# Programming With Functional Nets

**Martin Odersky**
**École Polytechnique Fédérale de Lausanne**

March 9, 2000

## 1 Introduction

What is a good foundation of programming? Such a foundation should be simple enough to be understood without too much effort, formal and precise, so that formal reasoning about programs is supported, and universal, so that a large range of programs can be mapped into the foundation directly, without too much encoding.

Over the last 20 years an operational view of program execution based on rewriting has become widespread. In this view, a program is seen as a term in some calculus, and program execution is modeled by stepwise rewriting of the term according to the rules of the calculus. The operational semantics still has to be complemented with a logic for program verification and a collection of laws of programming. This view is exemplified in functional programming [4, 17], in modern theories of objects [1], as well as in concurrent systems based on message passing such as CSP [15], CCS [19] or $\pi$-calculus [20].

Ideally, the terms of a programming notation should match closely the terms of the underlying calculus, so that insights gained from the calculus can immediately be applied to the language. This aim guided the design of modern functional languages such as Scheme [2], ML [22], or Haskell [6], based on $\lambda$-calculus, as well as concurrent languages such Occam [18], based on CSP, or Pict [23] and Piccola [3], which are based on $\pi$-calculus.

This chapter gives an introduction to *functional nets* which support functional, object-oriented, and concurrent programming in a simple calculus based on rewriting. Functional nets raise out of a combination of key ideas of functional programming and Petri nets [26]. As in functional programming, the basic computation step in a functional net rewrites function applications to function bodies. As in Petri-Nets, a rewrite step can require the combined presence of several inputs (where in this case inputs are function applications). This fusion of ideas from two different areas results in a style of programming which is at the same time very simple and very expressive.

We introduce functional nets by means of a sequence of examples taken from a broad range of programming areas and styles. We show how functional nets can directly model sequential functional programming, imperative programming, and their object-oriented variants. We also show how functional nets can concisely capture common schemes of concurrency control. Thus, functional nets can be taken as the foundation of a new breed of wide spectrum languages which allow flexible combinations of techniques from functional, imperative and object-oriented programming together with concurrency. Unlike some existing approaches to such languages [25, 13, 27], functional nets are based on a single concept, not an amalgamation of features coming from different domains.

Functional nets have a theoretical foundation in *join calculus* [10]. A companion paper [21] explores the relation between the two in more detail. We will concentrate here on the programming aspects of functional nets, without going too deeply into their semantics. The notation for writing examples of functional nets is derived from Funnel, a new language developed by the programming methods group at EPFL. Funnel has been designed as a small language that supports flexible combinations of functional and object-oriented programming styles with functional nets as a common basis[1]. Even though we have to explain a bit of the notation as we go along, the notation by itself is not the primary focus of this note. The principles and utility of functional nets would be equally valid using a different notation. There are also other languages which are based on join calculus, and which express the constructs of functional nets in a different way, e.g. Join[11] or JoCaml[9].

The notation we use here is statically typed, using a standard type system with recursive records, structural subtyping and polymorphism. A type theoretic foundation of the notation can be developed along the lines of system $F_2$ with subtyping [7]. We use *local type inference* [24] to reduce the amount of type annotations which need to be given explicitly. Again, we omit the details here.

The rest of this paper is structured as follows. Section 2 introduces a purely functional subset of functional nets. Section 3 presents the full formalism which supports concurrent execution. Section 4 shows how functional nets model imperative programming. Section 5 presents a collection of process synchronization techniques implemented as functional nets. Section 6 concludes.

## 2 Functions and Objects

In functional programming, the fundamental execution step rewrites a function application to a function body, replacing formal parameters with actual arguments. For example, consider the usual definition of a min function

**def** min (x: Int, y: Int) = **if** (x < y) x **else** y

and the program min (4, 3). Three rewrite steps are needed to reduce the program to its answer “3”:

[1]An earlier version of Funnel, described in [21], was named Silk. We changed the name because of the possible confusion with the concurrent C dialect Cilk, which is pronounced the same way.

$$
\begin{array}{ll}
 & \mathsf{min\ (4,\ 3)} \\
\rightarrow & \textbf{if}\ \mathsf{4} < \mathsf{3}\ \textbf{then}\ \mathsf{4}\ \textbf{else}\ \mathsf{3} \\
\rightarrow & \textbf{if}\ \mathsf{true}\ \textbf{then}\ \mathsf{4}\ \textbf{else}\ \mathsf{3} \\
\rightarrow & \mathsf{4}
\end{array}
$$

The first of these steps rewrites a function application, and the other two steps rewrite the built-in operator $<$ and the built-in conditional expression. In principle, the built-ins can themselves be represented as functions, so that one can do with *only* the rewrite rule for function applications. In practice, we will simply assume appropriate reduction rules for built-ins. In the following, we will let $\rightarrow$ denote single-step reduction by application of a single rewrite rule. $\twoheadrightarrow$ will denote multi-step reduction by application of 0 or more rules.

When one has nested function applications, one has a choice which application to evaluate first. We will follow a *strict* evaluation strategy which evaluates all function arguments before rewriting the application. E.g. min (4, min (3, 5)) rewrites as follows.

$$\mathsf{min\ (4,\ min\ (3,\ 5))} \twoheadrightarrow \mathsf{min\ (4,\ 3)} \twoheadrightarrow \mathsf{3}$$

Some functional languages, such as Haskell, follow a *lazy* strategy instead, where arguments are passed unevaluated.

### Aggregation

Functions in Funnel can aggregated to form records. Here is an example of a record type (with several more functions left out for brevity):

```
type Complex = {
  def re: Float
  def im: Float
  def plus (c: Complex): Complex
}
```

This defines a record with three functions, re, im, and plus. Functions re and im are parameter-less, whereas plus takes a single parameter of type Complex. A complex number can be constructed using a function like makeComplex:

```
makeComplex (r: Float, i: Float): Complex = {
  def re = r
  def im = i
  def plus (c: Complex) = makeComplex (re + c.re, im + c.im)
}
```

Values of type complex can then be formed by calling the constructor function:

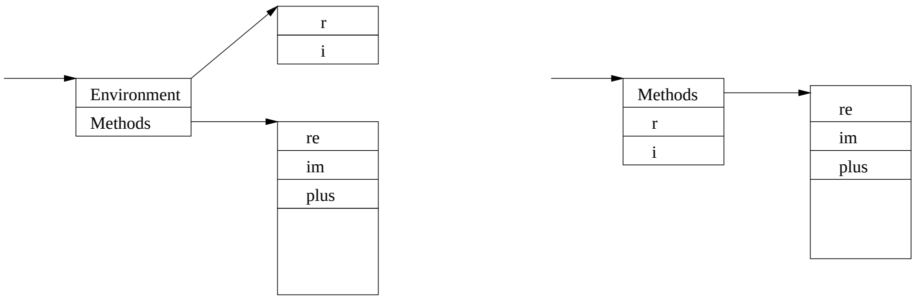


Figure 1: Object Representations

---

**val** one = makeComplex (1.0, 0)
**val** i = makeComplex (0, 1.0)

Value definitions like these evaluate their right hand sides, binding it to the name given on the left-hand side. Contrast this with a parameterless function definition like

**def** oneExp = makeComplex (1.0, 0)

This definition does not entail any evaluation itself, but each time oneExp is used the right-hand side is evaluated anew.

As usual, fields of record values are selected by an infix period. E.g. the expression one.plus (i) evaluates to the complex number $1 + i$.

### Objects

We have explained so far our notation in terms of functions and records. Equally well, we could have used object-oriented terminology. Records are then called objects, and their fields (which are always functions in our case) are called methods. Unlike many object-oriented programming languages, we have no need for a special syntactic form for object construction such as **new**. Instead, an object is simply defined by constructing its record of methods. The construction can be wrapped in a function itself, as we have seen in the case of makeComplex. Furthermore, the record methods may refer to variables bound in their environment. For instance, the methods in the record created by makeComplex refer to the parameters of makeComplex.

Looking at the situation from an implementation viewpoint, this has two consequences. First, since the created record survives the execution of the enclosing creator function, it will have to be stored in the heap rather than on the stack. Second, together with the record we will have to store the values of makeComplex's actual parameters, since these values can be accessed during evaluation of the record's methods. A good place to store these *environment variables* is as instance variables of the created heap object itself. The methods, on the other hand, need not be allocated once per call to makeComplex. Since their code is the same each time makeObject is called, we can share their implementation among all calls to makeComplex. A record can thus be represented by a *method table* and an *environment*, as it is depicted on the left side of Figure 1. When calling a method, we pass the environment as additional parameter. This is very similar the *closure* representation used in functional programming for first-class functions. The only difference is that a closure consists of an environment and a single function code pointer, whereas our record representation shares a single environment between all methods of a record.

We can also eliminate the indirection to the environment by making the method pointer itself the first entry of the environment. A record is then identified by just a pointer to its environment. This situation is depicted on the right of Figure 1. It corresponds exactly to the standard object layout used in many object-oriented languages. Two differences remain, however. First, fields of an environment are local to the object itself, so that they can be accessed only through a method of the record. In object-oriented terms, we would say that all instance variables are *private*[2]. Second, fields in an environment are always immutable, reflecting the purely functional nature of the language presented so far. In the next sections, we will introduce a richer language which lets us express mutable variables. But for now, we will stay in the purely functional subset.

Record type definitions can be recursive, as is shown in the following definition of a type for lists.

```
type SimpleList [t] = {
    def isEmpty: Bool
    def head: t
    def tail: SimpleList [t]
}
```

This says that a SimpleList is a record consisting of three functions. Function isEmpty returns true iff the list is empty, function head returns the first element of a (nonempty) list and function tail returns the list consisting of all elements but the first one. The definition has a type parameter 't' which represents the element type of the list. To distinguish type parameters from value parameters, we always enclose the former in square brackets [...]. Both types and functions can have type parameters.

Given a definition of lists as above, how can values of the type be defined? As in the case of makeComplex, we create constructor functions for lists. Two

[2] Actually, the term *private* does not have a standard meaning in OOP. In Simula or Smalltalk a private field can only be accessed from within methods of the same object, whereas in Java or C++ it can be accessed from any object which is a direct instance of the class which defined the field. Our scheme implements the Simula/Smalltalk meaning of the term.

constructor functions are needed, depending on whether the constructed list is empty or not. The following code defines the two list constructors Nil and Cons.

```
val SimpleList = {
    def Nil [t]: SimpleList [t] = {
        def isEmpty = true
        def head = error ("Nil.head")
        def tail = error ("Nil.tail")
    }
    def Cons [t] (x: t, xs: SimpleList [t]): SimpleList [t] = {
        def isEmpty = false
        def head = x
        def tail = xs
    }
}
```

The Nil function creates an empty list. Taking the head or tail of such a list results in an error, which is expressed here by calling the predefined error function. The Cons function creates a list consisting of a given head x and a given tail xs. Both Nil and Cons take a type parameter t. We also say they are *polymorphic* in the element type of the constructed list. The constructors Nil and Cons are themselves methods of a record value called SimpleList. This record plays the role of a module in Modula-2 or Oberon. Note that SimpleList is used as a name for both a record type and a record value. This is possible since Funnel has different name spaces for types and values.

Examples of clients of the SimpleList abstraction are the following definitions of a length function and a non-destructive append function:

```
def length [t] (xs: SimpleList [t]) =
    if (xs.isEmpty) 0
    else 1 + length (xs.tail)

def append [t] (xs: SimpleList [t], ys: SimpleList [t]) =
    if (xs.isEmpty) ys
    else SimpleList.Cons (xs.head, append (xs.tail, ys))
```

Actually, one could equally well define length and append as *methods* of SimpleList objects. The following code defines a List type which is itself defined in terms of SimpleList and adds a set of methods to it.

```
type List [t] = {
  with SimpleList [t]

  def tail: List [t]
  def length: Int
  def append (ys: List [t]): List [t]
  def forall (p: t → Bool): Bool
  def filter (p: t → Bool): List [t]
  def map [r] (f: t → r): List [r]
}
```

The **with** SimpleList [t] clause at the beginning of the List record simply inserts the definitions for those methods of SimpleList which are not defined later in the List record. In our case, because tail is redefined later, the cause would be equivalent to the two definitions:

```
  def isEmpty: Bool
  def head: a
```

The effect of **with** is analogous to *inheritance* in object-oriented programming, and the redefinition of tail is analogous to *method overriding*.

Constructors for List are also defined in terms of the constructors for SimpleList. Each constructor in List simply wraps the corresponding constructor in SimpleList in an application of function fromSimple. That function constructs a record which inherits the given SimpleList parameter, redefines function tail and adds the newly defined methods.

```
val List = {
  def fromSimple [t] (xs: SimpleList [t]): List [t] = {
    with xs

    def tail = fromSimple (xs.tail)

    def length = if (isEmpty) 0 else 1 + tail.length

    def append (ys) =
      if (isEmpty) ys else List.Cons (head, tail.append (ys))

    def forall (p) =
      p (head) && tail.forall (p)

    def filter (p) =
      if (isEmpty) List.Nil
      else if (p (x)) List.Cons (head, tail.filter (p))
      else tail.filter (p)

    def map (f) =
      if (isEmpty) List.Nil else List.Cons (f (head), tail.map (f))
  }
```

```
    def Nil [t]: List [t] =
        fromSimple (SimpleList.Nil)

    def Cons [t] (x: t, xs: List [t]): List [t] =
        fromSimple (SimpleList.Cons (x, xs))
}
```

The final type for lists is defined in terms of a basic implementation with just the operations isEmpty, head, and tail. Actually, it is possible to define lists with even fewer basic methods. It would be sufficient to have a single method match which takes a *visitor* [12] record as argument. Depending on whether the list was empty or not, one of two methods of the visitor would be invoked. This structure essentially models the *algebraic data types* found in many functional languages, with the visitor taking over the role of a pattern matching case expression. Details are found in [21]

### Higher-Order Functions

To keep the presentation short, we have restricted ourselves in the definition of the List data structure to a representative sample of methods. Besides length and append there are three other methods which all take functions as parameters:

- The method forall takes a predicate and returns true iff all elements of the list satisfy the predicate. The predicate is a function from the element type of the list t to Bool. Function types are expressed with an infix arrow, e.g. the given predicate has type t $\rightarrow$ Bool.
- The method filter also takes a predicate and returns a list of all those elements which satisfy the predicate.
- The method map takes a function f from list elements to some arbitrary result type r and returns the list resulting from applying f to each element.

The standard definition of List in Funnel defines these methods and several others as well. Functions such as forall, filter, and map, which take functions as arguments or return them as results are called *higher-order* functions. To streamline the application of such functions we introduce a lightweight form of functional abstraction, which lets us define anonymous functions: the expression $(x_1, ..., x_n \mid E)$ denotes a function which maps its formal argument $x_1, ..., x_n$ to the value computed by the expression E (which may refer to $x_1, ..., x_n$). For instance, the following expression would yield a list consisting of the squares of elements of a given list:

```
xs.map (x | x * x)
```

To demonstrate the expressive power and convenience of higher-order functions we conclude this section with a presentation of an algorithm for topologically sorting a graph. For a given Node type, we define the type Edge as a pair of two nodes, and the type Graph as a record of nodes and edges:

```
type Node = ...
type Edge = (Node, Node)
type Graph = {
  def nodes: List [Node]
  def edges: List [Edge]
}
```

A topological sort function maps a given graph to a list of all nodes in the graph, such that the predecessor of every edge appears in the list before its successor. This is possible only if the graph has no cycles; for cyclic graphs an error should be reported. An implementation of topSort is as follows.

```
def topSort (g: Graph): List [Node] = {
  def isSource (n: Node) = edges.forall (x, y | n != y)
  val sources = g.nodes.filter (isSource)
  if (sources.isEmpty)
    error ("topSort of cyclic graph")
  else
    sources.append (topSort {
      def nodes = g.nodes.filter (x | ! isSource (x))
      def edges = g.edges.filter (x, y | ! isSource (x))
    })
}
```

The implementation of this function is very close to the following high-level constructive definition of topological sorting: "To sort a graph, form a list consisting of all sources of the graph, followed by a sort of the remaining graph which results from removing all original sources and all edges emanating from them. A node is a source of a graph if it is not the successor of any edge in the graph." The functional definition is concise and elegant, even though it's not optimally efficient. Nevertheless, the present formulation is very attractive as either a prototype implementation or a high-level design which is to be improved in its efficiency by program transformations.

## 3 Forks and Joins

The subset of the Funnel language we have seen so far was purely functional. There was no way to express a side effect or a concurrent computation of several threads. We now extend Funnel to make it into a general purpose programming language in which these concepts can be expressed. Surprisingly little new mechanism is needed for these extensions. We will concentrate in this section on basic concurrency constructs. In the next section, we will show how how the same constructs also express imperative programming.

Any concurrent language needs an operator to initiate parallel computation. We introduce for this purpose the *fork* operator &. The term A & B executes its operands A and B in parallel.

A concurrent language also needs mechanisms for synchronizing threads running in parallel and for communications between them. A large number of different methods for doing this have been proposed. We use here the following simple scheme which fits very well with the functional approach we have followed so far. We admit the & operator not only in function bodies on the right hand side of a rewrite rule, but also in function headers on the left-hand side of a rewrite rule. For instance, the definition

**def** a (x) & b (y) = c (x + y) & d ()

would be interpreted as follows: "If there are two concurrent calls to functions a(x) and b(y), then these calls are to be replaced by two concurrent evaluations of c(x + y) and d(). As long as there is only one call to a or b this call will block until its partner in the rule is also called." An occurrence of & on the left-hand side of a definition is called a *join*, since it's effect is the joining of two previously independent threads of computation. Consequently, a left-hand side containing & symbols is also called a *join pattern*.

### Asynchronous Channels

As a first example of join synchronization, consider an implementation of an asynchronous channel, which connects a set of producers with a set of consumers. Producers call a function put to send data to the channel while consumers call a function get to retrieve data. An implementation of a channel is realized by the following simple functional net.

**def** get: T & put (x: T) = x

This definition jointly defines the two functions get and put. Only put takes a parameter, and only get returns a result. We call result-returning functions like get *synchronous* , whereas functions like put are called *asynchronous*. The definition rewrites two concurrent calls to put (x) and get to the value x, which is returned to the caller of the get operation. Generally, only the first function in a join pattern can return a result, all other functions are asynchronous. Likewise, only the first operand of a fork can return a result, all other operands are asynchronous or their result is discarded.

Here is an example of a program that uses the channel abstraction.

```
def get: Int & put (x: Int) = x
def P = { val x = get ; P & put (x + 1) }
P & put (1)
```

The program repeatedly reads out a number which was sent to a channel and sends back the number incremented by one. Initially, the number 1 is sent to the channel. A reduction of this program starts with the following steps (where we abbreviate the above definitions of get, put and P to **def** D).

|   |   |
|---|---|
|   | **def** D; P & put (1) |
| → | (by rewriting P to its definition) |
|   | **def** D; { **val** x = get ; P & put (x + 1) } & put (1) |
| → | (by joining put (1) and get) |
|   | **def** D; **val** x = 1 ; P & put (x + 1) |
| → | (by expanding the val) |
|   | **def** D; P & put (1 + 1) |
| → | (by numeric simplification) |
|   | **def** D; P & put (2) |
| → | (by rewriting P to its definition) |
|   | **def** D; { **val** x = get ; P & put (x + 1) } & put (2) |
| → ... |   |

This program is still deterministic, in the sense that at each point only one reduction is possible. But we can change this by adding another process Q:

```
def get: Int & put (x: Int) = x
def P = { val x = get ; P & put (x + 1) }
def Q = { val x = get ; P & put (x – 1) }
P & Q & put (1)
```

There are now many possible reduction sequences of this program, since either $P$ or $Q$, but not both, can react via their get operation with a previous put operation. The sequence in which the two processes execute is arbitrary, controlled only by the channel's rewrite rules. Generally, we do not assume *fairness*, so that it would be possible for P to execute infinitely often without any intervening execution of Q (or *vice versa*).

Channels can be turned into reusable abstractions by packing their operations in a record with a type parameterized with the type of messages:

```
type Channel [t] = {
    def get: t
    def put (x: t): nil
}
def newChannel [t]: Channel [t] = {
    def get & put (x) = x
}
```

The type of put is a function from t to **nil**, which represents the empty type.

**A note on precedence** We assume the following order of precedence, from strong to weak:

```
( )  (&)  (=)  (,)  (;)
```

That is, function application binds strongest, followed by parallel composition, followed by the equal sign, followed by comma, and finally followed by semicolon. Other standard operators such as $+$, $*$, $==$ fall between function application and & in their usual order of precedence. When precedence risks being unclear, we'll use parentheses to disambiguate.

**A Note on Indentation.** As a syntactic convenience, we allow indentation instead of ;-separators inside blocks delimited with braces { and }. Except for the significance of indentation, braces are equivalent to parentheses. The precise rules are: (1) In a block delimited with braces a semicolon is inserted in front of any non-empty line which starts at the same indentation level as the first symbol following the opening brace, provided the symbol before the insertion point is not already a semicolon. The only modification to this rule is: (2) If inserted semicolons would separate two **def** blocks, yielding **def** $\mathsf{D}_1$ ; **def** $\mathsf{D}_2$ say, then the two **def** blocks are instead merged into a single block, i.e. **def** $\mathsf{D}_1$, $\mathsf{D}_2$. (3) The top level program is treated like a block delimited with braces, i.e. indentation is significant.

### Qualified Definitions

The asynchronous channel suffers from a potentially unbounded pile-up of undelivered messages if messages are produced at a higher rate than they are consumed. This problem is avoided in bounded buffers which limit the number of unconsumed messages to some given number. For instance, the following functional net implements a one-place buffer in which the number of put operations cannot exceed the number of get operations by more than one:

```
def get: T & full (x: T)      =  x  & empty,
     put (x: T): () & empty =  ()  &  full (x)
```

There are two definitions which define four functions. Functions put and get are meant to be called from the producer and consumer clients of the buffer. The other two functions, full and empty, reflect the buffer's internal state, and should be called only from within the buffer.

In contrast to the situation with asynchronous channels, both get and put are now synchronous functions. The put operation returns the empty tuple () as its result. Its result type, also written (), consists of just this single value. Functions that return () are called *procedures*. The explicit () result is important since it tells us that the call to procedure put has terminated. If nothing was ever returned, put would be an asynchronous function, the termination of which cannot be observed. The "side effecting" part of the expression is full (x). Like empty, full is an asynchronous function. It's sole purpose is to enable calls to get and to pass the stored element x along.

When faced with the task of turning one-place buffers into a reusable abstraction, we encounter some problems. First, not all functions of a one-place buffer should be turned into methods of a record – empty and full should remain hidden. Furthermore, before using a buffer abstraction we also need to initialize it by invoking the empty method. But empty is supposed to be internal, hence it cannot be initialized from outside! One possible solution would be to have a local definition of the four functions making up a buffer and then constructing a record with the get and put methods, which forward to the corresponding local method. But the code for this, presented below, is rather cumbersome.

```
def Buffer [t] = {
   def get: t
   def put (x: t): ()
}
def newBuffer [t]: Buffer [t] = {
   def get': t & full (x: t)      =  x  & empty,
        put' (x: t): () & empty  =  ()  &  full (x)
   { def get = get',
          put (x) = put' (x) } & empty
}
```

A more streamlined presentation can be obtained using *qualified definitions*. Qualified definitions mirror the qualified names used for accessing record elements. As an example, consider a re-formulation of the newBuffer function.

```
def newBuffer [t]: Buffer [t] = {
   def this.get & full x        =  x  & empty,
        this.put (x) & empty  =  () & full x
   this & empty
}
```

The left-hand side of a definition can now contain not just simple identifiers but qualified names like this.get and this.put. In the above example the local definition thus introduces the three names this, full, and empty. this represents a record with two fields, get and put, while empty and full represent functions. Note that the naming of this is arbitrary, any other name would work equally well. Note also that empty and full are not methods of the record returned from newBuffer, so that they can be accessed only internally.

The identifiers which occur before a period in a join pattern always define new record names, which are defined only in the enclosing definition. It is not possible to use this form of qualified definition to add new fields to a record defined elsewhere.

We can now explain the anonymous form of record definition as an implicit qualified definition, where every defined method is prefixed with the name of the record being defined. I.e. the newChannel function given above would be regarded as syntactic sugar of the following expanded version:

```
def newChannel [t]: Channel [t] = {
   def this.get & this.put (x) = x
   this
}
```

The new interpretation gives up structured record values as a primitive language feature, replacing it by simple names. This view is a good fit with the join calculus theory underlying functional nets, since that calculus also represents every intermediate result with a name. Details are found in [21].

A simple usage of the one-place buffer abstraction is illustrated in the following example.

```
def newBuffer [t]: Buffer [t] = {
   def this.get & full x        = x & empty,
       this.put (x) & empty = () & full x
   this & empty
}
val b: Buffer [Int] = newBuffer
b.put (1) & ( val y = b.get ; b.put (y + y) )
```

There are two client processes composed in parallel. One process puts the number 1 into the buffer. The other process tries to get the buffer's contents and once this succeeds, puts a modified value back into the buffer. We will now trace the rewrite steps of this program. As a first step, the newBuffer call is replaced by its definition, yielding

```
def newBuffer [t] = ...
val b: Buffer [Int] = {
   def this.get & full x        =  x  & empty,
       this.put (x) & empty  =  () & full x
   this & empty
}
b.put (1) & ( val y = b.get ; b.put (y + y) )
```

The complex right-hand side of the **val** definition can be simplified by "pulling out" all elements which do not form part of the result (Precise rules for such permissible rearrangements of terms can be found in [21]). This yields:

```
def newBuffer [t] = ...
def this.get & full x        =  x  & empty,
    this.put (x) & empty  =  () & full x
val b: Buffer [Int] = this
b.put (1) & ( val y = b.get ; b.put (y + y) )  & empty
```

A simple-minded rearrangement of definitions from a local to a global scope can possibly create name clashes or shadow names that were visible before. We avoid these pitfalls by systematic renaming of the locally defined names where necessary. In our example, a renaming is not necessary, but let's do it anyway to demonstrate the principle. The following program is identical to the previous one, except that this has been renamed to B.

```
def newBuffer [t] = ...
def B.get & full x        =  x  & empty,
    B.put (x) & empty  =  () & full x
val b: Buffer [Int] = B
b.put (1) & ( val y = b.get ; b.put (y + y) )  & empty
```

The next rewrite step expands out the value definition, yielding:

```
def newBuffer [t] = ...
def B.get & full x        =  x  & empty,
    B.put (x) & empty  =  () & full x
B.put (1) & ( val y = B.get ; B.put (y + y) )  & empty
```

Abbreviating the two initial **def** blocks to **def** D, we finish with the following rewrite sequence:

**def** D ; B.put (1) & ( **val** y = B.get ; B.put (y + y) ) & empty

→ (by joining B.put (1) and empty)

**def** D ; () & B.full (1) & ( **val** y = B.get ; B.put (y + y) )

→ (by joining B.full (1) and B.get)

**def** D ; () & B.empty & ( **val** y = 1 ; B.put (y + y) )

→ (by expanding the **val** definition)

**def** D ; () & B.empty & B.put (2)

→ (by joining B.empty and B.put (2))

**def** D ; () & () & B.full (2)

→ (by discarding the second () result)

**def** D ; () & B.full (2)

# 4 Mutable State

The mechanisms introduced so far are also sufficient to explain imperative programming, where programs manipulate a state of mutable variables. After all, a mutable variable can be easily modeled as a functional net with the following two defining equations.

```
def value: T & state (x: T)         =  x  & state x,
    update (y: T) & state (x: T)    =  () & state y
```

The structure of these equations is similar to the one-place buffer in Section 3. The two synchronous functions value and update access and update the variable's current value, which is carried along as a parameter to the asynchronous function state.

The function state in the above definition appears on the left-hand sides of both equations. There is nothing strange about this. Remember that equations are rewrite rules. It is certainly possible for a function symbol to appear on the left-hand side of more than one rewrite rule, as long as all such rules form part of the same **def** block. On the other hand, it is not allowed that the same function appears several times in a single join pattern, i.e. **def** state x & state y = ... would be illegal.

Building on these definitions, we can create an abstraction for first-class mutable variables (also called a *reference cells*). The abstraction defines a type Ref [t] for reference cells and a function newRef for creating them. Here are the details.

```
type Ref [t] = {
    def value: t
    def update (y: t): ()
}
def newRef [t] (initial: t) = {
    def this.value & state (x: t) = x & state (x),
        this.update (y: t) & state (x: t) = () & state (y)
    this & state (initial)
}
```

As a simple usage example consider the following program which creates a reference and defines and applies an increment function for it:

```
val r: Ref [Int] = newRef (0)
def increment  =  r.update (r.value + 1)
increment
```

**Variables**

The last program shows that while it is possible to use references, it is certainly not very pleasant. To obtain a more conventional and streamlined notation we introduce the following syntactic abbreviations:

- The definition **var** x := E expands to the three definitions

  ```
  val _x = newRef (E)
  def x = _x.value
  def x! = _x.update
  ```

  The first of these creates a reference cell while the second and third definitions introduce an *accessor* function x and an *update* function x!. Only the last two names are visible whereas the name of the reference cell _x is meant to be inaccessible for user programs.

- Analogously, a signature **var** x: T in a record type expands to a pair of signatures for the accessor and update function.

  ```
  def x: T, x!: T→()
  ```

- Simple assignment expressions x := E expand to simple calls of the update function, e.g. x!(E).

- Qualified assignment expressions Q.x := E expand to qualified calls of the update function, e.g. Q.x!(E).

With these expansions one can write programs using a customary syntax for variable definition and access. For instance, the previous increment program can now be written more concisely as follows.

```
var count := 0
def increment  =  count := count + 1
increment
```

**Loops**

The second important aspect of imperative programming are its loop control structures. In principle, such structures can be written as higher-order functions, thus implementing directly their standard definitions. For instance, here is an implementation of a function for while loops:

```
def whileLoop (cond: ()→Bool) (body: ()→()) =
   if (cond ()) { body () ; whileLoop (cond) (body) } else ()
```

And here is a usage example, which employs two anonymous functions for the condition and the body of the loop.

```
def exp (x: Int, y: Int) =  {
   var m := 1
   var n := y
   whileLoop (| n > 0) (| m := m ∗ x ; n := n – 1 )
   m
}
```

The example of Smalltalk blocks [14] has shown that it is quite possible to base all control structures on higher-order functions. But in the interest of readability we will retain the usual forms of control structures, with their meaning established by expansions like the one above.

**Implementation Aspects**

Our presentation so far has shown that imperative programs can be seen as a special class of functional nets, and that, at least in principle, no new constructs are necessary to support them. So far, we have not yet touched on the issue of efficiency. Clearly, a literal implementation of mutable variables in terms of joins and forks would be much less efficient than a direct implementation in terms of memory cells. But nobody forces us to actually use the literal implementation. One can equally well choose an implementation of the abstract Ref type in terms of primitive memory cells. Such an implementation could no longer be represented as a functional net, but its high-level interface, as well as the programs using it, could.

Another source of inefficiency lies in the indirection entailed by reference cell objects. In many cases, it is possible to avoid the indirection by "inlining" variables at the point where they are defined. This is particularly attractive for our high-level **var** syntax since this syntax does not permit access to the variable as a first class reference, only read and write operations are permitted. Nevertheless, care must be taken because functions accessing variables can escape the environment where the variable is defined. Example:

```
def newCounter = {
   var count := 0
   def this.reset  =  count := 0,
       this.increment  =  count := count + 1,
       this.value =  count
   this
}
```

In this example, the count variable may not be stored in newCounter's stack activation record, since the record returned from newCounter still accesses count. The standard object representation of reference cells avoids the problem since we assume that objects are always stored on the heap. There exist good techniques for *escape analysis* which tell whether a variable can be stored on the stack or whether it must be stored on the heap.

### Functions and State

We now present an example which demonstrates how imperative and functional code can be combined with and substituted for each other. A often-used function over lists is reduce, which uses a given binary operator $\oplus$ and a zero value z to reduce all elements of a list to a single value:

$$\mathsf{reduce}\ (\oplus,\ \mathsf{z},\ (\mathsf{x}_1,\ ...,\ \mathsf{x}_n))\ =\ (\ ...\ (\mathsf{z} \oplus \mathsf{x}_1) \oplus ...\ ) \oplus \mathsf{x}_n\ .$$

Here's first a purely functional implementation of reduce:

```
def reduce [s,t] (f: (s,t)→s, z: s, xs: List [t]): s  =
   if (xs.isEmpty) z
   else reduce (f, f (z, xs.head), xs.tail)
```

An equivalent imperative implementation is:

```
def reduce [s,t] (f: (s,t)→s, z: s, xs: List [t]): s  =  {
   var acc := z
   var rest := xs
   while (! rest.isEmpty) {
      acc := f (acc, rest.head)
      rest := rest.tail
   }
   acc
}
```

Once defined, reduce can be used in a large number of programming tasks. Examples:

- To sum up a list of operands:

```
def plus (x, y)  =  x + y;
def sum xs  =  reduce (plus, 0, xs)
```

- To compute the product of a list of factors:

```
def times (x, y)  =  x * y;
def product xs  =  reduce (times, 1, xs)
```

- To reverse a list:

```
def snoc (xs, x)  =  List.Cons (x, xs);
def reverse xs  =  reduce (snoc, List.Nil, xs)
```

- To form the scalar product of two vectors represented as lists:

```
def scalprod (xs, ys)  =  reduce (plus, 0, map (times, zip (xs, ys)))
```

  Here, zip is a function which takes two lists and forms a list of pairs which combine corresponding elements of the two lists. This function can be defined as follows.

```
def zip [s, t] (xs: List [s], ys: List [t]): List [(s, t)] =
   if (xs.isEmpty || ys.isEmpty) List.Nil
   else List.Cons ((xs.head, ys.head), (xs.tail, ys.tail))
```

**Stateful Objects**

Objects with state arise naturally out of a combination of functional objects and variables. The newCounter function which was presented earlier provides an example. On the other hand, we can also implement stateful objects in a more direct way, by extending the implementation scheme for reference cells. An object with methods $m_1$, ..., $m_k$, and instance variables $x_1$, ..., $x_l$ with initial values $i_1$, ..., $i_l$ can be coded as follows.

$$
\begin{array}{rcl}
\textbf{def}\ \mathsf{this.m}_1\ \&\ \mathsf{state}\ (\mathsf{x}_1,\ ...,\ \mathsf{x}_l) & = & ...\ ;\ \mathsf{state}\ (\mathsf{x}_1^1,\ ...,\ \mathsf{x}_l^1), \\
& \vdots & \\
\mathsf{this.m}_k\ \&\ \mathsf{state}\ (\mathsf{x}_1,\ ...,\ \mathsf{x}_l) & = & ...\ ;\ \mathsf{state}\ (\mathsf{x}_1^k,\ ...,\ \mathsf{x}_l^k); \\
\mathsf{this}\ \&\ \mathsf{state}\ (\mathsf{i}_1,\ ...,\ \mathsf{i}_l) & &
\end{array}
$$

Here, the dots after the equal signs stand for elided method bodies. The object's instance variables are represented as arguments of a state function, which is initially applied to $i_1$, ..., $i_l$. Each method $m_j$ consumes the current state vector of all instance variables and terminates by installing a new state in which instance variables have new values $x_1^j$, ..., $x_l^j$. Here is a re-formulation of counter objects using the new scheme.

```
def newCounter = {
   def this.reset & state (count: Int)  =  state (0),
       this.increment & state (count: Int)  =  state (count + 1),
       this.value  =  count & state (count)
   this
}
```

This representation technique achieves a common notion of information hiding, in that an object's instance variables cannot be accessed from outside the object. It further achieves mutual exclusion in that only one method of a given object can execute at any one time. After all, each method starts by consuming the state vector which is not re-established before the end of the method's execution. Thus, after one method commences all other method calls to the same object are blocked until the first method has terminated. In other words, objects implemented in this way are *monitors*.

## 5 Concurrency

We now show how common constructs that control concurrent execution can be expressed as functional nets. The main problem in handling concurrency has to do with *process synchronization*. How can one process wait until some condition is established by some other process(es)? How can we ensure that certain operations are *atomic*, that is that they are performed from start to finish by one process at any one time, without other processes starting execution of the same action? In the following, we'll present several well-known schemes to handle this task and show how they can each be expressed as functional nets. For reasons of bounded space, we can only present a selection of common constructs. We refer the reader to [21] for a description of further constructs.

### Semaphores

A common mechanism for process synchronization is a *lock* (or: *semaphore*). A lock offers two atomic actions: getLock and releaseLock. The lock starts its life in released state. A call to getLock succeeds and returns () if the lock is in released state. At the same time, the lock is put into locked state, in which other calls to getLock block until the lock is put back into released state by an operation of releaseLock. Here's the implementation of a semaphore as a functional net:

```
type Lock = {
   def getLock: ()
   def releaseLock: nil
}
```

```
def newLock = {
    def this.getLock & this.releaseLock  =  ()
    this & this.releaseLock
}
```

A typical usage of a semaphore would be:

```
val s = newLock ;
...
s.getLock ; "< critical region >" ; s.releaseLock
```

It is interesting to note the similarities in the definitions of semaphores and asynchronous channels.

### Critical Regions

One problem with the previous usage pattern of semaphores is that it is easy to forget the releaseLock call, thus locking the semaphore forever. A safer solution is to pass the critical region into a higher order function which does both the getLock and releaseLock:

```
type Proc = () → ()
type Mutex = Proc → ()
def newMutex: Mutex = {
    val s = newLock
    ( region: Proc | s.getLock ; region () ; s.releaseLock )
}
```

A typical usage pattern for this would be:

```
val mutex = newMutex
...
mutex (| "< critical region >")
```

### Monitors

Modula-2 has popularized *monitors* [16] as a synchronization method. A monitor exports a set of functions $\mathsf{f}_1$, ..., $\mathsf{f}_k$ while ensuring that only one of these functions can be active at any one time. At the end of Section 4 we have already seen a scheme to implement monitors by joining methods with a state vector containing all instance variables. Alternatively, we can represent instance variables as normal variables in a record's environment and use a turn "token" to achieve mutual exclusion. Here is an implementation of counters using that scheme.

```
def newSyncCounter = {
    var count := 0
    def this.reset & turn  =  { count := 0 ; () & turn },
         this.increment & turn  =  { count := count + 1 ; () & turn },
         this.value & turn  =  { val x = count ; x & turn }
    this & turn
}
```

Many monitors are more involved than this example in that a process executing in a monitor also needs to wait for conditions to be established by other processes. We can use join patterns to implement this form of waiting as well as mutual exclusion. As an example, consider the following implementation of bounded buffers.

```
def newBoundedBuffer [t] (N: Int): Buffer [t] = {
    val elems: Array [t] = newArray [t] (N)
    var in := 0; var out := 0

    def this.put (elem: t) & available & turn  =  {
        elems (in) := elem ; in := (in + 1) % N
        () & filled & turn
    }
    def this.get & filled & turn =  {
        val elem = elems (out) ; out := (out + 1) % N
        elem & available & turn
    }
    var i := 0
    while (i < N) { () & available ; i := i + 1 }
    this & turn
}
```

The basic implementation of the buffer is as in [28]. Buffer elements are stored in an array elems with two variables in and out keeping track of the first free and occupied position in the array. These variables are incremented modulo N, the size of the buffer. What's new here is how processes wait for their preconditions. A producer process has to wait for the buffer to become nonfull, while a consumer process has to wait for the buffer to become nonempty. Our implementation mirrors each filled slot in the buffer with a filled token. Each unfilled slot in the buffer is mirrored with an available token. Thus, the total number of filled and available tokens always equals N, the number of elements in the buffer. Initially, all slots are free, so we create N avaliable tokens. Each subsequent put operation removes an available token and creates a filled token, while each get operation does the opposite; it removes a filled token and creates an available token. Mutual exclusion of the get and put operations is ensured by the standard turn token technique.

**Signals**

Instead of join patterns, Modula-2 uses *signals* to let processes executing in monitor wait for preconditions. The implementation of such signals as functional nets is a bit subtle. On the surface, a signal is quite similar to a semaphore. Like a semaphore, a signal also offers two operations, which are now called send and wait. The wait operation of a signal corresponds to the getLock operation of a semaphore in that it waits until there is a call to send and then returns. A send operation of a signal on the other hand will unblock a pre-existing wait but does nothing at all if no call to wait exists. Unlike the releaseLock operation of a semaphore, a send will not interact with a wait that is issued later. Hence, in the execution sequence releaseLock ; getLock the getLock operation will always succeed, whereas in the sequence send ; wait the wait will always block since the previous send got lost. A first attempt to implement signals as functional nets could be:

```
type Signal = {
   def wait: ()
   def send: nil
}
def newSignal: Signal = {
   def this.wait & this.send  =  (),
       this.send              =  nil
   this
}
```

The hope is that if there are calls to both send and wait, the wait operation will return with () whereas a send alone will reduce to process **nil**, which does nothing at all.

This implementation of signals would work if join patterns were always tried in sequence, with earlier join patterns taking precedence over later ones. Unfortunately, there's nothing that stops an implementation of functional nets from always immediately reducing a send operation to **nil**, such that a wait might block even after subsequent send's. Hence, our first attempt to implement signals fails.

It is still possible to formulate signals as functional nets. Here's one way to do it:

```
def newSignal: Signal = {
   def this.send & inactive  =  inactive,
       this.wait & inactive  =  wait1,
       wait1 & this.send     =  () & inactive
   this & inactive
}
```

The solution makes use of a token, inactive, which signals that there are no active waits. A send in inactive state simply leaves the state inactive. A wait in inactive state suppresses inactive state and continues with the blocking operation wait1. Finally, a wait1 and a send return () to the wait1 and re-establish inactive state.

The second implementation of signals "feels" more deterministic than the first one, but its correctness is not easy to establish. Indeed, if one compares actual reduction sequences one finds that every reduction sequence of send's and wait's possible in the first implementation is also possible in the second and *vice versa*. The problem is that even with the second formulation of signals a processor might never reduce an incoming wait to a wait1. For instance, it might be busy doing other things indefinitely. Therefore, send's might still overtake wait's and get lost. So have we gained nothing by refining the implementation of signals? Not quite. The second formulation still has the following property, which is generally good enough for reactive systems: Say some process in a reactive system issues a wait on some signal and at some later time all processes of that system block. If at some later time some process is triggered by an external event and, once resumed, issues a send on the same signal then the wait (or some other wait on the same signal) is guaranteed to be able to proceed.

We now reformulate the bounded buffer example to closely match the implementation in the Modula-2 book, making use of our signal abstraction. This implementation provides an explanation of the interaction between monitors and signals, which is also quite subtle. Here is the code:

```
def newBoundedBuffer [t] (N: Int): Buffer [t] = {
    val elems: Array [t] = newArray [t] (N)
    var in := 0; var out := 0; var n := 0

    val nonFull: Signal = newSignal
    val nonEmpty: Signal = newSignal

    def this.put (elem: t) & turn  =  {
        n := n + 1
        while (n > N) { waitUntil (nonFull) }
        elems (in) := elem ; in := (in + 1) % N
        if (n ≤ 0) nonEmpty.send
        () & turn
    }
    def this.get & turn  =  {
        n := n − 1
        while (n < 0) { waitUntil (nonEmpty) }
        val elem = elems (out) ; out := (out + 1) % N
        if (n ≥ N) nonFull.send
        elem & turn
    }
```

```
    def waitUntil s = ( s.wait & turn ; reEnter )
    def reEnter & turn = ()

    this & turn
}
```

The correct interaction of signals with monitors, hidden in Modula-2 by the underlying implementation, requires some effort to model explicitly. When waiting for a signal, a process has to re-issue the turn token, to let other processes enter the monitor. On the other hand, once a process wakes up, it has to re-consume the turn token to make sure that it executes under mutual exclusion. The re-issue and re-consumption of the turn token is done in functions waitUntil and reEnter. Because a process might block waiting for its turn after being woken up by a send, it is not guaranteed that the awaited condition will still be established when the process finally resumes. Hence, the awaited condition needs to be tested repeatedly in a **while** loop. This style of monitors has been discussed together with other alternatives by Hoare [16].

The second implementation of bounded buffers is more complicated than the first. But one might still prefer it because of efficiency considerations. After all, the implementation in terms of signals uses fewer tokens than the first implementation, because it sends a signal only when a processes is waiting for it. This does not invalidate our initial implementation, however, which remains attractive as a concise and executable design.

## 6 Conclusion

Functional nets are an attempt to distill a large number of programming concepts into a uniform and very simple kernel. We have presented here a tour through functional, object-oriented, imperative and concurrent programming concepts, which were all mapped into our kernel language Funnel. A companion paper [21] takes this approach further, by translating Funnel itself to object-based join calculus, a simple operational semantics which can be described comfortably on a single page. The reduction of a large number of phenomena to a common foundation is inherent to a large part of science. Like in other areas, it is also useful in programming since it helps clarify the meaning of derived constructs and the interactions between them.

An analogous approach is well established in the field of functional programming, where elaborate functional programming languages are reduced to $\lambda$-calculus as a kernel language. Classical $\lambda$-calculus [8, 5] is ideally suited as a basis for functional programs, but less well suited for imperative and reactive programs. Extensions of $\lambda$-calculus for modeling these aspects have been devised, but these extension quickly become too complex to seem completely natural. Join calculus, on the other hand, supports all these aspects in the kernel language itself. Consequently, it is possible to express a much wider variety of programming constructs than with $\lambda$-calculus alone.

The introduction given here remains necessarily sketchy. Issues which have not been addressed include a formal language specification of Funnel including its type system, a program logic, programming methodology and laws of programming, implementation and optimization techniques, and the modeling of more involved programming concepts such as classes and inheritance.

# References


[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1984.

[3] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts – a tour of Piccola. In *Software Architectures and Component Technology*. Kluwer, 2000.

[4] John W. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21:613–641, 1978.

[5] Hendrik P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[6] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998. 2nd edition.

[7] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[8] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.

[9] Fabrice Le Fessant. *The JoCaml reference manual*. INRIA Rocquencourt, 1998. Available from `http://join.inria.fr`.

[10] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, January 1996.

[11] Cédric Fournet and Luc Maranget. *The Join-Calculus Language*. INRIA Rocquencourt, 1997. Available from `http://join.inria.fr`.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[13] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[14] Adele Goldberg and David Robson. *Smalltalk-80; The Language and Its Implementation*. Addison-Wesley, 1983.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[16] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 12(10), October 74.

[17] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[18] Jon Kerridge. *Occam programming: a practical approach*. Blackwell Scientific, 1987.

[19] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[21] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, 2000.

[22] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN 0-521-39022-2.

[23] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[24] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, 1998.

[25] The Munich Project. *CIP, Volume 1: The Wide Spectrum Language CIP/L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[26] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[27] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 293–305, June 1991.

[28] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 2nd edition, 1983.