An Overview of Functional Nets

Martin Odersky
École Polytechnique Fédérale de Lausanne

August 16, 2000

Abstract

Functional nets combine key ideas of functional programming and Petri nets to yield a simple and general programming notation. They have their theoretical foundation in Join calculus. This paper gives an overview of functional nets as a kernel programming language, it presents an object-based version of Join calculus, and it shows how the two relate.

Over the last decades 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 [Bac78, Hud89], in modern theories of objects [AC96], as well as in concurrent systems based on message passing such as CSP [Hoa85], CCS [Mil80] or π-calculus [MPW92].

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 [ASS84], ML [Pau91], or Haskell [Bir98], based on λ-calculus, as well as concurrent languages such as Occam [Ker87], based on CSP, or Pict [PT97] and Piccola [AN00], which are based on π-calculus.

These notes give an introduction to *functional nets* which support functional, object-oriented, and concurrent programming in a simple calculus based on rewriting. Functional nets arise out of a combination of key ideas of functional programming and Petri nets [Rei85]. 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.

Functional nets have a theoretical foundation in *join calculus* [FG96, FGL$^+$96]. They have the same relation to join calculus as classical functional programming has to λ-calculus. That is, functional nets constitute a programming method which derives much of its simplicity and

[1]Lecture Notes, to be given at the APPSEM Summer School,Caminha,Portugal,September 2000.

elegance from close connections to a fundamental underlying calculus. λ-calculus [Chu51, Bar84] is ideally suited as a basis for functional programs, but it can support mutable state only indirectly, and nondeterminism and concurrency not at all. The pair of join calculus and functional nets has much broader applicability – functional, imperative and concurrent program constructions are supported with equal ease.

The purpose of these notes is two-fold. First, they aim to promote functional nets as an interesting programming method of wide applicability. We present a sequence of examples which show how functional nets can concisely model key constructs of functional, imperative, and concurrent programming, and how they often lead to better solutions to programming problems than conventional methods.

Second, the notes develop concepts to link our programming notation of functional nets with the underlying calculus. To scale up from a calculus to a programming language, it is essential to have a means of aggregating functions and data. We introduce *qualified definitions* as a new syntactic construct for aggregation. In the context of functional nets, qualified definitions provide more flexible control over visibility and initialization than the more conventional record- or object-constructors. They are also an excellent fit to the underlying join calculus, since they maintain the convention that every value has a name. We will present object-based join calculus, an extension of join calculus with qualified definitions. This extension comes at surprisingly low cost, in the sense that the calculus needs to be changed only minimally and all concepts carry over unchanged. By contrast, conventional record constructors would create anonymous values, which would be at odds with the name-passing nature of join.

The notation for writing examples of functional nets is derived from Funnel[AOS$^+$00], a small language which maps directly into our object-based extension of join. An implementation of Funnel is publicly available. There are also other languages which are based in some form on join calculus, and which express the constructs of functional nets in a different way, e.g. Join[FM97] or JoCaml[Fes98]. We have chosen to develop and present a new notation since we wanted to support both functions and objects in a way which was as simple as possible.

The notation used here is statically typed. Its type theoretic foundation can be developed along the lines of system F_2 with subtyping [CMMS94]. *Local type inference* [PT98, OZZ00] is used to reduce the amount of type annotations which needs to be given explicitly. Details of the type system are not covered in these notes.

The rest of this paper is structured as follows. Section 1 introduces a purely functional subset of functional nets. Section 2 presents the full formalism which supports concurrent execution. Section 3 shows how functional nets model imperative programming. Section 4 presents a collection of process synchronization techniques implemented as functional nets. Section 5 presents object-based join calculus. Section 6 explains how the high-level language is mapped to that calculus. Section 7 discusses related work and concludes.

1 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":

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

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, one simply assumes appropriate reduction rules for built-ins. In the following, we 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.

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

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

1.1 Aggregation

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

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

This defines a new type of records containing three functions as fields, 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:

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

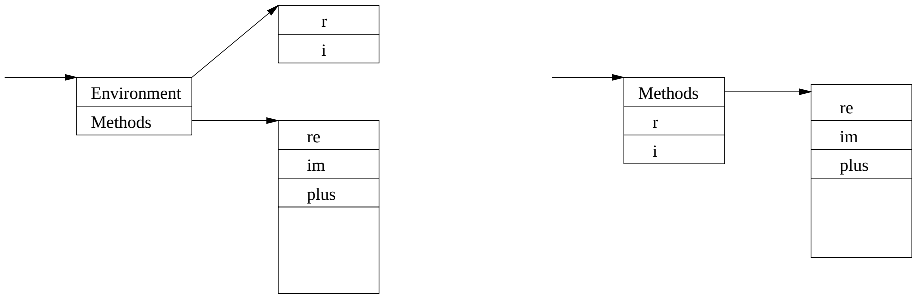

Figure 1: Object Representations

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

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

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 (1.0f, 0f)

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 $1 + i$.

1.2 Objects

So far, our notation was explained in terms of functions and records. Equally well, one 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 in many object-oriented programming languages, there is 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 was demonstrated by 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 one needs 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 makeComplex is called, one 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, the environment is passed as additional parameter. This is very similar to 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.

One 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, one would say that all instance variables are *private*[1]. Second, fields in an environment are always immutable, reflecting the purely functional nature of the language presented so far. The next sections will introduce a richer language which lets us express mutable variables. But for now we stay in the purely functional subset.

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

```
newtype SimpleList [t] = {
    def isEmpty: Boolean
    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 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")
    }
```

[1]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.

```
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.

The constructors Nil and Cons are themselves methods of a record value called SimpleList. We have chosen to wrap the Nil and Cons constructors in another record, named SimpleList. The SimpleList record value acts as a module, which provides the constructors of the list data type. Clients of the SimpleList module then construct lists using qualified names SimpleList.Nil and SimpleList.Cons. Examples:

```
def length [t] (xs: SimpleList [t]): Int =
    if (xs.isEmpty) 0
    else 1 + length (xs.tail)
def append [t] (xs: SimpleList [t], ys: SimpleList [t]): SimpleList[t] =
    if (xs.isEmpty) ys
    else SimpleList.Cons (xs.head, append (xs.tail, ys))
```

Note that Nil, Cons, length and append are *polymorphic* – they work for list's with arbitrary element types. The element type is represented in each case in the type parameter [t], which precedes any value parameters of the functions. When applying polymorphic functions, one can supply an actual instance type for the type parameter, as in

```
val l: SimpleList[Int] = SimpleList.Cons [Int] (1, SimpleList.Nil [Int])
append [Int] (l, l)
```

But it is also possible to omit the actual type parameter, if that parameter can be inferred from the types of function's value parameters or its result type. In order to avoid clutter, we will usually omit type arguments where possible.

1.3 Algebraic Datatypes

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* [GHJV94] 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. Here's the new type for lists:

```
newtype List[t] = {
    def match [r] (v: ListVisitor [t,r]): r
}
```

A visitor encodes the branches of a pattern matching case expression. It is represented as a record with one method for each branch. For instance, a visitor for lists would always have two methods, Nil and Cons:

```
newtype ListVisitor [t, r] = {
   def Nil: r
   def Cons (x: t, xs: List [t]): r
}
```

The intention is that the match method in List would call either the Nil method or the Cons method of its visitor argument, depending what kind of list was encountered. If the encountered list resulted from a Cons we also need to pass the arguments of the original Cons to the visitor's Cons.

Using match, one can write length and append over lists as follows:

```
def length [t] (xs: List [t]): Int = xs.match {
   def Nil = 0
   def Cons (x, xs1) = 1 + length (xs1)
}
def append [t] (xs: List[t], ys: List[t]): List[t] = xs.match {
   def Nil = ys
   def Cons (x, xs1) = List.Cons (x, append (xs1, ys))
}
```

More generally, every function over lists can be defined in terms of match. It only remains to implement this method. Clearly, its behavior will depend on whether it is called on an empty or non-empty list. Therefore, we define two list constructors Nil and Cons, with two different implementations for match. The implementations are straightforward:

```
val List = {
   def Nil [t]: List [t]                        = { def match v = v.Nil }
   def Cons [t] (x: t, xs: List[t]): List [t]   = { def match v = v.Cons (x, xs) }
}
```

In each case, match simply calls the appropriate method of its visitor argument v, passing any parameters along.

Note that the qualification with List in the visitor record in functions length and append lets us distinguish the constructor Cons, defined in List, from the visitor method Cons, which is defined locally.

1.4 List Methods

Actually, one could equally well define length and append as *methods* of List objects. The following example enriches the List with these and other methods.

```
newtype List [t] = {
  def match [r] (v: ListVisitor [t, r]): r

  def isEmpty: Boolean
  def head: t
  def tail: List [t]

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

The constructors List.Nil and List.Cons now define in addition to match specialized implementations of all other methods in List's type:

```
val List = {
  def Nil [t]: List [t] = {
    def match (v) = v.Nil
    def isEmpty = true
    def head = error ("Nil.head")
    def tail = error ("Nil.tail")
    def length = 0
    def append (ys) = ys
    def filter (p) = Nil
    def map (f) = Nil
  }
  def Cons [t] (x: t, xs: List[t]): List [t] = {
    def match (v) = v.Cons (x, xs)
    def isEmpty = false
    def head = x
    def tail = xs
    def length = 1 + xs.length
    def append (ys) = Cons (x, xs.append (ys))
    def filter (p) = if (p (x)) Cons (x, xs.filter (p)) else xs.filter (p)
    def map (f) = Cons (f (x), xs.map (f))
  }
}
```

1.5 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 two other methods which take functions as parameters:

- The method filter takes a predicate (i.e. a function from the list's element type to Boolean 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 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 a lightweight form of functional abstraction is introduced which lets us define anonymous functions: the expression (x_1, ..., x_n | 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]
}
```

Unlike **newtype**, which always defines a new type, the **type** definitions above define type aliases, where the left-hand side of an alias is seen as an abbreviation of its right-hand side. Only **newtype** definitions, but not aliases, may be recursive.

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) = g.edges.filter (x, y | n == y).isEmpty
    val sources = g.nodes.filter (isSource)

    if (g.nodes.isEmpty) g.nodes
    else if (sources.isEmpty) error ("topSort of cyclic graph")
    else
        sources.append (topSort {
            val nodes = g.nodes.filter (x | ! isSource (x))
            val 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.

2 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 the same constructs also support 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: Int) & b (y: Int)  =  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*.

2.1 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: **nil** = { **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: **nil** = { **val** x = get ; P & put (x + 1) }
def Q: **nil** = { **val** x = get ; Q & 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:

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

2.2 Buffers

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. By contrast, the put function of the asynchronous channel above does not return anything, and therefore its termination cannot be observed. Functions that do not return are called *asynchronous*. Their static return type is the empty type, written **nil**.

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.

2.3 Qualified Definitions

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.

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

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 can be followed by a semicolon and the next symbol can be preceded by one. The only modification to this rule is: (2) If inserted semicolons would separate two **def** blocks, yielding **def** D_1 ; **def** D_2 say, then the two **def** blocks are instead merged into a single block, i.e. **def** D_1, D_2. (3) The top level program is treated like a block delimited with braces, i.e. indentation is significant.

With these rules, the newBuffer example can alternatively be written as follows.

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

Rule (1) allows us to drop the semicolon after the local definition. With modification (2), one can use **def**'s in front of several defined functions which are conceptually part of a single definition. Which of the two styles is preferable is a matter of taste. Generally, the style with a single **def** works better for small definitions, whereas the style using indentation and multiple **def**'s tends to be easier to read if the individual definitions span many lines.

It is also possible to define several record names in a single definition. For instance, the following function would define a buffer with two aspects, one to be used by a consumer, the other to be used by a producer.

```
def newSplitBuffer [t] = {
   def outBuf.get & full (x: t)      =  x  & empty,
        inBuf.put (x: t) & empty     =  () & full (x)
   (inBuf, outBuf) & empty
}
```

The newSplitBuffer function returns two names: inBuf, which names a record with the single field put, and outBuf, which also names a record with the single field get.

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. This is further explained in Section 5.

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
def P (i: Int): nil = { b.put (i) ; P (i + 1) }
def Q (i: Int): nil = { val x = b.get ; Q(x + i) }
P (0) & Q (0)
```

Process P writes consecutive integers to the buffer b, which are read and summed up by process Q.

3 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 2. 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.

```
newtype 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
```

3.1 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
```

3.2 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 while (cond: ()→Boolean) (body: ()→()): () =
  if (cond ()) { body () ; while (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
  while (|n > 0) (| m := m ∗ x ; n := n − 1 )
  m
}
```

3.3 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.

3.4 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 foldl, which uses a given binary operator $\oplus$ and a zero value z to reduce all elements of a list to a single value:

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

Here's first a purely functional implementation of foldl:

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

An equivalent imperative implementation is:

```
def foldl [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, foldl can be used in a large number of programming tasks. Examples:

- To sum up a list of operands:

  ```
  def plus (x: Int, y: Int)  =  x + y;
  def sum (xs: List [Int])  =  foldl (plus, 0, xs)
  ```

- To compute the product of a list of factors:

  ```
  def times (x: Int, y: Int)  =  x * y;
  def product (xs: List [Int])  =  foldl (times, 1, xs)
  ```

- To reverse a list:

```
def snoc [t] (xs: List[t], x: t)  =  List.Cons (x, xs);
def reverse (xs: List[t])  =  foldl (snoc, List.Nil, xs)
```

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

  ```
  def scalprod (xs: List[Int], ys: List[Int])  =  foldl (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. It is 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))
  ```

3.5 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.

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

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 & state (count: Int)  =  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*.

3.6 Object Identity

In the object-oriented design and programming area, an object is often characterized as having "state, behavior, and identity". Our encoding of objects expresses state as a collection of applications of private asynchronous functions, and behavior as a collection of externally visible functions. But what about identity? If functional net objects had an observable identity it should be possible to define a method eq which returns true if and only if its argument is the same object as the current object. Here "sameness" has to be interpreted as "created by the same operation", structural equality is not enough. E.g., assuming that the – as yet hypothetical – eq method was added to reference objects, it should be possible to write **val** (r1, r2) = (newRef 0, newRef 0) and to have r1.eq(r1) == true and r1.eq(r2) == false.

Functional nets have no predefined operation which tests whether two names or references are the same. However, it is still possible to implement an eq method. Here's our first attempt, which still needs to be refined later.

```
newtype ObjId =
   def eq (other: ObjId): Boolean
   def testFlag: Boolean      // only for internal use
}
def newObjId: ObjId = {
   def this.eq (other: ObjId) & flag (x: Boolean) = resetFlag (other.testFlag) & flag (true)
       this.testFlag & flag (x) = x & flag (x)
       resetFlag (result: Boolean) & flag (x) = x & flag (false)
   this & flag (false)
}
```

This defines a generator function for objects with an eq method that tests for identity. The implementation of eq relies on three helper functions, flag, testFlag, and resetFlag. Between calls to the eq method, flag false is always asserted. The trick is that the eq method asserts flag true and at the same time tests whether other.flag is true. If the current object and the other object are the same, that test will yield true. On the other hand, if the current object and the other object are different, the test will yield false, provided there is not at the same time another ongoing eq operation on object other. Hence, we have arrived at a solution of our problem, provided we can prevent overlapping eq operations on the same objects. In the next section, we will develop techniques to do so.

4 Concurrency

The previous sections have shown how functional nets can express sequential programs, both in functional and in imperative style. In this section, we will show their utility in expressing common patterns of concurrent program execution.

Functional nets support an resource-based view of concurrency, where calls model resources, & expresses conjunction of resources, and a definition acts as a rewrite rule which maps input sets of resources into output sets of resources. This view is very similar to the one of Petri nets [Pet62, Rei85]. In fact, there are direct analogies between the elements of Petri nets and functional nets. This is illustrated in Figure 2.

$$\mathsf{f}_1\ \&\ \ldots\ \&\ \mathsf{f}_m \quad = \quad \mathsf{g}_1\ \&\ \ldots\ \&\ \mathsf{g}_n$$

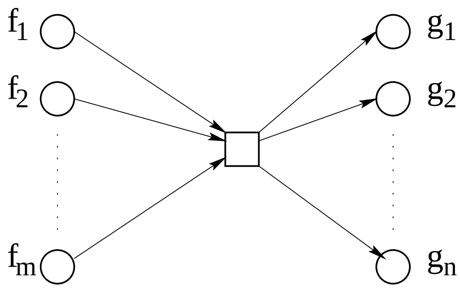

Figure 2: Analogy to Petri nets

A *transition* in a Petri net corresponds to a rewrite rule in a functional net. A *place* in a Petri net corresponds to a function symbol applied to some (formal or actual) arguments. A *token* in a Petri net corresponds to some actual call during the execution of a functional net (in analogy to Petri nets, we will also call applications of asynchronous functions *tokens*). The basic execution step of a Petri net is the firing of a transition which has as a precondition that all in-going places have tokens in them. Quite similarly, the basic execution step of a functional net is a rewriting according to some rewrite rule, which has as a precondition that all function symbols of the rule's left-hand side have matching calls.

Functional nets are considerably more powerful than conventional Petri nets, however. First, function applications in a functional net can have arguments, whereas tokens in a Petri net are unstructured. Second, functions in a functional net can be higher-order, in that they can have functions as their arguments. In Petri nets, such self-referentiality is not possible. Third, definitions in a functional net can be nested inside rewrite rules, such that evolving net topologies are possible. A Petri-net, on the other hand, has a fixed connection structure.

Colored Petri nets [Jen92] let one pass parameters along the arrows connecting places with transitions. These nets are equivalent to first-order functional nets with only global definitions. They still cannot express the higher-order and evolution aspects of functional nets. Bussi and Asperti have translated join calculus ideas into standard Petri net formalisms. Their mobile Petri nets [AB96] support first-class functions and evolution, and drop at the same time the locality restrictions of join calculus and functional nets. That is, their notation separates function name introduction from rewrite rule definition, and allows a function to be defined collectively by several unrelated definitions.

In the following, we will present several well-known schemes for process synchronization and how they each can be expressed as functional nets.

4.1 Semaphores

Semaphores A common mechanism for process synchronization is a *lock* (or: *semaphore*). A lock offers two atomic actions: getLock and releaseLock. Here's the implementation of a lock as a functional net:

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

It is interesting to note the similarities in the definitions of semaphores and asynchronous channels. A typical usage of a semaphore would be:

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

With semaphores, we can now complete our example to define objects with identity:

```
val global = newLock
def newObjId: ObjId  =  {
   def this.testEq (other: ObjId) & flag (x: Boolean) = resetFlag (other.testFlag) & flag (true)
        this.testFlag & flag (x) = x & flag (x)
        resetFlag (result: Boolean) & flag (x)  =  x & flag (false)
   def this.eq (other: ObjId) =  {
      global.getLock ; val res = this.testEq (other) ; res & global.releaseLock
   }
   this & flag (false)
}
```

This code makes use of a global lock to serialize all calls of eq methods. This is admittedly a brute force approach to mutual exclusion, which also serializes calls to eq over disjoint pairs of objects. A more refined locking strategy is hard to come by, however. Conceptually, a critical region consists of a pair of objects which both have to be locked. A naive approach would lock first one object, then the other. But this would carry the risk of deadlocks, when two concurrent eq operations involve the same objects, but in different order.

4.2 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 >")
```

4.3 Synchronous Channels

A potential problem with asynchronous channels introduced in Section 2.1 is that the producer might produce data much more rapidly than the consumer consumes them. In this case, the number of pending write operations might increase indefinitely, or until memory is exhausted. The problem can be avoided by connecting producer and consumer with a synchronous channel.

In a synchronous channel, both reads and writes return and each operation blocks until the other operation is called. Synchronous channels are the fundamental communication primitive of classical π-calculus[MPW92]. They can be represented as functional nets as follows.

```
def newSyncChannel [t] = {
   def this.read & noReads          =  read1 & read2,
       this.write (x: t) & noWrites =  write1 & write2 x,
       read1 & write2 (x: t)        =  x & noWrites,
       write1 & read2               =  () & noReads
   this & noReads & noWrites
}
```

This implementation is more involved than the one for asynchronous channels. The added complexity stems from the fact that a synchronous channel connects two synchronous operations, yet in each join pattern there can be only one function that returns. Our solution is similar to a double handshake protocol. It splits up read and write into two sub-operations each, read1, read2 and write1, write2. The sub-operations are then matched in two join patterns, in opposite senses. In one pattern it is the read sub-operation which returns whereas in the second one it is the write sub-operation. The noReads and noWrites tokens are necessary for serializing reads and writes, so that a second write operation can only start after the previous read operation is finished and vice versa. With synchronous channels, our producer/consumer example can be sketched as follows.

```
val c = newSyncChannel [Int]
def producer = {
   var x := 1
   while (|true) {| c.write x  ;  x := x + 1 }
}
def consumer = {
   while (|true) {| val y = c.read  ;  print y }
}
producer & consumer
```

This looks very similar to our previous formulation. But now the while loop in producer can proceed only once the synchronous function c.write returns, which happens only once the written datum has been read in consumer. Hence, there is never more than a single pending write operation, and all values are printed in sequence.

4.4 Monitors

Another scheme for process communication is to use a common store made up of mutable variables, and to use mutual exclusion mechanisms to prevent multiple processes from updating the same variable at the same time. A simple mutual exclusion mechanism is the *monitor* [Han72, Hoa74]. A monitor exports a set of functions f_1, ..., f_k while ensuring that only one of these functions can be active at any one time. At the end of Section 3 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 [Wir83]. 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.

4.5 Signals

Instead of join patterns, languages such as Modula-2 or Java use *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:

```
newtype 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 conventional implementations, 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
    () & turn & (if (n ≤ 0) nonEmpty.send)
  }

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

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

    this & turn
}
```

The correct interaction of signals with monitors, hidden in languages which define Monitors as primitive constructs, 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 [Hoa74].

Monitor languages often permit a process executing in a monitor to re-enter the monitor by calling one of its methods. This behavior is not modelled in our implementation. If a monitor method called itself recursively, or called another method of the same monitor, the call would block indefinitely waiting for a turn token. In our modelization such situations would have to be addressed explicitly, for instance by providing a second version of the method which was not guarded by a turn token.

The last 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.

4.6 Readers and Writers

A more complex form of synchronization distinguishes between *readers* which access a common resource without modifying it and *writers* which can both access and modify it. To synchronize readers and writers we need to implement operations startRead, startWrite, endRead, endWrite, such that:

- there can be multiple concurrent readers,
- there can only be one writer at one time,
- pending write requests have priority over pending read requests, but don't preempt ongoing read operations.

This form of access control is common in databases. It can be implemented using traditional synchronization mechanisms such as semaphores, but this is far from trivial. We arrive at a functional net solution to the problem in two steps.

The initial solution is given at the top of Figure 3. We make use of two auxiliary tokens. The token readers n keeps track in its argument n of the number of *ongoing* reads, while writers n

Initial solution:

```
    def this.startRead   & writers 0  =  startRead1,
            startRead1   & readers n  =  () & writers 0 & readers (n+1),
        this.startWrite  & writers n  =  startWrite1 & writers (n+1),
            startWrite1  & readers 0  =  (),

        this.endRead     & readers n  =  readers (n–1),
        this.endWrite    & writers n  =  writers (n–1) & readers 0

    this & readers 0 & writers 0
}
```

After factorization:

```
def newReadersWriters = {
    def this.startRead  & noWriters  =  startRead1,
            startRead1  & noReaders  =  () & noWriters & readers 1,
            startRead1  & readers n  =  () & noWriters & readers (n+1),

        this.startWrite & noWriters  =  startWrite1 & writers 1,
        this.startWrite & writers n  =  startWrite1 & writers (n+1),
            startWrite1 & noReaders  =  (),

        this.endRead    & readers n  =  if (n == 1) noReaders
                                        else readers (n–1),
        this.endWrite   & writers n  =  noReaders &
                                        if (n == 1) noWriters
                                        else writers (n–1)

    this & noReaders & noWriters
}
```

Figure 3: Readers/writers synchronization

keeps track in n of the number of *pending* writes. A startRead operation requires that there are no pending writes to proceed, i.e. writers 0 must be asserted. In that case, startRead continues with startRead1, which reasserts writers 0, increments the number of ongoing readers, and returns to its caller. By contrast, a startWrite operation immediately increments the number of pending writes. It then continues with startWrite1, which waits for the number of readers to be 0 and then returns. Note the almost-symmetry between startRead and startWrite, where the different order of actions reflects the different priorities of readers and writers.

This solution is simple enough to trust its correctness. But the present formulation is not yet valid Funnel because we have made use of numeric arguments in join patterns. For instance readers 0 expresses the condition that the number of readers is zero. We arrive at an equivalent formulation in Funnel through factorization. A function such as readers which represents a condition is split into several sub-functions which together partition the condition into its cases of interest. In our case we should have a token noReaders which expresses the fact that there are no ongoing reads as well as a token readers n, where n is now required to be positive. Similarly, writers n is now augmented by a case noWriters. After splitting and introducing the necessary case distinctions, one obtains the functional net listed at the bottom of Figure 3.

5 Foundations: The Join Calculus

Functional nets have their formal basis in join calculus [FG96]. We present *object-based join calculus* which is a variant of the original calculus. The calculus is developed in two stages. In the first stage, we study a sequential subset comprising functions and objects. This subset can be taken as the formal basis of purely functional programs of the kind we presented in Section!1. The calculus is equivalent to (call-by-value) λ-calculus[Plo75] with a record construct, but takes the opposite position on naming functions. Where λ-calculus knows only anonymous functions, functional join calculus insists that every function have a name. Furthermore, it also insists that every intermediate result be named. As such it is quite similar to common forms of intermediate code found in compilers for functional languages.

The second stage adds fork and join operators to the constructs introduced in the first stage. The calculus developed at this stage is roughly equivalent to the original join calculus. We have dropped the polyadic functions of original join calculus in favor of the qualified definitions and names that allow us to model records and tuples. We have also simplified the syntax of original join calculus and have replaced its "heating" and "cooling" rules by structural equivalences between terms.

Both stages represent functional nets as reduction systems. There is in each case only a single rewrite rule, which is similar to the β-reduction rule of λ-calculus, thus closely matching intuitions of functional programming. By contrast, the original treatment of join calculus is based on a chemical abstract machine[BB90], a concept well established in concurrency theory. The two versions of join calculus complement each other and are (modulo some minor syntactical details) equivalent.

5.1 Purely Functional Calculus

Figure 4 presents the subset of join calculus which can express purely functional programs. The syntax of this calculus is quite small. The basic building blocks of terms are names, written here x, y, z. Names can be concatenated with periods to form qualified names, written X, Y, Z. A *term* M is either a function application XY where both the function and the argument can be qualified names. Or it is a term with a local definition, **def** D ; M. A *definition* D is a sequence of rewrite rules of the form $L = M$, where ϵ stands for the empty sequence. The *left-hand side* L of a rewrite rule is again a function application Xy, where the function name can be a qualified name, but the argument must be a simple name. The right-hand side of a rewrite rule is an arbitrary term.

Given a qualified name X, let the *simple-name prefix* first(X) be its leading simple name. I.e.

$$\begin{array}{lcl} \text{first}(x) & = & x \\ \text{first}(X.y) & = & \text{first}(X) \end{array}$$

The set of *defined names* dn(D) of a definition D of the form $Xy = M$ consists of just the

Syntax:

Names	x, y, z		
Qualified names	X, Y, Z	$=$	$x \mid X.y$
Terms	M, N	$=$	$\mathbf{def}\, D\ ;\ M \mid XY$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	Xy

Structural Equivalence:

1. α-renaming:

$$\begin{array}{rcll} \mathbf{def}\, D\ ;\ M & \equiv & \mathbf{def}\, [y/x]D\ ;\ [y/x]M & \text{if } x \in \mathrm{dn}(D), y \notin \mathrm{fn}(D, M) \\ L = M & \equiv & [y/x]L = [y/x]M & \text{if } x \in \mathrm{dn}(L), y \notin \mathrm{fn}(M) \end{array}$$

2. $(,)$ on definitions is associative and commutative with ϵ as an identity:

$$\begin{array}{rcl} D_1, D_2 & \equiv & D_2, D_1 \\ D_1, (D_2, D_3) & \equiv & (D_1, D_2), D_3 \\ D, \epsilon & \equiv & D \end{array}$$

3. Nested definitions can be merged:

$$\mathbf{def}\, D_1\ ;\ (\mathbf{def}\, D_2\ ;\ M) \quad \equiv \quad \mathbf{def}\, D_1, D_2\ ;\ M \qquad \text{if } \mathrm{dn}(D_2) \cap \mathrm{fn}(D_1) = \emptyset$$

Reduction:

$$\mathbf{def}\, D, Xy = M\ ;\ Xz \quad \rightarrow \quad \mathbf{def}\, D, Xy = M\ ;\ [Z/y]M$$

Figure 4: Purely functional calculus

simple-name prefix of X, first(X). The defined names $\mathrm{dn}(L)$ of a left-hand side L of the form Xy consist of just the formal parameter $\{y\}$. The *free names* $\mathrm{fn}(M)$ of a term M are all names which are not defined by or local to a definition in M. The free names of a definition D are its defined names and any names which are free in the definition's right hand side, yet different from the local names of D. All names occurring in a term M that are not free in M are called *bound* in M. Figure 5 presents a formal definition of these sets for the full version of join calculus with concurrency.

To avoid unwanted name capture, where free names become bound inadvertently, we will always write terms subject to the following *hygiene condition*: We assume that the set of free and bound names of every term we write are disjoint. This can always be achieved by a suitable renaming of bound variables, according to the *α-renaming* law. This law lets us rename local and defined names of definitions, provided that the new names do not clash with names which already exist in their scope.

The law is formalized by two equations, detailed in Figure 4. In these equations, $[y/x]$ denotes a substitution which maps x to y. Substitutions are functions from names to identifiers which map all but a finite number of names to themselves. The results of applying a substitution to a qualified name, a term, or a definition are tabulated in Figure 6.

$$
\begin{array}{lcl}
\mathrm{dn}(X_1y_1\& \ldots \&X_ny_n = M) & = & \{\mathrm{first}(X_1),\ \ldots,\ \mathrm{first}(X_n)\} \\
\mathrm{dn}(D_1, D_2) & = & \mathrm{dn}(D_1) \cup \mathrm{dn}(D_2) \\
\mathrm{dn}(\epsilon) & = & \emptyset \\
\mathrm{dn}(X_1y_1\& \ldots \&X_ny_n) & = & \{y_1,\ \ldots,\ y_n\} \\
 & & \\
\mathrm{fn}(XY) & = & \{\mathrm{first}(X), \mathrm{first}(Y)\} \\
\mathrm{fn}(\mathbf{def}\ D\ ;\ M) & = & (\mathrm{fn}(D) \cup \mathrm{fn}(M)) \backslash \mathrm{dn}(D) \\
\mathrm{fn}(M_1 \& M_2) & = & \mathrm{fn}(M_1) \cup \mathrm{fn}(M_2) \\
\mathrm{fn}(L = M) & = & \mathrm{dn}(L) \cup (\mathrm{fn}(M) \backslash \mathrm{dn}(L)) \\
\mathrm{fn}(D_1, D_2) & = & \mathrm{fn}(D_1) \cup \mathrm{fn}(D_2) \\
\mathrm{fn}(\epsilon) & = & \emptyset
\end{array}
$$

Figure 5: Local, defined, and free names

$$
\begin{array}{lcll}
[Y/x]x & = & Y & \\
[Y/x]z & = & z & \mathbf{if}\ x \neq z \\
[Y/x](X.z) & = & ([Y/x]X).z & \\
[Y/x](XZ) & = & ([Y/x]X)\ ([Y/x]Z) & \\
[Y/x](\mathbf{def}\ D\ ;\ M) & = & \mathbf{def}\ [Y/x]D\ ;\ [Y/x]M & \mathbf{if}\ x \notin \mathrm{dn}(D), \mathrm{first}(Y) \notin \mathrm{dn}(D) \\
[Y/x](M\ \&\ N) & = & [Y/x]M\ \&\ [Y/x]N & \\
[Y/x](L = M) & = & [Y/x]L = [Y/x]M & \mathbf{if}\ x \notin \mathrm{dn}(L), \mathrm{first}(Y) \notin \mathrm{dn}(L) \\
[Y/x](D_1, D_2) & = & [Y/x]D_1, [Y/x]D_2 & \\
[Y/x]\epsilon & = & \epsilon &
\end{array}
$$

Figure 6: Substitution

Some care is required in the definition of permissible renamings. For instance, consider the expression:

$$\mathbf{def}\ \mathsf{this.f(k)} = \mathsf{k(this)}\ ;\ \mathsf{g(this)}$$

Here the name this can be consistently renamed. For instance, the following expression would be considered equivalent to the previous one:

$$\mathbf{def}\ \mathsf{that.f(k)} = \mathsf{k(that)}\ ;\ \mathsf{g(that)}$$

On the other hand, the qualified function symbol f cannot be renamed without changing the meaning of the expression. For instance, renaming f to e would yield:

$$\mathbf{def}\ \mathsf{this.e(k)} = \mathsf{k(this)}\ ;\ \mathsf{g(this)}$$

This is clearly different from the expression we started with. The new expression passes a record with an e field to the function g, whereas the previous expressions passed a record with an f field.

Besides α-renaming there are also other equalities that identify terms. All equalities together are detailed in Figure 4. The first set of laws states that the $(,)$-connective for definitions is associative and commutative, with identity ϵ. In other words, sequences of definitions can be regarded as multi-sets. The second law states that nested definitions may be merged, provided this does not lead to name clashes. *Structural equivalence* $\equiv$ is the smallest relation which includes α-renaming and these laws and which is reflexive, transitive, and compatible (i.e. closed under formation of contexts). Terms that are related by $\equiv$ are identified with each other.

The inclusion of multi-sets of definitions in the syntax is essentially for convenience, as one can translate every program with definitions consisting of multiple rewrite rules to a program that uses just one rewrite rule for each definition [FG96]. The convenience is great enough to warrant a syntax extension because the encoding is rather heavy.

Execution of terms in our calculus is defined by rewriting. Figure 4 defines a single rewrite rule, which is analogous to β-reduction in λ-calculus. The rule states that if there is an application Xz which matches a definition of X, say $Xy = M$, then we can rewrite the application to the definition's right hand side M, after replacing the formal parameter y by the actual argument Z.

All functions in purely functional join have a single argument. However, it is possible to encode multiple argument functions using records. For instance, the two argument function

def f (x, y) = x + y

can be expressed as

def f xy = { **val** x = xy.x ; **val** y = xy.y ; x + y }

This makes use of a value definition with a qualified name on its right hand side. Such value definitions can themselves be encoded as follows.

$$\textbf{val}\ x = Y\ ;\ M \quad = \quad \textbf{def}\ k\ x = M\ ;\ k\ Y$$

where k is a fresh function name. Value definitions with more complicated right-hand sides can also be encoded using a continuation passing transform. This is further explained in Section 6.

We also admit functions which take an empty tuple of arguments, taking **def** $f() = M$ to be equivalent to **def** $fx = M$ where $x \notin \text{fn}(M)$.

As an example of functional reduction, consider the reduction of a forwarding function:

def f(x) = g(x) ; f(y) $\quad\to\quad$ **def** f(x) = g(x) ; g(y)

A slightly more complex example is the reduction of a call to an evaluation function, which takes two arguments and applies one to the other:

def apply(f,x) = f(x) ; apply(print, 1) $\quad\to\quad$ **def** apply(f,x) = f(x) ; print(1)

Names	x, y, z		
Qualified names	X, Y, Z	$=$	$x \mid X.y$
Terms	M, N	$=$	$\textbf{def}\, D \,;\, M \mid XY \mid M \,\&\, N$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	$Xy \mid L \,\&\, L$

Structural Equivalence: As in Figure 4, plus

4. (&) on terms is associative and commutative:

$$\begin{aligned} M_1 \,\&\, M_2 &\equiv M_2 \,\&\, M_1 \\ M_1 \,\&\, (M_2 \,\&\, M_3) &\equiv (M_1 \,\&\, M_2) \,\&\, M_3 \end{aligned}$$

5. Scope extrusion:

$$(\textbf{def}\, D \,;\, M) \,\&\, N \;\equiv\; \textbf{def}\, D \,;\, (M \,\&\, N) \qquad \text{if } \mathrm{dn}(D) \cap \mathrm{fn}(N) = \emptyset.$$

Reduction:

$$\begin{array}{ll} & \textbf{def}\, D, X_1y_1 \,\&\, \ldots \,\&\, X_ny_n = M \,;\, X_1Z_1 \,\&\, \ldots \,\&\, X_nZ_n \,\&\, N \\ \rightarrow & \textbf{def}\, D, X_1y_1 \,\&\, \ldots \,\&\, X_ny_n = M \,;\, [Z_1/x_1, \ldots, Z_n/x_n]M \,\&\, N \end{array}$$

Figure 7: Full Calculus with Concurrency

5.2 Full Calculus with Concurrency

Figure 7 presents the full version of object-based join calculus with concurrency. It needs only one syntactic extension compared to the purely functional subset: the & operator is now introduced as *fork* operator on terms and as a *join* operator on left-hand sides.

The notion of structural equivalence is now richer than in the purely functional subset. Besides α-renaming and the laws concerning definitions, there are two other sets of laws which identify terms. First, the fork operator is assumed to be associative and commutative. Second, we have a *scope extrusion* law, which states that the scope of a local definition may be extended dynamically over other operands of a parallel composition, provided this does not lead to clashes between names bound by the definition and free names of the terms that are brought in scope.

There is still just one reduction rule, and this rule is essentially the same as in the functional subset. The major difference is that now a rewrite step may involve sets of function applications, which are composed in parallel.

The new laws of structural equivalence are necessary to bring parallel subterms which are "far apart" next to each other, so that they can match the join pattern of left-hand side. For instance, in the following example of semaphore synchronization two transformations with structural equivalences are necessary before rewrite steps can be performed.

Funnel program:

```
def newChannel = ( def this.read & this.write(x) = x ; this );
val chan = newChannel;
chan.read & chan.write(1)
```

Join calculus program and its reduction:

```
     def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
     def k3(chan) = chan.read(k0) & chan.write(1);
     newChannel(k3)
→
     def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
     def k3(chan) = chan.read(k0) & chan.write(1);
     def this'.read(k2') & this'.write(x') = k2'(x');
     k3(this')
→
     def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
     def k3(chan) = chan.read(k0) & chan.write(1);
     def this'.read(k2') & this'.write(x') = k2'(x');
     this'.read(k0) & this'.write(1);
→
     def newChannel(k1) = (def this.read(k2) & this.write(x) = k2(x); k1(this));
     def k3(chan) = chan.read(k0) & chan.write(1);
     def this'.read(k2') & this'.write(x') = k2'(x');
     k0(1)
```

Figure 8: Reduction involving an asynchronous channel object

```
     def getLock(k) & releaseLock() = k();
     releaseLock() & (def k'() = f() & g(); getLock(k'))
≡    (by commutativity of &)
     def getLock(k) & releaseLock() = k();
     (def k'() = f() & g(); getLock(k')) & releaseLock()
≡    (by scope extrusion)
     def getLock(k) & releaseLock() = k();
     def k'() = f() & g(); getLock(k') & releaseLock()
→    def getLock(k) & releaseLock() = k(); def k'() = f() & g(); k'()
→    def getLock(k) & releaseLock() = k(); def k'() = f() & g(); f() & g()
```

As an example of reduction of a program with objects and concurrency, consider the Funnel program at the top of Figure 8. The program defines an asynchronous channel using function newChannel and then reads and writes that channel.

This program is not yet in the form mandated by join calculus since it uses a synchronous function and a **val** definition. We can map this program into join calculus by adding continuation functions which make control flow for function returns and value definitions explicit. The second half of Figure 8 shows how this program is coded in object-based join calculus and how it is reduced. Schemes which map from our programming notation to join calculus are further discussed in the next section.

6 Syntactic Abbreviations

Even the extended calculus discussed in the last section is a lot smaller than the Funnel programming notation we have used in the preceding sections. This section fills the gap, by showing how Funnel constructs which are not directly supported in object-based join calculus can be mapped into equivalent constructs which are supported.

Direct style An important difference between Funnel and join calculus is that Funnel has synchronous functions and **val** definitions which bind the results of synchronous function applications. To see the simplifications afforded by these additions, it suffices to compare the Funnel program of Figure 8 with its join calculus counterpart. The join calculus version is much more cluttered because of the occurrence of the continuations k_i. Programs which make use of synchronous functions and value definitions are said to be in *direct style*, whereas programs that don't are said to be in *continuation passing style*. Join calculus supports only continuation passing style. To translate direct style programs into join calculus, we need a *continuation passing transform*. This transformation gives each synchronous function an additional argument which represents a continuation function, to which the result of the synchronous function is then passed.

The source language of the continuation passing transform is object-based join calculus extended with result expressions X and value definitions **val** $x = M$; N.

For the sake of the following explanation, we assume different alphabets for synchronous and asynchronous functions. We let X^s range over qualified names whose final selector is a synchronous function, whereas X^a ranges over qualified names whose final selector is an asynchronous function. In practice, we can distinguish between synchronous and asynchronous functions also by means of a type system, so that different alphabets are not required.

Our continuation passing transform for terms is expressed as a function $[[M]]_k$ which takes a term M in the source language and a name k representing a continuation as arguments, mapping these to a term in object-based join calculus. It makes use of a helper function which maps a definition in the source language to one in object-based join calculus. The transforms are defined as follows.

$$
\begin{array}{llcl}
[[\ \mathbf{val}\ \mathsf{x} = \mathsf{M}\ ;\ \mathsf{N} &]]\mathsf{k} & = & \mathbf{def}\ \mathsf{k'}\ \mathsf{x} = [[\ \mathsf{N}\ ]]\mathsf{k}\ ;\ [[\ \mathsf{M}\ ]]\mathsf{k'} \\
[[\ \mathsf{X} &]]\mathsf{k} & = & \mathsf{k(X)} \\
[[\ \mathsf{X}^s\mathsf{(Y)} &]]\mathsf{k} & = & \mathsf{X}^s\mathsf{(Y, k)} \\
[[\ \mathsf{X}^a\mathsf{(Y)} &]]\mathsf{k} & = & \mathsf{X}^a\mathsf{(Y)} \\
[[\ \mathsf{M}\ \&\ \mathsf{N} &]]\mathsf{k} & = & [[\ \mathsf{M}\ ]]\mathsf{k}\ \&\ [[\ \mathsf{N}\ ]]\bot \\
[[\ \mathbf{def}\ \mathsf{D}\ ;\ \mathsf{M} &]]\mathsf{k} & = & \mathbf{def}\ [[\ \mathsf{D}\ ]]\ ;\ [[\ \mathsf{M}\ ]]\mathsf{k} \\
[[\ \mathsf{L} = \mathsf{M} &]] & = & [[\ \mathsf{L}\ ]]\mathsf{k'} = [[\ \mathsf{M}\ ]]\mathsf{k'} \\
[[\ \mathsf{D, D'} &]] & = & [[\ \mathsf{D}\ ]],\ [[\ \mathsf{D'}\ ]] \\
[[\ \epsilon &]] & = & \epsilon
\end{array}
$$

Here, the k' in the first equations for terms and definitions represent fresh continuation names. $\bot$ represents an error continuation. We can guarantee with a type system that $\bot$ continuations are never called. Note that the transform for rewrite rules $L = M$ applies the transform for terms to the pattern L. This is well-defined since the syntax of patterns is contained in the syntax of terms.

The original paper on join [FG96] defines a different contination passing transform. That transform allows several functions in a join pattern to carry results. Consequently, in the body of a function it has to be specified to which of the functions of a left hand side a result should be returned to. The advantage of this approach is that it simplifies the implementation of rendevouz situations like the synchronous channel of Section 4. The disadvantage is a more complex construct for function returns.

Structured Terms In Funnel, the function part and arguments of a function application can be arbitrary terms, whereas join calculus admits only identifiers. Terms as function arguments can be expanded out by introducing names for intermediate results.

$$\mathsf{M}\ (\mathsf{N}) \quad \Rightarrow \quad \mathbf{val}\ \mathsf{x} = \mathsf{M};\ \mathbf{val}\ \mathsf{y} = \mathsf{N};\ \mathsf{x}\ \mathsf{y}$$

The resulting expression can be mapped into join calculus by applying the continuation passing transform. The same principle is also applied in other situations where structured terms appear yet only identifiers are supported. E.g. a term M used as function result can be expanded to

$$\mathbf{val}\ \mathsf{x} = \mathsf{M}\ ;\ \mathsf{x}$$

We assume here that names in the expanded term which are not present in the original source term are fresh.

Sequencing Funnel supports sequencing of expressions. A sequential composition $M\ ;\ N$ is simply regarded as a shorthand for a value definition, the expansion being as follows:

$$\mathsf{M}\ ;\ \mathsf{N} \quad \Rightarrow \quad \mathbf{val}\ \mathsf{x} = \mathsf{M};\ \mathsf{N}$$

Primitive data types Funnel supports boolean values and conditional expressions of the form **if** (M) N_1 **else** N_2. These can be expanded using the visitor technique of Section 1. For instance, here are the expansions for the boolean values true and false and the conditional expression:

$$\begin{array}{lll}
\mathsf{true} & \Rightarrow & (\ \mathbf{def}\ \mathsf{match}\ \mathsf{v} = \mathsf{v.true}\ ) \\
\mathsf{false} & \Rightarrow & (\ \mathbf{def}\ \mathsf{match}\ \mathsf{v} = \mathsf{v.false}\ ) \\
\mathbf{if}\ (\mathsf{M})\ \mathsf{T}\ \mathbf{else}\ \mathsf{F} & \Rightarrow & \mathsf{M.match}\ (\ \mathbf{def}\ \mathsf{true} = \mathsf{T},\ \mathsf{false} = \mathsf{F}\ )
\end{array}$$

This is essentially the Church encoding for booleans in λ-calculus. The only difference is that we use records where the encoding into λ-calculus uses λ-abstractions. Analogous techniques can be employed to encode other primitive types such as integers or characters.

7 Conclusion and Related Work

The first five sections of this paper have shown how a large variety of program constructs can be modelled as functional nets. The last two sections have shown how functional nets themselves can be expressed in object-based join calculus. Taken together, these steps constitute a reductionistic approach, where a large body of notations and patterns of programs is to be distilled into a minimal kernel. The reduction to essentials is useful since it helps clarify the meaning of derived program constructs and the interactions between them.

Ever since the inception of Lisp [MAE+69] and Landin's ISWIM [Lan66], functional programming has pioneered the idea of developing programming languages from calculi. Since then, there has been an extremely large body of work which aims to emulate the FP approach in a more general setting. One strand of work has devised extensions of lambda calculus with state [FH92, SF93, SRI91, ORH93, AS98] or non-determinism and concurrency [Bou89, dP95, Bou97]. Another strand of work has been designed concurrent functional languages [GMP89, Rep91, AMST97] based on some other operational semantics. Landin's program has also been repeated in the concurrent programming field, for instance with Occam and CSP [Hoa85], Pict [PT97] and π-calculus [MPW92], or Oz and its kernel [SHW95].

Our approach is closest to the work on join calculus [FG96, FGL+96, FM97, Fes98]. Largely, functional nets as described here constitute a simplification and streamlining of the original treatment of join, with object-based join calculus and qualified definitions being the main innovation. These notes are an adaptation and extension of two previous papers [Ode00a, Ode00b].

Acknowledgements Matthias Zenger and Christoph Zenger have designed several of the examples. I thank them and also Philippe Altherr and Michel Schinz for suggesting numerous improvements to the paper.

References

[AB96] Andrea Asperti and Nadia Bussi. Mobile petri nets. Technical Report UBLCS-96-10, University of Bolognia, May 1996.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.

[AMST97] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, 1997.

[AN00] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts – a tour of Piccola. In *Software Architectures and Component Technology*. Kluwer, 2000.

[AOS+00] Philippe Altherr, Martin Odersky, Michel Schinz, Matthias Zenger, and Christoph Zenger. Funnel distribution. available from `http://lampwww.epfl.ch/funnel`, July 2000.

[AS98] Zena Ariola and Amr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 62–74, 1998.

[ASS84] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1984.

[Bac78] John W. Backus. Can programming be liberated from the von neumann style? *Communications of the ACM*, 21:613–641, 1978.

[Bar84] 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.

[BB90] Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 81–94, January 1990.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998. 2nd edition.

[Bou89] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT '1989*, pages 149–161, New York, March 1989. Springer-Verlag. Lecture Notes in Computer Science 351.

[Bou97] Gérard Boudol. The pi-calculus in direct style. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1997.

[Chu51] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.

[CMMS94] 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.

[dP95] Ugo de'Liguoro and Adolfo Piperno. Non deterministic extensions of untyped λ-calculus. *Information and Computation*, 122(2):149–177, 1 November 1995.

[Fes98] Fabrice Le Fessant. *The JoCaml reference manual*. INRIA Rocquencourt, 1998. Available from `http://join.inria.fr`.

[FG96] 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.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, August 26-29 1996. Springer-Verlag. LNCS 1119.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[FM97] Cédric Fournet and Luc Maranget. *The Join-Calculus Language*. INRIA Rocquencourt, 1997. Available from `http://join.inria.fr`.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GMP89] 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.

[Han72] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 12(10), October 74.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.

[Ker87] Jon Kerridge. *Occam programming: a practical approach.* Blackwell Scientific, 1987.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966.

[MAE+69] John McCarthy, Paul W. Abrahams, D. J. Edwards, T. P. Hart, and I. L. Levin. *Lisp 1.5 Programmer's Manual.* MIT Press, 1969.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Ode00a] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.

[Ode00b] Martin Odersky. Programming with functional nets. Technical Report 2000/331, Departement d'Informatique, EPFL, March 2000.

[ORH93] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[OZZ00] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. submitted for publication, available from `http://lampwww.epfl.ch/papers/clti/clti-colored.ps.gz`, July 2000.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991. ISBN 0-521-39022-2.

[Pet62] C.A. Petri. Kommunikation mit Automaten. Schriften des IIM 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation: Technical Report RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, New Jersey, Contract AF 30 (602)-3324, 1966.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PT97] 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.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, 1998.

[Rei85] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[Rep91] 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.

[SF93] Amr Sabry and John Field. Reasoning about explicit and implicit representations of state. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 17–30, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[SHW95] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 29–48. The MIT Press, 1995.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 2nd edition, 1983.