that g is of type [num]; then using the decomposition property x is of type {proc} num in (for more practical examples see the next section):

```
x where (x : xs) = {Par} g
```

With similar reasoning the following type specification is accepted:

```
phd:: {proc} [num] -> {proc} num
phd (x : xs)     =  x
```

With the assigned process types the standard type substitutions and unifications are performed, with the following two exceptions:

- Where a process type is specified but a process type cannot be assigned, a process type error will occur.

This definition of pipeline will be rejected since on the right-hand side in the application of npipe for gen no process type is assigned, while in the type definition of npipe a process type is specified:

```
pipeline:: * -> [* -> *] -> {proc} *
pipeline gen filters  =  npipe gen filters

npipe:: {proc} * -> [* -> *] -> {proc} *
```

- Where a non-process type is specified but a process type is assigned no error will occur. In that case the specified type is used in substitutions and for unification (**deprocessing**). This deprocessing, however, does not exclude the possibility that process types are substituted for type variables.

  With the following definition:

```
f:: [num] -> num
f (x : xs)    =  x
```

  f ({Par} generator 3) has type num due to deprocessing. But, with the more general polymorphic definition:

```
f:: [*] -> *
f (x : xs)    =  x
```

  f ({Par} generator 3) has type {proc} num by decomposition and substitution.

The type system is such that in well-typed programs it is guaranteed that expressions that have a process type are in PNF.

13.3 Examples of concurrent functional programs

In this section two more elaborate examples of concurrent functional programs are given: the sieve of Eratosthenes computing prime numbers and Warshall's algorithm solving the shortest path problem. The purpose of the examples is to show that more complex process topologies can be expressed elegantly with help of the presented annotations and type attributes. It is not the intention to show how ultimate speed-ups can be achieved for the problems in question.

13.3.1 Sieve of Eratosthenes

The sieve of Eratosthenes (see also the sequential Clean version in Chapter 8) is a classical example generating all prime numbers. In the parallel version a pipeline of processes is created. There is a process for each sieve. Those sieves hold the prime numbers in ascending order, one in each sieve. Each sieve accepts a stream of numbers as its input. Those numbers are not divisible by any of the foregoing primes in the pipeline. If an incoming number is not divisible by the local prime as well, it is sent to the next sieve in the pipeline. A newly created sieve process accepts the first incoming number as its own prime and returns this prime as result such that it can be printed. After that it starts sieving. A generator process is used to feed the first sieve in the pipeline with a stream (list) of increasing numbers greater than one (see Figure 13.8).

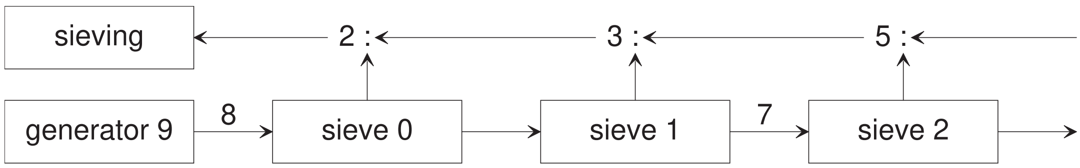


Figure 13.8 Snapshot of the process structure of the sieve processes.

In the programs below two concurrent solutions for the sieve of Eratosthenes are given. In the first toy example only a fixed number (four) of sieve processes is created. No more prime numbers can be found than the number of sieves created. So only four prime numbers will be found. The program shows very clearly that each sieve process is returning two results in a tuple: the prime number and a stream of numbers that is communicated to the next sieving process.

Sieve of Eratosthenes with a fixed number of sieve processes in the pipeline:

```
static_sieving:: [{proc} num]
static_sieving     =  [p1, p2, p3, p4]   where   s0        =  {Par} generator 2
                                                 (p1, s1)  =  {Par} sieve s0
                                                 (p2, s2)  =  {Par} sieve s1
                                                 (p3, s3)  =  {Par} sieve s2
                                                 (p4, s4)  =  {Par} sieve s3

sieve:: [num] -> (num, [num])
sieve (prime : stream)  =  (prime, filter prime stream)

generator:: num -> [num]
generator n    =  [n..100]

filter:: num -> [num] -> [num]
filter n [ ]        =  [ ]
filter n (x : xs)   =  x : filter n xs,   if (x mod n) ~= 0
                    =  filter n xs,       otherwise
```

The local selector function (p_i, s_i) in static_sieving selects objects being evaluated by a (parallel) process. So the argument s_i of a sieve is already under calculation by the previous sieving process. As explained in Section 13.2.5, a process type can be assigned to the sieve arguments. In this way the required communication stream between the sieving processes is accomplished.

In the second more general solution as many sieves are created as necessary. Each time a new prime number is produced at the end of the pipeline a fresh sieve is created and the pipeline is extended. Each individual sieve works as described above.

Sieve with as many sieve processes as necessary in the pipeline (with the functions sieve and generator as defined above).

```
dynamic_sieving:: [{proc} num]
dynamic_sieving     =  dynpipe ({Par} generator 2)

dynpipe:: {proc} [num] -> [{proc} num]
dynpipe [ ]  =  [ ]
dynpipe in  =  p : {Self} dynpipe s      where   (p, s)    =  {Par} sieve in
```

13.3.2 Warshall's algorithm

The following algorithm is a parallel version of Warshall's solution for the *shortest path problem*:

Given a graph G consisting of N nodes and directed edges with a distance associated with each edge, the graph can be represented by an $N * N$ matrix in which the element at the ith row and jth column is equal to the distance from node i to node j. Warshall's shortest path algorithm is able to find the shortest path within this graph between any two nodes.

Warshall's shortest path algorithm:

A path from node j to node k is said to contain a node i if it can be split in two paths, one from node j to node i and one from node i to node k ($i \neq j$ and $i \neq k$). Let $SP(j,k,i)$ denote the length of the shortest path from node j to node k that contains only nodes less than or equal to i ($0 \leq i$ and $1 \leq j,k$ and $i,j,k \leq N$).

So

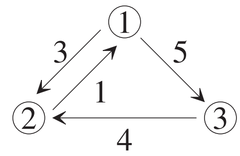
$$
\begin{aligned}
SP\,(j,k,0) &= 0 && \text{if } j=k \\
&= d && \text{if there is an edge from } j \text{ to } k \text{ with distance } d \\
&= \infty && \text{otherwise} \\
SP\,(j,k,i) &= \mathit{minimum}\,(SP\,(j,k,i-1),\, SP\,(j,i,i-1) + SP\,(i,k,i-1))
\end{aligned}
$$

Define a matrix M as follows: $M[\,j,k] = SP\,(j,k,i)$ for some i. The final shortest path matrix can be computed iteratively by varying i from 0 to N

using the equations as described above. In the *i*th iteration it is considered for each pair of nodes whether a shorter path exists via node *i*.

The Warshall algorithm is an interesting algorithm to test the expressiveness of parallel languages (Augusteijn, 1985) since it requires a special process structure containing a cycle (Figure 13.9).

To illustrate the algorithm it is applied to the graph on the right with the corresponding matrixes given below:



$M[j,k]_0$

0	3	5
1	0	∞
∞	4	0

$M[j,k]_1$

0	3	5
1	0	6
∞	4	0

$SP(2,3,1) = min\ (SP(2,3,0), SP(2,1,0)+SP(1,3,0)) = min\ (\infty,1+5)$

$M[j,k]_2$

0	3	5
1	0	6
5	4	0

$SP(3,1,2) = min\ (SP(3,1,1), SP(3,2,1)+SP(2,1,1)) = min\ (\infty,4+1)$

$M[j,k]_3$

0	3	5
1	0	6
5	4	0

Observing the algorithm it can be concluded that during the *i*th iteration all updating can be performed in parallel. It seems a good decision to create *N* parallel processes: one for each row that updates its row during each iteration step. In the *i*th iteration all the parallel processes need to have access to row *i* as well as to their own row. This can be achieved by letting parallel process *i* distribute its own row as soon as the *i*th iteration starts. At the end of the distributed computation the rows of the solution have to be collected.

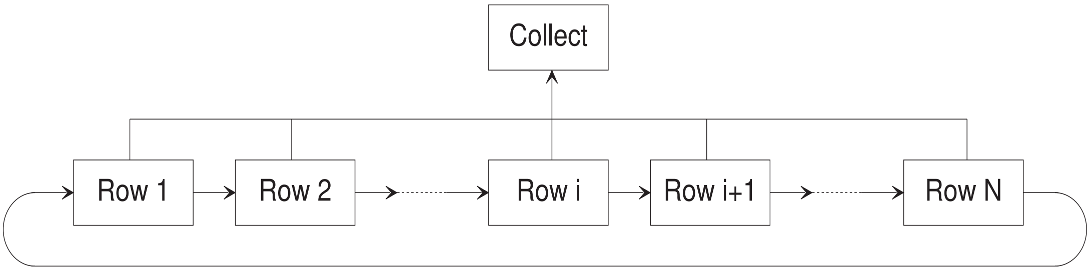


Figure 13.9 Snapshot of the process structure of Warshall's algorithm.

Initially, a parallel process $rowproc_i$ is created for each row of the matrix. Before $rowproc_i$ performs its *i*th iteration it distributes its own row to the other rowprocs. This is done in a cyclic pipeline, i.e. $rowproc_i$ sends its own row to $rowproc_j$ via $rowproc_{i+1}$, ... , $rowproc_{j-1}$ and $rowproc_j$ (counting modulo *N* from *i* to *j*).

It is rather difficult to express this distributing, updating and iterating in a parallel functional language. The cyclic process structure is created via a recursive local definition of a pair with as first element the final solution and as second element the output that will be produced by the *N*th process after it is created.

```
matrix * = =   [ [*] ]

warshall:: matrix num -> matrix num
warshall mat
   =  solution
      where (solution, output_rp_N) = create_procs (#mat) 1 mat output_rp_N

create_procs:: num -> num -> matrix num -> {proc}[[num]]
                                              -> ([{proc}[num]], {proc}[[num]])
create_procs size k [row_N] input_left_rp
   =  ([row_N_solution], output_rp_N)
      where   (row_N_solution, output_rp_N)
                         =  {Self} iterate size k 1 row_N input_left_rp
create_procs size k (row_k : restmat) input_left_rp
   =  (row_k_solution : rest_solutions, output_rp_N)
      where   (row_k_solution, output_rp_k)
                         =  {Self} iterate size k 1 row_k input_left_rp
              (rest_solutions, output_rp_N)
                         =  {Par} create_procs size (k+1) restmat output_rp_k

iterate:: num -> num -> num -> [num] -> [[num]] -> ([num],[[num]])
iterate size k i row_k rows
   =  (row_k, [ ]),                     if iterations_finished
   =  (solution, row_k : rest_output),  if start_sending_this_row
   =  (solution, row_i : rest_output),  otherwise
      where   iterations_finished          =  i > size
              start_sending_this_row       =  i = k
              row_i : xs                   =  rows
              (solution, rest_output)      =  iterate size k (i+1) next_row_k xs
              next_row_k     =  row_k,                              if i = k
                             =  updaterow row_k row_i dist_k_i,     otherwise
              dist_k_i       =  row_k ! (i – 1)

updaterow::[num] -> [num] -> num -> [num]
updaterow [ ]                    row_i                  dist_j_i  =  [ ]
```

```
updaterow (dist_j_k : restrow_j)    (dist_i_k : restrow_i)    dist_j_i
   =  min dist_j_k (dist_j_i + dist_i_k) : updaterow restrow_j restrow_i dist_j_i
      where    min m n     =  m,     if m < n
                           =  n,     otherwise

warshall  [ [   0, 100, 100,  13, 100, 100 ] ,
            [100,   0, 100, 100,   4,   9 ] ,
            [  11, 100,   0, 100, 100, 100 ] ,
            [100,   3, 100,   0, 100,   7 ] ,
            [  15,   5, 100,   1,   0, 100 ] ,
            [  11, 100, 100,  14, 100,   0 ] ]
```

13.4 Discussion

Writing concurrent programs is in general a much more difficult task than writing ordinary sequential programs. Writing concurrent programs in a functional language instead of in an imperative language has certain advantages and disadvantages.

With only *two* annotations, one for the creation of parallel processes and one for the creation of interleaved processes, already very complicated concurrent programs can be specified in an elegant and readable way using the associated type system. Processes can be created dynamically. For the communication between processes no additional primitives are needed. Communication is demand driven: whenever a process needs information from another process the information is communicated as soon as it is available. Flexible and powerful tools for the construction of frequently occurring process topologies can be defined using the expressive power of functional languages. Concurrent functional programs can be executed on any processor configuration, in parallel or just sequentially. In principle, the programmer can start with writing an ordinary sequential program. When this program is finished he or she can turn this program into a parallel version by creating processes for some of the function applications in the program.

Of course, many problems remain that are connected with concurrent programming in general. Sometimes it is very difficult to tell which function application really is worthwhile to be evaluated in parallel. In the worst case, the program has to be fully rewritten simply because the chosen algorithm is not suited for parallel evaluation at all. So one cannot expect a real speed-up when the chosen algorithm is not suited for parallel evaluation.

It is possible to write complicated concurrent functional programs with the proposed language extensions, but there are still many things to be desired. With explicit use of sharing, e.g. in graph rewriting systems, it would be not only possible to specify cyclic structures explicitly, but it would also be possible to specify process topologies directly. Besides, for some applications one would like to have the possibility of creating processes that reduce to head normal form or to spine normal form instead of to normal form. Furthermore, it should be possible to assign a particular process to a specific concrete processor. With such a facility a concurrent program can be optimally tuned to the available parallel architecture. For certain applications one would also like to have better control of the kind of information that is passed from one process(or) to another. One would like to ship not only data but also work (redexes). The control of the amount of information that is communicated and the moment at which this happens (synchronous, asynchronous, blocked) can be important as well.

To tackle the problems and desires mentioned above it is necessary to extend the graph rewriting model (see Chapter 14).

Summary

- A functional program can be turned into a *concurrent* functional program with help of a {Par} annotation (for the creation of *parallel* processes) and a {Self} annotation (for the creation of *interleaved* subprocesses).
- For the *communication* between processes *no special primitives are needed* since due to the lazy evaluation scheme process communication is also lazy.
- A process can *communicate* with *several other processes* at the same time using subprocesses that run interleaved *serving* demanding processes.
- In the *type specifications* of the function definitions the programmer can explicitly indicate that certain subexpressions of the program will be evaluated by a process. In this way complex process topologies can be specified in a clear way.
- With the proposed primitives and the use of higher order functions and function composition complex concurrent programs can be expressed elegantly. Divide-and-conquer parallelism can be specified, but also more complicated process topologies such as a *pipeline* of processes or a *cyclic* process structure.
- Concurrent functional programming is still in its infancy. Many other language constructs for special forms of concurrency are possible.
- For a better understanding of the possibilities, limits and properties of concurrent functional programming the underlying computational model of graph rewriting has to be extended to a parallel context.

EXERCISES

13.1 Give a sequential definition of fib n that has a linear complexity with respect to n. Discuss the differences in efficiency compared with the parallel version of Figure 13.5.

13.2* Write a Miranda function that computes the determinant of a matrix. Use this function to compute the inverse of a matrix using the method of Cramer. Use {Par} and {Self} to construct a parallel program. Let the main process compute the determinant of the matrix. Each element of the inverse matrix is computed by a parallel process.

13.3 Write a function that computes the *n* queens problem (see Exercise 2.5) in parallel using a divide-and-conquer approach. The function should take the board size and a threshold as an argument.