Functional Nets

Martin Odersky

École Polytechnique Fédérale de Lausanne

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 presents functional nets, reviews Join calculus, and shows how the two relate.

1 Introduction

Functional nets are a way to think about programs and computation which is born from a fusion of the essential ideas of functional programming and Petri nets. 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* [15, 16]. 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 elegance from close connections to a fundamental underlying calculus. λ-calculus [10, 5] 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 this paper is two-fold. First, it aims 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 paper develops 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

[1] Revised from a paper in Proc. European Symposium on Programming 2000, Lecture Notes in Computer Science, © Springer Verlag.

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[1], 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[17] or JoCaml[14]. 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.

As every program notation should be, functional nets are intended to be strongly typed, in the sense that all type errors should be detected rather than leading to unspecified behavior. We leave open whether type checking is done statically at compile time or dynamically at run time. Our examples do not mention types, but they are all of a form that would be checkable using a standard type system with recursive records, subtyping and polymorphism.

The rest of this paper is structured as follows. Section 2 introduces functional nets and qualified definitions. Sections 3 and 4 show how common functional and imperative programming patterns can be modeled as functional nets. Section 5 discusses concurrency and shows how functional nets model a wide spectrum of process synchronization techniques. Section 6 introduces object-based join calculus as the formal foundation of functional nets. Section 7 discusses how the programming notation used in previous sections can be encoded in this calculus. Section 8 discusses related work and concludes.

2 A First Example

Consider the task of implementing a one-place buffer, which connects producers and consumers of data. Producers call a function put to deposit data into the buffer while consumers call a function get to retrieve data from the buffer. There can be at most one datum in the buffer at any one time. A put operation on a buffer which is already full blocks until the buffer is empty. Likewise, a get on an empty buffer blocks until the buffer is full. This specification is realized by the following simple functional net:

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

[1] Previously named Silk in the original, published version of the paper. We changed the name because of the possible confusion with the concurrent C dialect Cilk.

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

Function put takes a single argument, x. We often write a function argument without surrounding parentheses, e.g. put x instead of put(x). We also admit functions like get that do not take any argument; one can imagine that every occurrence of such a function is augmented by an implicit empty tuple as argument, e.g. get becomes get().

The two equations define *rewrite rules*. A set of function calls that matches the left-hand side of an equation may be rewritten to the equation's right-hand side. The & symbol denotes parallel composition. We sometimes call & a *fork* if it appears on an equation's right-hand side, and a *join* if it appears on the left. Consequently, the left-hand sides of equations are also called *join patterns*.

For instance, the equation

```
get & full x  =  x & empty
```

states that if there are two concurrent calls, one to get and the other to full x for some value x, then those calls may be rewritten to the expression x & empty. That expression returns x as get's result and in parallel calls function empty. Unlike get, empty does not return a result; it's sole purpose is to enable via the second rewrite rule calls to put to proceed. We call result-returning functions like get *synchronous*, whereas functions like empty are called *asynchronous*.

In general, only the leftmost operand of a fork or a join can return a result. All function symbols of a left-hand side but the first one are asynchronous. Likewise, all operands of a fork except the first one are asynchronous or their result is discarded.

It's now easy to interpret the second rewrite rule,

```
put x & empty   =  () & full x
```

This rule states that two concurrent calls to put x & empty and may be rewritten to () & full x. The result part of that expression is the unit value (); it signals termination and otherwise carries no interesting information.

Clients of the buffer still need to initialize it by calling empty. A simple usage of the one-place buffer is illustrated in the following example.

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

put 1 &
  ( val y = get ; val r = y + y ; print r ; put r ) &
  ( val z = get ; val r = y * y ; print r; put r ) &
  empty
```

Besides the initializer empty there are three client processes composed in parallel. One process puts the number 1 into the buffer. The other two processes both try to get the buffer's contents and put back a modified value. The construct

```
val y = get ; ...
```

evaluates the right-hand side expression get and defines y as a name for the resulting value. The defined name y remains visible in the expression following the semicolon. By contrast, if we had written **def** y = get; ... we would have defined a function y, which each time it was called would call in turn get. The definition itself would not evaluate anything.

As usual, a semicolon between expressions stands for sequencing. The combined expression print r; put r first prints its argument r and then puts it into the buffer.

The sequence in which the client processes in the above example execute is arbitrary, controlled only by the buffer's rewrite rules. The effect of running the example program is hence the output of two numbers, either (2, 4) or (1, 2), depending which client process came first.

Objects The previous example mixed the definition of a one-place buffer and the client program using it. A better de-coupling is obtained by defining a constructor function for one-place buffers. The constructor, together with a program using it can be written as follows.

```
def newBuffer = {
   def get & full x     =  x  & empty,
       put x & empty =  () & full x;
   (get, put) & empty
};
val (get', put') = newBuffer;
put' 1 &
  ( val y = get' ; val r = y + y ; print r ; put' r ) &
  ( val z = get' ; val r = y * y ; print r ; put' r )
```

The defining equations of a one-place buffer are now local to a block, from which the pair of locally defined functions get and put is returned. Parallel to returning the result the buffer is initialized by calling empty. The initializer empty is now part of the constructor function; clients no longer can call it explicitly, since empty is defined in a local block and not returned as result of that block. Hence, newBuffer defines an object with externally visible methods get and put and private methods empty and full. The object is represented by a tuple which contains all externally visible methods.

This representation is feasible as long as objects have only a few externally visible methods, but for objects with many methods the resulting long tuples quickly become unmanageable. Furthermore, tuples do not support a notion of subtyping, where an object can be substituted for another one with fewer methods. We therefore introduce *records* as a more suitable means of aggregation where individual methods can be accessed by their names, and subtyping is possible.

The idiom for record access is standard. If r denotes a record, then $r.f$ denotes the field of r named f. We also call references of the form $r.f$ *qualified names*. The idiom for record creation is less conventional. In most programming languages, records are defined by enumerating all field names with their values. This notion interacts poorly with the forms of definitions employed in functional nets. In a functional net, one often wants to export only some of the functions defined in a join pattern whereas other functions should remain hidden. Moreover, it is often necessary to call some of the hidden functions as part of the object's initialization.

To streamline the construction of objects, we introduce qualified names not only for record accesses, but also for record definitions. For instance, here is a re-formulation of the newBuffer function using qualified definitions.

```
def newBuffer = {
   def this.get & full x      =  x  & empty,
       this.put x & empty  =  () & full x;
   this & empty
};
val buf = newBuffer;
buf.put 1 &
  ( val y = buf.get ; val r = y + y ; print r ; buf.put r ) &
  ( val z = buf.get ; val r = y * y ; print r ; buf.put r )
```

Note the occurrence of the qualified names this.get and this.put on the left-hand side of the local definitions. These definitions introduce three local names:

- the local name this, which denotes a record with two fields, get and put, and
- local names empty and full, which denote functions.

Note that the naming of this is arbitrary, any other name would work equally well. Note also that empty and full are not part of the record returned from newBuffer, so that they can be accessed only internally.

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

Some Notes on Syntax We assume the following order of precedence, from strong to weak:

( ) and (.)   ,   (&)   ,   (=)   ,   (,)   ,   (;)   .

That is, function application and selection bind strongest, followed by parallel composition, followed by the equal sign, followed by comma, and finally followed by semicolon. Function application and selection are left associative, & is associative, and ; is right associatve. 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.

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 rules are as follows: (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 after the insertion point can start an expression or definition. 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 = {
    def this.get & full x      =  x  & empty
    def this.put x & empty  =  () & full x
    this & empty
}
val buf = newBuffer
buf.put 1 &
  { val y = buf.get ; val r = y + y ; print r ; buf.put r } &
  { val z = buf.get ; val r = y * y ; print r ; buf.put r }
```

A common special case of a qualified definition is the definition of a record with only externally visible methods:

```
( def this.f = ... , this.g = ... ; this )
```

This idiom can be abbreviated by omitting the this qualifier and writing only the definitions.

```
( def f = ... , g = ... )
```

3 Functional Programming

A functional net that does not contain any occurrences of & is a purely functional program. For example, here's the factorial function written as a functional net.

```
def factorial n  =  if (n == 0) 1
                    else n * factorial (n–1)
```

Except for minor syntactical details, there's nothing which distinguishes this program from a program written in a functional language like Haskell or ML. We assume that evaluation of function arguments is strict: In the call f (g x), g x will be evaluated first and its value will be passed to f.

Functional programs often work with recursive data structures such as trees and lists. In Lisp or Scheme such data structures are primitive S-expressions, whereas in ML or Haskell they are definable as algebraic data types. Our functional net notation does not have a primitive tree type, nor has it constructs for defining algebraic data types and for pattern matching their values. It does not need to, since these constructs can be represented with records, using the *Visitor* pattern[18].

The visitor pattern is the object-oriented version of the standard Church encoding of algebraic data types. 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:

```
def Nil = ...
def Cons (x, xs) = ...
```

The intention is that our translation of pattern matching would call either the Nil method or the Cons method of a given visitor, 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.

Assume we have already defined a method match for lists that takes a list visitor as argument and has the behavior just described. Then one could write an isEmpty test function over lists as follows:

```
def isEmpty xs = xs.match {
   def Nil = true
   def Cons (x, xs1) = false
}
```

More generally, every function over lists can be defined in terms of match. So, in order to define a record which represents a list, all we need to do is to provide a match method. How should match be defined? 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 different implementations for match. The implementations are straightforward:

```
val List = {
   def Nil          = { def match v = v.Nil }
   def Cons (x, xs) = { 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. We have chosen to wrap the Nil and Cons constructors in another record, named List. List acts as a module, which provides the constructors of the list data type. Clients of the List module then construct lists using qualified names List.Nil and List.Cons. Example:

```
def concat (xs, ys) = xs.match {
   def Nil = ys
   def Cons (x, xs) = List.Cons (x, concat (xs1, ys))
}
```

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

4 Imperative Programming

Imperative programming extends purely functional programming with the addition of mutable variables. A mutable variable can be modeled as a reference cell object, which can be constructed as follows.

```
def newRef initial = {
   def this.value     & state x  =  x  & state x,
       this.update y  & state x  =  () & state y

   this & state initial
}
```

The structure of these definitions 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. The asynchronous function state serves to remember the variable's current value. The reference cell is initialized by calling state with the initial value.

Here is a simple example of how references are used:

```
val count = newRef 0
def increment = count.update (count.value + 1)
increment
```

Building on reference cell objects, we can express the usual variable access notation of imperative languages by two simple syntactic expansions:

```
var x := E      expands to     val _x = newRef E ; def x = _x.value
x := E          expands to     _x.update E
```

The count example above could then be written more conventionally as follows.

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

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.

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

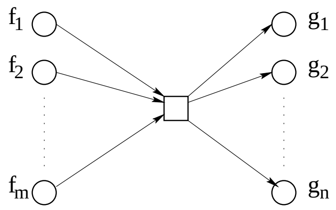

Fig. 1. Analogy to Petri nets

```
def newObjectWithIdentity = {
   def this.eq other      & flag x  =  resetFlag (other.testFlag & flag true)
       this.testFlag      & flag x  =  x & flag x
       resetFlag result   & 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.

5 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 [29, 32]. In fact, there are direct analogies between the elements of Petri nets and functional nets. This is illustrated in Figure 1.

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 [24] 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 [4] 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.

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
}
```

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 newObjectWithIdentity  =  {
   def this.eq other = global.getLock ; this.testEq other ; global.releaseLock
       this.testEq other & flag x  =  resetFlag (other.testFlag & flag true)
       this.testFlag       & flag x  =  x & flag x
       resetFlag result   & flag x  =  x & flag false

   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.

Asynchronous Channels Quite similar to a semaphore is the definition of an asynchronous channel with two operations, read and write:

```
def newAsyncChannel = {
    def this.read & this.write x  =  x
    this
}
```

Asynchronous channels are the fundamental communication primitive of asynchronous π calculus [8, 23] and languages based on it, e.g. Pict[30] or Piccola[1]. A typical usage scenario of an asynchronous channel would be:

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

The producer in the above scenario writes consecutive integers to the channel c which are read and printed by the consumer. The writing is done asynchronously, in parallel to the rest of the body of the producer's while loop. Hence, there is no guarantee that numbers will be read and printed in the same order as they were written.

Synchronous Channels A potential problem with the previous example 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 was 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[27]. They can be represented as functional nets as follows.

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

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 written as follows.

```
val c = newSyncChannel
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
```

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* [20, 21] which ensures that only one of a set of functions f_1, ..., f_k can be active at any one time. A monitor is easily represented using an additional asynchronous function, turn. The turn token acts as a resource which is consumed at the start of each function f_i and which is reproduced at the end:

```
def f1 & turn  =   ... ; turn,
                    ⋮
    fk & turn  =   ... ; turn
```

For instance, here is an example of a counter which can be incremented and decremented:

```
def newBiCounter = {
  var count := 0
  def this.increment & turn = count := count + 1 ; turn
  def this.decrement & turn = count := count – 1 ; turn
  this
}
```

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 2. 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 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 2.

6 Foundations: The Join Calculus

Functional nets have their formal basis in join calculus [15]. We now present this basis, in three stages. In the first stage, we study a subset of join calculcus which

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
}
```

Fig. 2. Readers/writers synchronization

can be taken as the formal basis of purely functional programs. This calculus is equivalent to (call-by-value) λ-calculus[31], 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 equivalent in principle to the original join calculus, but some syntactical details have changed.

The third stage adds qualified names in definitions and accesses. The calculus developed in this stage can model the object-based functional nets we have used.

All three stages represent functional nets as a reduction system. There is

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Terms	M, N	$=$	$\mathbf{def}\, D\ ;\ M \ \mid\ x(\tilde{y})$
Definitions	D	$=$	$L = M$
Left-hand sides	L	$=$	$x(\tilde{y})$
Reduction contexts	R	$=$	$[\,]\ \mid\ \mathbf{def}\, D\ ;\ R$

Structural Equivalence: α-renaming.

Reduction:

$$\mathbf{def}\, x(\tilde{y}) = M\ ;\ R[x(\tilde{z})] \quad \rightarrow \quad \mathbf{def}\, x(\tilde{y}) = M\ ;\ R[[\tilde{z}/\tilde{y}]M]$$

Fig. 3. Pure functional calculus

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[6], a concept well established in concurrency theory. The two versions of join calculus complement each other and are (modulo some minor syntactical details) equivalent.

6.1 Pure Functional Calculus

Figure 3 presents the subset of join calculus which can express purely functional programs. The syntax of this calculus is quite small. A *term* M is either a function application $x(\tilde{y})$ or a term with a local definition, **def** D ; M (we let $\tilde{x}$ stand for a sequence $x_1, \ldots, x_n$ of names, where $n \geq 0$). A *definition* D is a single rewrite rule the form $L = M$. The *left-hand side* L of a rewrite rule is again a function application $x(\tilde{y})$. We require that the formal parameters y_i of a left-hand side are pairwise disjoint. The right-hand side of a rewrite rule is an arbitrary term.

The set of *defined names* $\mathrm{dn}(D)$ of a definition D of the form $x(\tilde{y}) = M$ consists of just the function name x. Its *local names* $\mathrm{ln}(D)$ are the formal parameters $\tilde{y}$. 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 6 presents a formal definition of these sets for the object-based extension of join calculus.

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. It is formalized by two equations. First,

$$\mathbf{def}\, x(\tilde{y}) = M \; ; N \equiv \mathbf{def}\, u(\tilde{y}) = [u/x]M \; ; [u/x]N$$

if $u \notin \mathrm{fn}(M) \cup \mathrm{fn}(N)$. Second,

$$\mathbf{def}\, x(\tilde{y}) = M \; ; N \equiv \mathbf{def}\, x(\tilde{v}) = [\tilde{v}/\tilde{y}]M \; ; N$$

if $\{\tilde{v}\} \cap \mathrm{fn}(M) = \emptyset$ and the elements of $\tilde{v}$ are pairwise disjoint. Here, $[u/x]$ and $[\tilde{v}/\tilde{y}]$ are substitutions which map x and $\tilde{y}$ to u and $\tilde{v}$. Generally, substitutions are idempotent functions over names which map all but a finite number of names to themselves. The *domain* $\mathrm{dom}(\sigma)$ of a substitution σ is the set of names not mapped to themselves under σ.

Generally, we will give in each case a structural equivalence relation $\equiv$ which is assumed to be reflexive, transitive, and compatible (i.e. closed under formation of contexts). Terms that are related by $\equiv$ are identified with each other. For the purely functional calculus, $\equiv$ is just α-renaming. Extended calculi will have richer notions of structural equivalence.

Execution of terms in our calculus is defined by rewriting. Figure 3 defines a single rewrite rule, which is analogous to β-reduction in λ-calculus. The rule can be sketched as follows:

$$\mathbf{def}\, x(\tilde{y}) = M \; ; \; \ldots x(\tilde{z}) \ldots \quad \rightarrow \quad \mathbf{def}\, x(\tilde{y}) = M \; ; \; \ldots [\tilde{z}/\tilde{y}]M \ldots$$

That is, if there is an application $x(\tilde{z})$ which matches a definition of x, say $x(\tilde{y}) = M$, then we can rewrite the application to the definition's right hand side M, after replacing formal parameters $\tilde{y}$ by actual arguments $\tilde{z}$.

The above formulation is not yet completely precise because we still have to specify where exactly a reducible application can be located in a term. Clearly, the application must be within the definition's scope. Also, we want to reduce only those applications which are not themselves contained in another local definition. For instance, in

```
def f (x, k) = k x ;
def g (x, k) = f (1, k) ;
f(2, k)
```

we want to reduce only the second application of f, not the first one which is contained in the body of function g. This restriction in the choice of reducible applications avoids potentially unnecessary work. For instance in the code fragment above g is never called, so it would make no sense to reduce its body. More importantly, once we add side-effects to our language, it is essential that the body of a function is executed (i.e. reduced) only when the function is applied.

The characterization of reducible applications can be formalized using the idea of a *reduction context*[2]. A *context* C is a term with a hole, which is written $[\,]$. The expression $C[M]$ denotes the term resulting from filling the hole of the context C with M. A *reduction context* R is a context of a special form, in which the hole can be only at places where a function application would be reducible. The set of possible reduction contexts for our calculus is generated by a simple context free grammar, given in Figure 3. This grammar says that reduction can only take place at the top of a term, or in the scope of some local definitions.

Reduction contexts are used in the formulation of the reduction law in Figure 3. Generally, we let the reduction relation $\rightarrow$ between terms be the smallest compatible relation that contains the reduction law.

An alternative formulation of the reduction rule abstracts from the concrete substitution operator:

$$\mathbf{def}\, L = M\ ;\ R[\sigma L] \quad \rightarrow \quad \mathbf{def}\, L = M\ ;\ R[\sigma M]$$

if σ is a substitution from names to names with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

The advantage of the alternative formulation is that it generalizes readily to the more complex join patterns which will be introduced in the next sub-section.

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

$$\mathbf{def}\, \mathsf{f(x)} = \mathsf{g(x)}\ ;\ \mathsf{f(y)} \quad \rightarrow \quad \mathbf{def}\, \mathsf{f(x)} = \mathsf{g(x)}\ ;\ \mathsf{g(y)}$$

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

$$\mathbf{def}\, \mathsf{apply(f,x)} = \mathsf{f(x)}\ ;\ \mathsf{apply(print,\ 1)} \quad \rightarrow \quad \mathbf{def}\, \mathsf{apply(f,x)} = \mathsf{f(x)}\ ;\ \mathsf{print(1)}$$

6.2 Canonical join calculus

Figure 4 presents the standard version of join calculus. Compared to the purely functional subset, there are three syntax additions: First and second, & is now introduced as *fork* operator on terms and as a *join* operator on left-hand sides. Third, definitions can now consist of more than one rewrite rule, so that multiple definitions of the same function symbol are possible.

The latter addition 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 [15]. The convenience is great enough to warrant a syntax extension because the encoding is rather heavy.

The notion of structural equivalence is now more refined than in the purely functional subset. Besides α-renaming, there are three other sets of laws which identify terms. First, the fork operator is assumed to be associative and commutative. Second, the comma operator which conjoins rewrite rules is also taken to

[2] The concept is usually known as under the name "evaluation context" [11], but there's nothing to evaluate here.

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Terms	M, N	$=$	$\mathbf{def}\, D \,;\, M \mid x(\tilde{y}) \mid M \,\&\, N$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	$x(\tilde{y}) \mid L \,\&\, L$
Reduction contexts	R	$=$	$[\,] \mid \mathbf{def}\, D \,;\, R \mid R \,\&\, M \mid M \,\&\, R$

Structural Equivalence: α-renaming +

1. $(\&)$ on terms is associative and commutative:

$$M_1 \,\&\, M_2 \equiv M_2 \,\&\, M_1$$
$$M_1 \,\&\, (M_2 \,\&\, M_3) \equiv (M_1 \,\&\, M_2) \,\&\, M_3$$

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

$$D_1, D_2 \equiv D_2, D_1$$
$$D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$$
$$D, \epsilon \equiv D$$

3. Scope extrusion:

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

Reduction:

$$\mathbf{def}\, D, L = M \,;\, R[\sigma L] \quad \to \quad \mathbf{def}\, D, L = M \,;\, R[\sigma M]$$

where σ is a substitution from names to names with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

Fig. 4. Canonical join calculus

be associative and commutative, with the empty definition ϵ as identity. Finally, 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 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 structural equivalences are necessary before rewrite steps can be performed.

Syntax:

Names	$a, b, c, \ldots, x, y, z$		
Identifiers	I, J	$=$	$x \mid I.x$
Terms	M, N	$=$	$\mathbf{def}\, D\, ;\, M \mid I(\tilde{J}) \mid M \,\&\, N$
Definitions	D	$=$	$L = M \mid D, D \mid \epsilon$
Left-hand sides	L	$=$	$I(\tilde{y}) \mid L \,\&\, L$
Reduction contexts	R	$=$	$[\,] \mid \mathbf{def}\, D\, ;\, R \mid R \,\&\, M \mid M \,\&\, R$

Structural Equivalence: α-renaming +

1. (&) on terms is associative and commutative:
$$M_1 \,\&\, M_2 \equiv M_2 \,\&\, M_1$$
$$M_1 \,\&\, (M_2 \,\&\, M_3) \equiv (M_1 \,\&\, M_2) \,\&\, M_3$$

2. $(,)$ on definitions is associative and commutative with ϵ as an identity:
$$D_1, D_2 \equiv D_2, D_1$$
$$D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$$
$$D, \epsilon \equiv D$$

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

Reduction:

$$\mathbf{def}\, D, L = M\, ;\, R[\sigma L] \quad \rightarrow \quad \mathbf{def}\, D, L = M\, ;\, R[\sigma M]$$

where σ is a substitution from names to identifiers with $\mathrm{dom}(\sigma) \subseteq \mathrm{ln}(L)$.

Fig. 5. Object-based join calculus

---

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()

6.3 Object-Based Calculus

Figure 5 presents the final stage of our progression, object-based join calculus. The only syntactical addition with respect to Figure 4 is that identifiers can now

$\mathrm{first}(x)$	$= x$	$\mathrm{ln}(I(x_1, \ldots, x_n))$	$= \{x_1, \ldots, x_n\}$
$\mathrm{first}(I.f)$	$= \mathrm{first}(f)$	$\mathrm{ln}(L_1 \& L_2)$	$= \mathrm{ln}(L_1) \cup \mathrm{ln}(L_2)$
		$\mathrm{fn}(I(J_1, \ldots, J_n))$	$= \{\mathrm{first}(I), \mathrm{first}(J_1), \ldots, \mathrm{first}(J_n)\}$
$\mathrm{dn}(I(\tilde{x}))$	$= \mathrm{first}(I)$	$\mathrm{fn}(\mathbf{def}\, D \;;\, M)$	$= (\mathrm{fn}(D) \cup \mathrm{fn}(M)) \backslash \mathrm{dn}(D)$
$\mathrm{dn}(L_1 \& L_2)$	$= \mathrm{dn}(L_1) \cup \mathrm{dn}(L_2)$	$\mathrm{fn}(M_1 \& M_2)$	$= \mathrm{fn}(M_1) \cup \mathrm{fn}(M_2)$
$\mathrm{dn}(D_1, D_2)$	$= \mathrm{dn}(D_1) \cup \mathrm{dn}(D_2)$	$\mathrm{fn}(L = M)$	$= \mathrm{dn}(L) \cup (\mathrm{fn}(M) \backslash \mathrm{ln}(L))$
		$\mathrm{fn}(D_1, D_2)$	$= \mathrm{fn}(D_1) \cup \mathrm{fn}(D_2)$

Fig. 6. Local, defined, aiand free names

---

be qualified names. A qualified name I is either a simple name x or a qualified name followed by a period and a simple name. Qualified names can appear as the operands of a function application and as defined function symbols in a definition.

Perhaps surprisingly, this is all that changes! The structural equivalences and reduction rules stay exactly as they were formulated for canonical join calculus. However, a bit of care is required in the definition of permissible renamings. For instance, consider the following object-based functional net:

def this.f(k) & g(x) = k(x) ; k'(this) & g(0)

In this net, both this and g can be consistently renamed. For instance, the following expression would be considered equivalent to the previous one:

def that.f(k) & h(x) = k(x) ; k'(that) & h(0)

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:

def this.e(k) & g(x) = k(x) ; k'(this) & g(0)

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

Figure 6 reflects these observations in the definition of local, defined, and free names for object-based join calculus. Note that names occurring as field selectors are neither free in a term, nor are they defined or local. Hence α-renaming does not apply to record selectors.

The α-renaming rule is now formalized as follows. Let a *renaming* θ be a substitution from names to names which is injective when considered as a function from $\mathrm{dom}(\theta)$ (remember that $\mathrm{dom}(\theta) = \{x \mid \theta(x) \neq x\}$). Then,

$$\mathbf{def}\, D \;;\, M \equiv \mathbf{def}\, \theta D \;;\, \theta M$$

if θ is a renaming with $\mathrm{dom}(\theta) \subseteq \mathrm{dn}(D)$ and $\mathrm{codom}(\theta) \cap (\mathrm{fn}(D) \cup \mathrm{fn}(M)) = \emptyset$. Furthermore,

$$\mathbf{def}\, D, L = M \;;\, N \equiv \mathbf{def}\, D, \theta L = \theta M \;;\, N$$

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(k_1) = (**def** this.read(k_2) & this.write(x) = k_2(x); k_1(this));
def k_3(chan) = chan.read(k_0) & chan.write(1);
newChannel(k_3)

$\rightarrow$

def newChannel(k_1) = (**def** this.read(k_2) & this.write(x) = k_2(x); k_1(this));
def k_3(chan) = chan.read(k_0) & chan.write(1);
def this'.read(k_2') & this'.write(x') = k_2'(x');
k_3(this')

$\rightarrow$

def newChannel(k_1) = (**def** this.read(k_2) & this.write(x) = k_2(x); k_1(this));
def k_3(chan) = chan.read(k_0) & chan.write(1);
def this'.read(k_2') & this'.write(x') = k_2'(x');
this'.read(k_0) & this'.write(1);

$\rightarrow$

def newChannel(k_1) = (**def** this.read(k_2) & this.write(x) = k_2(x); k_1(this));
def k_3(chan) = chan.read(k_0) & chan.write(1);
def this'.read(k_2') & this'.write(x') = k_2'(x');
k_0(1)

Fig. 7. Reduction involving an asynchronous channel object

if θ is a renaming with $\mathrm{dom}(\theta) \subseteq \mathrm{ln}(L)$ and $\mathrm{codom}(\theta) \cap \mathrm{fn}(M) = \emptyset$.

The definitions of Figure 6 and the α-renaming rule apply as stated to all three versions of join calculus, not only to the final object-based version. When reduced to the simpler syntax of previous calculi, the new definitions are equivalent to the old ones.

As an example of object-based reduction consider the Funnel program at the top of Figure 7. 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 7 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.

7 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 7 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 $(\mathsf{I}_1, ..., \mathsf{I}_n)$ and value definitions **val** $(\mathsf{x}_1, ..., \mathsf{x}_n) = \mathsf{M}\ ;\ \mathsf{N}$. Single names in results and value definitions are also included as they can be expressed as tuples of length 1.

For the sake of the following explanation, we assume different alphabets for synchronous and asynchronous functions. We let I^s range over identifiers whose final selector is a synchronous function, whereas I^a ranges over identifiers 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 TC which takes a term in the source language and a name representing a continuation as arguments, mapping these to a term in object-based join calculus. It makes use of a helper function TD which maps a definition in the source language to one in object-based join calculus. To emphasize the distinction between the transforms TC, TD and their syntactic expression arguments, we write syntactic expressions in [[ ]] brackets. The transforms are defined as follows.

$$
\begin{array}{lll}
TC[\![\ \mathbf{val}\ (\tilde{\mathsf{x}}) = \mathsf{M}\ ;\ \mathsf{N}\ ]\!]\mathsf{k} & = & \mathbf{def}\ \mathsf{k'}\ (\tilde{\mathsf{x}}) = TC[\![\ \mathsf{N}\ ]\!]\mathsf{k}\ ;\ TC[\![\ \mathsf{M}\ ]\!]\mathsf{k'} \\
TC[\![\ (\mathsf{I}_1, ..., \mathsf{I}_n)\ ]\!]\mathsf{k} & = & \mathsf{k}(\mathsf{I}_1, ..., \mathsf{I}_n) \\
TC[\![\ \mathsf{I}^s(\mathsf{J}_1, ..., \mathsf{J}_n)\ ]\!]\mathsf{k} & = & \mathsf{I}^s(\mathsf{J}_1, ..., \mathsf{J}_n, \mathsf{k}) \\
TC[\![\ \mathsf{I}^a(\mathsf{J}_1, ..., \mathsf{J}_n)\ ]\!]\mathsf{k} & = & \mathsf{I}^a(\mathsf{J}_1, ..., \mathsf{J}_n) \\
TC[\![\ \mathbf{def}\ \mathsf{D}\ ;\ \mathsf{M}\ ]\!]\mathsf{k} & = & \mathbf{def}\ TD[\![\ \mathsf{D}\ ]\!]\ ;\ TC[\![\ \mathsf{M}\ ]\!]\mathsf{k} \\
 & & \\
TD[\![\ \mathsf{L} = \mathsf{M}\ ]\!] & = & TC[\![\ \mathsf{L}\ ]\!]\mathsf{k'} = TC[\![\ \mathsf{M}\ ]\!]\mathsf{k'} \\
TD[\![\ \mathsf{D}, \mathsf{D'}\ ]\!] & = & TD[\![\ \mathsf{D}\ ]\!],\ TD[\![\ \mathsf{D'}\ ]\!] \\
TD[\![\ \epsilon\ ]\!] & = & \epsilon
\end{array}
$$

Here, the k' in the first equations for *TC* and *TD* represent fresh continuation names.

The original paper on join [15] 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 5. 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.

$$\mathsf{M}\ (\mathsf{N}_1,\ ....,\ \mathsf{N}_k) \quad \Rightarrow \quad \mathbf{val}\ \mathsf{x} = \mathsf{M};\ \mathbf{val}\ \mathsf{y}_1 = \mathsf{N}_1;\ ...\ \mathbf{val}\ \mathsf{y}_k = \mathsf{N}_k;\ \mathsf{x}(\mathsf{y}_1,...,\mathsf{y}_k)$$

The resulting expression can be mapped into join calculus by applying the continuation passing transform *TC*. The same principle is also applied in other situations where structured terms appear yet only identifiers are supported. E.g.:

$$\begin{array}{lcl} (\mathsf{M}_1,\ ....,\ \mathsf{M}_k) & \Rightarrow & \mathbf{val}\ \mathsf{x}_1 = \mathsf{M}_1;\ ...\ \mathbf{val}\ \mathsf{x}_k = \mathsf{M}_k;\ (\mathsf{x}_1,...,\mathsf{x}_k) \\ \mathsf{M.f} & \Rightarrow & \mathbf{val}\ \mathsf{x} = \mathsf{M};\ \mathsf{x.f} \end{array}$$

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

8 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 [26] and Landin's ISWIM [25], 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 [13, 34, 36, 28, 3] or nondeterminism and concurrency [7, 12, 9]. Another strand of work has been designed concurrent functional languages [19, 33, 2] based on some other operational semantics. Landin's programme has also been repeated in the concurrent programming field, for instance with Occam and CSP [22], Pict [30] and π-calculus [27], or Oz and its kernel [35].

Our approach is closest to the work on join calculus [15, 16, 17, 14]. 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.

Acknowledgements Many thanks to Matthias Zenger and Christoph Zenger, for designing several of the examples and suggesting numerous improvements.

References

1. F. Achermann and O. Nierstrasz. Applications = components + scripts – a tour of Piccola. In *Software Architectures and Component Technology*. Kluwer, 2000.
2. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, 1997.
3. Z. Ariola and A. 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.
4. A. Asperti and N. Bussi. Mobile petri nets. Technical Report UBLCS-96-10, University of Bolognia, May 1996.
5. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
6. G. Berry and G. Boudol. The chemical abstract machine. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 81–94, January 1990.
7. G. 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.
8. G. Boudol. Asynchrony and the pi-calculus. Research Report 1702, INRIA, May 1992.
9. G. Boudol. The pi-calculus in direct style. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1997.
10. A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, second edition, 1951.
11. E. Crank and M. Felleisen. Parameter-passing and the lambda-calculus. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 233–244, January 1991.
12. U. de'Liguoro and A. Piperno. Non deterministic extensions of untyped λ-calculus. *Information and Computation*, 122(2):149–177, 1 Nov. 1995.
13. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.
14. F. L. Fessant. *The JoCaml reference manual*. INRIA Rocquencourt, 1998. Available from `http://join.inria.fr`.
15. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, Jan. 1996.
16. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, Aug. 26-29 1996. Springer-Verlag. LNCS 1119.
17. C. Fournet and L. Maranget. *The Join-Calculus Language*. INRIA Rocquencourt, 1997. Available from `http://join.inria.fr`.

18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
19. A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
20. P. B. Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.
21. C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 12(10), Oct. 74.
22. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
23. K. Honda and N. Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, Dec. 1993.
24. K. Jensen. *Coloured Petri Nets. Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1992.
25. P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966.
26. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and I. L. Levin. *Lisp 1.5 Programmer's Manual*. MIT Press, 1969.
27. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.
28. M. Odersky, D. Rabin, and P. Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.
29. C. 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.
30. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.
31. G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
32. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
33. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 293–305, June 1991.
34. A. Sabry and J. 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.
35. G. Smolka, M. Henz, and J. 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.
36. V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

This article was processed using the LaTeX macro package with LLNCS style