Process annotations and process types

M.C.J.D. van Eekelen and M.J. Plasmeijer
Department of Computer Science, University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.
email: marko@cs.kun.nl, rinus@cs.kun.nl

Abstract

In a concurrent functional language processes are functions that are executed concurrently. Using special annotations based on lazy copying arbitrary dependencies between these functions can be used to specify arbitrary networks of processes. The communication and synchronization between the processes is realized using the lazy evaluation principle without any additional communication primitves. Communication takes place when a process demands a value that is being calculated by another process. A type system is proposed that enables the programmer to specify higher order functions as process skeletons for frequently occurring process structures.

Introduction

Functional languages have as advantage that, when a result is obtained, it will always be the *same* independent of the chosen order of reduction. This property makes functional languages very suited for *interleaved* and *parallel* evaluation. As is the case with eager evaluation, one has to be careful since changing the evaluation order can change the termination behaviour of a program.

For concurrent functional programming one certainly would like to have an analyser that automatically marks expressions that can safely be executed in parallel. A strictness analyzer can be used for this purpose. But, other kinds of analyzers are needed to determine whether parallel evaluation of expressions is worthwhile. With the creation of a 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. In order to gain efficiency tasks will have to represent a sufficiently large amount of work with limited inter-process communication. The amount of work performed by a task and the amount of communication between them is of course undecidable. The actual overhead is also depending 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 even difficult to solve for a human-being.

So, in this paper we assume that the programmer *explicitly* has to define the concurrent behaviour of the functional program, 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 of processors (with local memory) 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 support concurrent programming. However, in several experimental languages concurrency primitives have been proposed in the form of annotations or special identity functions (Kluge (1983), Goguen *et al.* (1986), Hudak & Smith (1986), Burton (1987), Glauert *et al.* (1987), Vree & Hartel (1988), (Eekelen *et al.* (1990)). With these primitives the default evaluation order of an ordinary functional program can be changed such that a concurrent functional program is obtained. There is not yet a common view on which kind of basic primitives are handy or definitely needed for concurrent functional programming although there is some agreement in the sense that all proposals provide a way to create concurrent processes dynamically.

This paper therefore does not reflect *the* way to achieve concurrency but it presents a promising method to define concurrency in an elegant way. The concurrent behaviour of a program is defined by means of special high-level concurrency *annotations* with an associated *type system*. The annotations used in this paper are based on the concept of lazy copying (Eekelen *et al.* (1990)). Using them together with the associated type system it is possible to specify a very large class of process structures in an elegant manner.

First a brief introduction is given on concurrency in general and concurrency in the context of functional languages (Section 1). A couple of special annotations for concurrency are added to a Miranda-like syntax in Section 2. These two annotations make the specification of rather complex networks of communicating functional processes possible. The type system enables the programmer to specify higher order functions for frequently occurring process skeletons (this is in contrast to the process skeletons in Darlington *et al.* (1991) which are inherently predefined). In Section 3 some more elaborate examples of concurrent functional programs are shown. Section 4 discusses the advantages and the disadvantages of the functional concurrency primitives that are used in this paper.

1 Concurrency and functional programming

A **concurrent program** is a program in which parts of the program are running **concurrently**, i.e. *interleaved* or in *parallel* with each other. Such a part of the program is called a **process**. The algorithm that is executed by a program is called a **task**. With each task a certain amount of work is involved. In a concurrent program the task that has to be performed is split-up in *sub-tasks* and each such a sub-task is assigned to a concurrently executing 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.

Why concurrent functional programming?

Imperative programming languages have as disadvantage that one cannot always assign an *arbitrary* sub-task in a program (such as a procedure call) to a process. A procedure call can have side-effects via access to global variables. Generally, unintended communication between processes may take place in this way such that the correctness of the program is no longer guaranteed.

The main disadvantage however is that inter-process communication has to be defined *explicitly*. For distributed architectures message passing primitives have to be used (e.g. send and receive primitives or rendez-vous calls). With these primitives all possible communication situations have to be handled. As a consequence, reasoning about such programs is often practically impossible.

Advantages of concurrent functional programming

In a functional language the evaluation of *any* expression (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 albeit that some evaluation orders may influence the termination behaviour of a program.

In a concurrent functional program a task assigned to a process consists of the evaluation of a function. Communication between the processes takes place *implicitly* when one process needs the result calculated by another. No additional primitives for process communication are needed. Reasoning about the correctness of the algorithm is of the same complexity as 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 is defect or when it is overloaded with work it is always possible to change the order of evaluation and the number of task created in optimal response to the new actual situation that arises at run-time.

Besides the advantages mentioned above the programmer of concurrent functional program has the full power of a functional language to his disposal. This means that it is possible to write elegant and short programs.

Disadvantages of concurrent functional programming

In a concurrent functional programming the programmer has to define how the concurrent evaluation of his program will take place. This means that a program in a functional language is no longer just an executable specification in which one does not have to worry about how expressions are being evaluated.

On the other hand, already for ordinary functional programs there are situations in which a programmer cannot be totally unaware of the evaluation order of his program. For instance, patterns specified in 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.

Most functional languages already have facilities to influence the default evaluation order (e.g. Miranda[1] (Turner (1985)), Haskell (Hudak *et al.* (1991)), Clean (Brus *et al.* (1987)) and LML (Augustsson (1984))). In concurrent functional programming the programmer has to be even *more* aware of the evaluation order of a functional program. He should be able to control the reduction order in such a way that he can turn the program into the desired concurrent program. In the compilation process of functional languages so many transformations take place that for the average programmer it is very hard to predict in which order expressions are evaluated. For this reason for the control of the evaluation order we have chosen for the following approach. Whenever a programmer decides to control the concurrent behaviour of the program, he must explicitly specify which processes are to be created, what their task is, and how the original process should proceed. The programmer must have some understanding of the standard evaluation order, but he should not rely on knowledge of how the evaluation takes place in a particular implementation.

[1] Miranda™ is a trademark of Research Software Ltd.

2 Process annotations and an associated type system

Suppose that one would like to increase the execution speed of the following function definition of the well-known fibonacci function by introducing parallelism in the computation.

Standard definition of the fibonacci function:

```
fib:: num -> num
fib 1  =  1
fib 2  =  1
fib n  =  fib (n - 1) + fib (n - 2),   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 so-called **divide-and-conquer** parallelism.

To create a parallel process we will prefix a function application with a special annotation: {Par}. The semantics of this annotation will be explained in the following section. A process created with a {Par} will have as task the parallel evaluation of the annotated function to *normal form*. With this annotation the desired parallel variant of fib can be specified as follows:

Divide-and-conquer fibonacci:

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

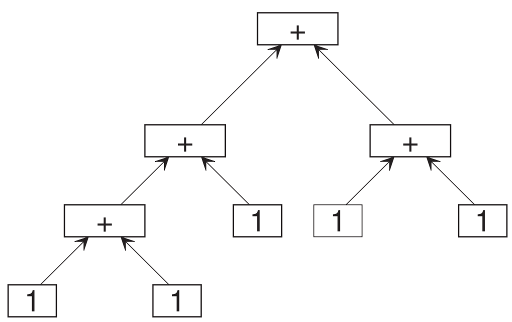


Figure 1 Snapshot of a tree of processes computing fib 5; the arrows indicate the direction of the flow of information between the processes.

On many parallel architectures it will not be worthwhile to evaluate fib n in parallel in such a way. To turn the fine-grain parallelism into coarse-grain parallelism, in the example below a threshold is introduced that ensures that the processes at the bottom of the tree have some substantial amount of work to do.

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),      n > threshold
        =  fib (n - 1) + fib (n - 2),                  n > 2

threshold:: num
threshold    =  10
```

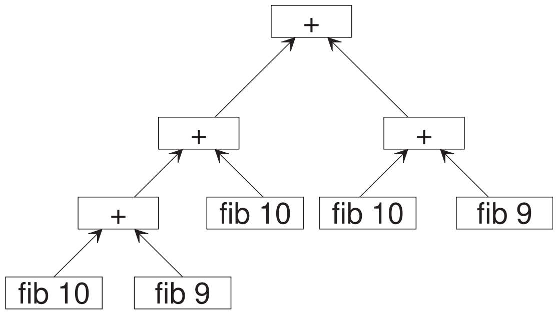


Figure 2 Tree of processes with threshold computing fib 13.

2.1 Creating parallel processes with the {Par} annotation

The {Par}-annotation can be used in the body (the right-hand-side) of any function definition to create a parallel process.

When a function with a *{Par}-annotation* in its body is evaluated by a process (the **father** process) the steps described below are performed for each occurrence of {Par} in that body. After these steps have been performed the father process continues as usual with the regular evaluation of the function body. 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 simple 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 not a fundamental restriction.

A function application that is annotated with a {Par}-annotation is said to be in **Process Normal Form** (**PNF**). There are more cases in which expressions are known to be in PNF. A complete survey of these cases is given in Section 2.5. When an expression is in PNF this means that, *when its evaluation is demanded*, it will either be in *normal* form or it will be reduced to *normal* form by one or more *processes*. This is in contrast with the standard evaluation order in which only head-normal-form reduction of subexpressions is demanded.

The following steps are performed when a **Par-annotated function application** {Par} f a_1 a_2 ... a_n is encountered in a function body during reduction:

1. First the father process reduces each argument of f (from left to right) to *normal form* (!) unless the argument is in PNF;
2. Then a new process is created (the **child** process) that *preferably* runs in *parallel* on a *different* processor, with as task the evaluation of f a_1 a_2 ... a_n to *normal form*.

When a process needs the information calculated by another process it has to *wait* until the result is available. As soon as part of the result has reached a normal form the information is passed to the demanding process (see also 2.3).

Motivation for the chosen semantics

To reduce the complexity of the creation of the task and to reduce the overhead involved with inter-process communication, it is generally more efficient to evaluate the arguments of the function before the task is created. One has to keep in mind that a parallel task on a distributed memory architecture has to be copied to another processor. A small expression requires fewer communication than a large one. It is assumed that an expression in normal form often requires less space than the original redex. For instance, in the fib-example above fib 5 is a smaller expression than fib (6 - 1).

When a task is created, all arguments of f are in PNF because they either have been reduced to normal form by the father process or they where already in PNF. When the father process is evaluating arguments there is a danger that the termination properties of the program are changed when they represent infinite calculations. This can be avoided by creating an additional process for the evaluation of such an argument. When an argument is in PNF the father process skips the evaluation of the argument since a process is already taking care of it already or, upon demand, a process will take care of it in the future.

The idea of the {Par}-annotation is to create a parallel process executing on another processor. However, if the creation on another processor is somehow in practice not possible it is allowed to create the process on the same processor as the father process.

In many cases parallel processes are created to perform a substantial task. Therefore by default the task consists of the reduction of the indicated expression to normal form and not just to head normal form.

2.2 Creating interleaved processes with the {Self} annotation

Consider again the fib-example. In the presented solutions processes are not used in an optimal way. This is caused by the fact that the father process cannot do much useful work because it has to wait for the results of its child processes. A better solution would be that one of the two arguments would be reduced by the father process itself.

The idea in the fib example below is that the first argument is reduced by the father process in parallel with the evaluation of the second argument.

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

However, the solution assumes that the father process will continue with the evaluation of the first argument. In reality this may not be the case. As specified, the first argument will be calculated by the father process and the second one by another parallel process.

But, if the father process happens to start with the calculation of the second argument, it will wait until the parallel child process has communicated the result. Hereafter the father can evaluate the first argument. So, although the specification is fulfilled, the desired parallel effect may not be obtained. The evaluation of the first argument is then not really performed *concurrently* with the evaluation of the second argument, but sequentially *after* the evaluation of the second argument has been completed by the parallel process.

The problem illustrated in the example arises because the programmer has made some (possibly wrong) assumptions on the standard evaluation order of the program. It is important not to make any assumptions like this. The problem can be solved by explicitly controlling the reduction order in such situations.

A new annotation is introduced that creates an *interleaved* executing child process on the same processor as the father process: the *{Self}-annotation.*

Divide-and-conquer fibonacci in which the father processor is forced to do some substantial work as well:

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

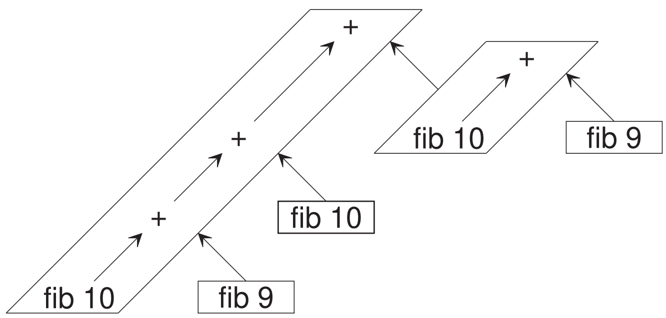


Figure 3 Tree of processes computing fib; the left branch is now computed on the same processor by different interleaved running processes.

The {Self}-annotation can be used where a {Par} can be used: in the body (the right-hand-side) of any function definition. When a function with some {Self}-annotations in its body is evaluated by the father process the appropriate step is performed for each occurrence of {Self} in that body.

When a **Self-annotated function application** {Self} f a_1 a_2 ... a_n is encountered in a function body during reduction, a **child** process is created as a sub-process that runs *interleaved* on the *same* processor as the father process with as task the evaluation of f a_1 a_2 ... a_n to *normal form.*

Hence, a {Self}-annotated function application is in PNF.

Motivation for the chosen semantics

In many cases a better utilisation of the machine capacity can be achieved when a sub-process is created on the same processor as the father process. The annotation is in particular handy to create so-called **channel processes**. When several processes demand information from a particular process, it is useful to create a sub-process for each demanding process to serve the communication demand.

In contrast with the {Par}-annotation now the arguments a_1 a_2 ... a_n are not evaluated to normal form. This is not needed now because, since the child process is created on the same processor, no information has to be copied.

The indicated expression f a_1 a_2 ... a_n is evaluated to normal form just as for processes created with the {Par}-annotation. This has the advantage that it makes communication transparent with respect to the kind of process that is communicating (created with {Self} or with {Par}).

2.3 Communication and lazy evaluation

A very important and nice aspect of concurrent programming in functional languages is that the communication is not defined explicitly. **Communication** is defined *implicitly* by making use of the lazy evaluation order: communication between processes takes place when one process needs a value that is being evaluated by another process.

In the sequential case when a value is needed, it is evaluated by the process itself. In the concurrent case when a value is needed that is being evaluated by another process, the demanding process has to *wait* until the value becomes available. Just as is the case when the result of an ordinary program is printed as soon as possible (communicated to an external device), communication between processes within a program can take place as soon as part of the result of a sub-task is available: as soon as a head-normal-form is reached. So, communication is lazy just as the evaluation is lazy. This property is used to create information streams between processes.

Example of a communication stream between processes. The definition of the function generator is assumed to be used also in several other examples in this paper.

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

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

When the father 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 father process (e.g. 3 :). In this example effectively the generator and the map process are operating in a (very small) pipeline.

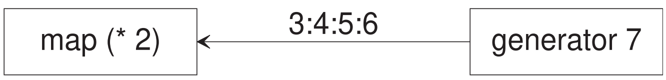


Figure 4 Simple pipeline between the processes map and generator; the values 3:4:5:6 are being communicated.

2.4 Expressive power of {Par} and {Self}

An advantage of functional languages is that it is relatively easy to define general tools for the creation of parallelism by using annotations like the {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,                                               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 communicating processes.

Static pipeline of processes:

```
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))
```

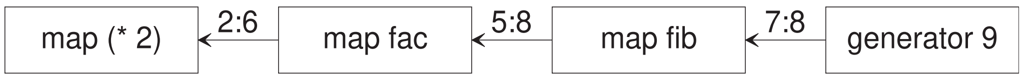


Figure 5 A pipeline of processes.

Using higher order functions general skeletons can be defined to create frequently occurring process structures. Often these are parallel variants of the basic building blocks used in sequential functional programming.

A general pipeline defined with the {Self}-annotation to force the construction of the pipeline:

```
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]
```

The following parallel version of map implements vector-like processing since 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] → … → [1, 1, 2, 720]
```

A process can create one or more sub-processes with the {Self}-construct. These sub-processes (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 sub-process that reduces the demanded information to normal form. A process with its sub-processes in a functional language acts more or less like a process with its channels in a message passing language like CSP (Hoare (1978)). Serving sub-processes is like sending information over a channel to any process requesting that information.

Parallel quicksort. In this parallel version of the quicksort algorithm 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 sub-list. The father process will supply the appropriate sub-list to each of its child processes.The father process can perform both these tasks "simultaneously" with help of two sub-processes that run interleaved with each other.

```
sorter:: [num] -> [num]
sorter list     =  quick_sort list,         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]
```

2.5 A type system for processes

The knowledge that an expression is in PNF is of importance when a new parallel process is created (see Section 2.1).

In the examples above, the knowledge that an expression is in PNF was only *locally* used inside a function body. To make full usage of the expressive power of functional languages it is necessary that the knowledge that an expression is in PNF can be expressed on a *global* level. To make this possible a special type attribute {proc} is introduced.

An expression has type **{proc} T** when it is of type T and furthermore *known* to be 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 can be specified as follows making use of process attributes:

```
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 father process upon the creation of a new parallel process does not need to force the evaluation of its first parameter knowing that it is already in PNF.

A type inferencer cannot derive that an argument of a function has a process type because it cannot be guaranteed that the function is always called with an actual argument in PNF. Therefore, process types have to be defined *explicitly* by the programmer. 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 we assume that these actions are performed after the normal type inferencing/checking has been done.

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 $e_1 \dots e_n$ or {Self} f $e_1 \dots e_n$ 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 is returning 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 also referred to as the **decomposition property**.

Assume that g is of type [num], then using the decomposition property the expression x in

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

is of type {proc} num and with similar reasoning the following type specification is accepted:

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

A more practical example of the use of the decomposition property will be given in the next section.

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 in 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
```

  the type of f ({Par} generator 3) is num due to deprocessing.

  However, with the more general polymorphic definition:

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

  the type of f ({Par} generator 3) is {proc} num due to 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.

3 Examples of concurrent functional programs

In this section two more elaborate examples of concurrent functional programs are given: a concurrent version of the sieve of Eratosthenes to compute prime numbers and of Warshall's algorithm to solve 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. It is not the intention to show how ultimate speed-ups can be achieved for both problems.

3.1 Sieve of Eratosthenes

The sieve of Eratosthenes 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. The process topology is shown in Figure 6.

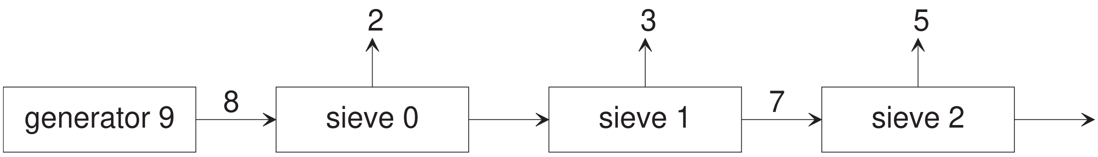


Figure 6 Snapshot of the process structure of the sieve of Eratosthenes.

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,    x mod n ~= 0
                   =  filter n xs,        otherwise
```

Notice that 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 2.6 a process type can be assigned to the sieve arguments. In this way the wanted 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 of Eratosthenes with as many sieve processes as necessary in the pipeline.

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

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

3.2 Warshall's algorithm

The following algorithm that is considered, 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 x N* matrix in which the element at the i^{th} row and in the j^{th} 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:

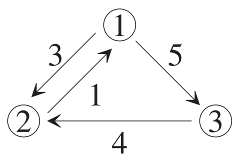
> A path from node *j* to node *k* is said to contain a node *i* if it can be split in two paths, one from node *j* to node *i* and one from node *i* to node *k* ($i \neq j$ & $i \neq k$). Let *SP(j,k,i)* denote the length of the shortest path from node *j* to node *k* that contains only nodes less than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

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

> 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.

To illustrate the algorithm it is applied to the following graph



with the corresponding matrixes:

$M[j,k]_0$

0	3	5
1	0	∞
∞	4	0

$M[j,k]_1$

0	3	5
1	0	6
∞	4	0

P(2,3,1) = min (SP(2,3,0),SP(2,1,0)+SP(1,3,0)) = min (∞,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 (∞,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.

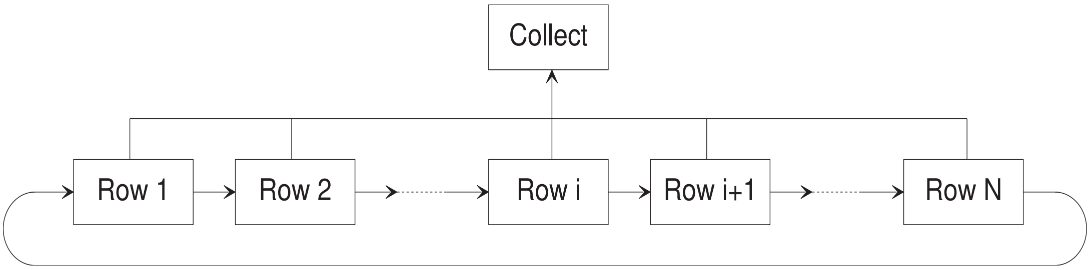


Figure 7 Snapshot of the process structure of Warshall's algorithm.

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
    =  shortest_paths
       where
       (shortest_paths, output_rowproc_N)
           =  create_rowprocs #mat 1 mat output_rowproc_N

create_rowprocs::num -> num -> matrix num -> {proc} [[num]] ->
                                             ([{proc} [num]], {proc} [[num]])
create_rowprocs size k (row_k:restmat) input_left_rowproc
    =  (row_k_solution:rest_solutions, output_rowproc_N)
       where
       (row_k_solution, output_rowproc_k)
           =  {Self} iterate size k 1 row_k input_left_rowproc
       (rest_solutions, output_rowproc_N)
           =  {Par} create_rowprocs size (k+1) restmat output_rowproc_k
```

```
create_rowprocs size k [row_N] input_left_rowproc
    =  ([row_N_solution], output_rowproc_N)
       where
       (row_N_solution, output_rowproc_N)
          =  {Self} iterate size k 1 row_N input_left_rowproc

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

updaterow::[num] -> [num] -> num -> [num]
updaterow [ ] rowi dist_j_i    =  [ ]
updaterow (dist_j_k:restrow_j) (dist_i_k:restrow_i) dist_j_i
    =  minimum dist_j_k (dist_j_i + dist_i_k) : updaterow restrow_j restrow_i dist_j_i
       where   minimum m n   =  m,     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 ] ]
```

4 Concluding Remarks

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.

An advantage of the proposed method is that with only *two* annotations, one for the creation of parallel processes and one for the creation of interleaved processes, due to the associated type system rather complicated concurrent programs can be specified in an elegant and readable way. 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 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 real speed-up when the chosen algorithm is not suited for parallel evaluation. Furthermore, when the process topology and communication structure is very complex, the corresponding functions will be very complex as well and therefore, still hard to understand.

Although functional programs can be turned into a parallel version and real speed-ups can be obtained, functional languages are less suited when ultimate efficiency has to be achieved on parallel architectures. One has to bear in mind that sequential functional programs in general run slower than their imperative counterparts. This factor also holds for parallel programs (on each processor). For ultimate speed-ups the assembly language of the concrete machine is best suited at the costs of reliability and readability.

With the proposed language extensions it is possible to write elegant concurrent functional programs. Nevertheless many wishes remain. With explicit use of sharing e.g. in graph rewriting systems it would be possible to specify process topologies directly. Also, for some applications one would like to have the possibility to create 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 a better control on 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 on which this happens can be important as well.

The practical usability and efficiency of the proposed language extensions will be further investigated in the context of the lazy functional graph rewriting language Concurrent Clean (Nöcker *et al.* (1991)).

References

Augusteijn L. (1985). The Warshall shortest path algorithm in POOL-T. Philips Research Laboratories, Eindhoven, The Netherlands. Esprit project 415 A Doc. 0105.

Augustsson, L.A. (1984). A Compiler for Lazy ML. Proceedings of the 1984 ACM LISP and Functional Programming Conference.

Brus, T., M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer (1987). Clean - A Language for Functional Graph Rewriting. Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, *Springer Lec. Notes on Comp.Sci.* 274, 364 - 384.

Burton, F.W. (1987). Functional Programming for Concurrent and Distributed Computing. *The Computer Journal* **30-5**, 437-450.

Darlington, J., A.J. Field, P.G. Harrison, D. Harpe, G.K. Jouret, P.L. Kelly, K.M. Sephton, D.W. Sharp (1991). Structured Parallel Functional Programming. in: Glaser, H. and P. Hartel (Eds.). Proceedings of the Third International Workshop on the Implementation of Functional Languages on Parallel Architectures. Southampton, June 1991. pp 31-51.

Eekelen, M.C.J.D. van, M.J. Plasmeijer, J.E.W. Smetsers (1990). Parallel Graph Rewriting on Loosely Coupled Machine Architectures, in Kaplan, S. and M. Okada (Eds.) Proc. of the 2nd Int. Worksh. on Conditional and Typed

Rewriting Systems (CTRS'90), 1990. Montreal, Canada, *Springer Lec. Notes Comp. Sci.* **516**, pp 354 - 370.

Glauert, J.R.W., J.R. Kennaway, M.R. Sleep (1987). DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction. *ICL Technical Journal* **5**, 509-537.

Goguen, J., C. Kirchner, J. Meseguer (1986). Concurrent term rewriting as a model of computation. Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico. *Springer Lec. Notes Comp. Sci.* **279**, 53-94.

Hoare, C.A.R. (1978). Communicating Sequential Processes. *Comm. ACM.* 21 (8), page 666-677.

Hudak, P., L. Smith (1986). Para-functional Programming: A Paradigm for Programming Multiprocessor Systems. *12th ACM Symp. on Principles of Programming Languages*, 243-254.

Hudak, P., Ph. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, S. Peyton Jones, M. Reeve, D. Wise, J. Young (1991). Report on the functional programming language Haskell. Version 1.1.

Kluge, W.E. (1983). Cooperating reduction machines. *IEEE Transactions on computers* **C-32/11**, 1002-1012.

Nöcker E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen, M.J. Plasmeijer, Concurrent Clean, in Aarts, E.H.L., J. van Leeuwen, M. Rem (Eds.), Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91) Vol II, Eindhoven, The Netherlands, LNCS 506, pp 202-220, Springer Verlag, June 1991.

Turner, D.A. (1985) Miranda: A non-strict functional language with polymorphic types. Proc. of the conference on Functional Programming Languages and Computer Architecture, *Springer Lec. Notes Comp. Sci.* 201, 1 - 16 .

Vree, W.G., P.H. Hartel (1988). Parallel graph reduction for divide-and-conquer applications; Part I - programme transformations. University of Amsterdam. Internal Report D-15.