Part 4: Functional Nets and Join Calculus

Extended version of "Functional Nets", ESOP 2000, Berlin

What's a Functional Net?

- Functional nets arise out of a fusion of key ideas of functional programming and Petri nets.
- Functional programming: Rewrite-based semantics with function application as the fundamental computation step.
- Petri nets: Synchronization by waiting until all of a given set of inputs is present, where in our case

  input = function application.
- A functional net is a concurrent, higher-order functional program with a Petri-net style synchronization mechanism.
- Theoretical foundation: Join calculus.

Thesis of this Talk

Functional nets are a simple, intuitive model

of { imperative, functional, concurrent } programming.

Functional nets combine well with OOP.

Elements

Functional nets have as elements:

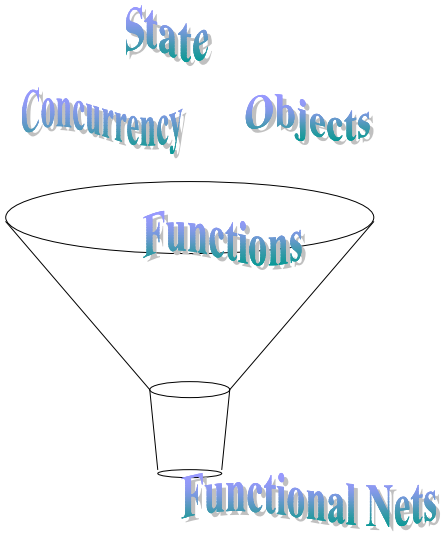
- functions
- objects
- parallel composition

They are presented here as a calculus and as a programming notation.

Calculus: (Object-based) join calculus

Notation: Funnel (alternatives are Join or JoCAML)

The Principle of a Funnel



Stage 1: Functions

- A simple function definition:

```
def gcd (x, y) =
  if (y == 0) x
  else gcd (y, x % y)
```

- Function definitions start with **def**.
- Operators as in C/Java.
- Usage:

```
val x = gcd (a, b)
print (x * x)
```

- Call-by-value: Function arguments and right-hand sides of **val** definitions are always evaluated.

Stage 2: Objects

- One often groups functions to form a single value. Example:

```
def makeRat (x, y) = {
  val g = gcd (x, y)
  {  def numer  =  x / g
     def denom  =  y / g
     def add r  =  makeRat (
                     numer * r.denom + r.numer * denom,
                     denom * r.denom)
     ...
  }
}
```

- This defines a record with functions numer, denom, add, ...
- We identify: Record = Object, Function = Method
- For convenience, we admit parameterless functions such as numer.

Functions + Objects Give Algebraic Types

- Functions + Records can encode algebraic types
  - Church Encoding
  - Visitor Pattern
- Example: Lists are represented as records with a single method, match.
- match takes as parameter a *visitor* record with two functions:

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

- match invokes the Nil method of its visitor if the List is empty, the Cons method if it is nonempty.

Lists

- Here is an example how match is used.

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

- It remains to explain how lists are constructed.

Lists

- Here is an example how match is used.

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

- It remains to explain how lists are constructed.
- We wrap definitions for Nil and Cons constructors in a List "module". They each have the appropriate implementation of match.

```
val List = {
  def Nil          =  { def match v = ??? }
  def Cons (x, xs) =  { def match v = ??? }
}
```

Lists

- Here is an example how match is used.

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

- It remains to explain how lists are constructed.

- We wrap definitions for Nil and Cons constructors in a List "module". They each have the appropriate implementation of match.

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


Stage 3: Concurrency

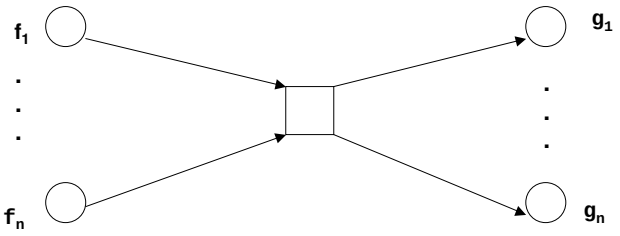
- Principle :
  - Function calls model *events*.
  - & means *conjunction* of events.
  - = means left-to-right rewriting.
  - & can appear on the right hand side of a = (*fork*) as well as on the left hand side (*join*).

- Analogy to Petri-Nets :

call	≈	place
equation	≈	transition

f_1 & ... & f = g_1 & ... & g_n

corresponds to



- Functional Nets are more powerful:
  - parameters,
  - nested definitions,
  - higher order.

Example : One-Place Buffer

Functions : put, get (external)
empty, full (internal)

Definitions :

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

Usage :

```
val x = get ; put (sqrt x)
```

- An equation can now define more than one function.
- Exercise: Write a Petri net modelling a one-place buffer.

Function Results

- In the rewrite rules for a one place buffer we still have to specify to which function call a result should be returned.
- Principle: In a rewrite rule wich joins n functions

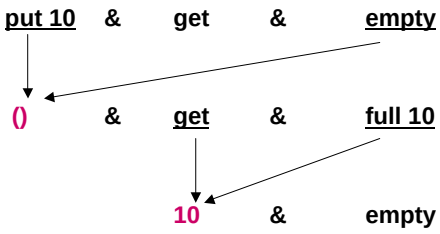
  f_1 & ... & f_n = E

  the result of E (if there is one) is returned to the first function f_1. All other functions do not return a result.
- We call functions which return a result *synchronous* and functions which don't *asynchronous*.
- It's also possible to have rewrite rules with only asynchronous functions. Example:

  def double & g x = g x & g x

Rewriting Semantics

- A set of calls which matches the left-hand side of an equation is replaced by the equation 's right-hand side (after formal parameters are replaced by actual parameters).
- Calls which do not match a left-hand side block until they form part of a set which does match.
- Example:



Objects and Joins

- We'd like to make a constructor function for one-place buffers.
- We could use tuples of methods:

```
def newBuffer = {
  def put x &  empty  =   () &  full x,
      get   &  full x   =   x  &  empty
  (put, get)  &  empty
}
val (bput, bget) = newBuffer ; ...
```

- But this quickly becomes combersome as number of methods grows.
- Usual record formation syntax is also not suitable
  - we need to hide function symbols
  - we need to call some functions as part of initialization.

Qualified Definitions

- Idea: Use qualified definitions:

```
def newBuffer = {
  def this.put x &  empty  =   () &  full x,
      this.get   &  full x   =   x  &  empty
  this  &  empty
}
val buf  =  newBuffer ; ...
```

- Three names are defined in the local definition:

this	- a record with two fields, get and put.
empty	- a function
full	- a function

- this is returned as result from newBuffer; empty and full are hidden.

- The choice of this as the name of the record was arbitrary; any other name would have done as well.
- We retain a conventional record definition syntax as an abbreviation, by inserting implicit prefixes. E.g.

```
{ def numer = x / g
  def denom = y / g }
```

is equivalent to

```
{ def r.numer = x / g, r.denom = y / g ; r }
```


Mutable State

- A *variable* (or reference cell) with functions

read, write	(external)
state	(internal)

is created by the following function:

```
def newRef init = {
  def this.read  & state x = x & state x,
      this.write y & state x = () & state y
  this & state init
}
```

- Usage:

```
val r = newRef 0 ; r.write (r.read + 1)
```

Control Structures

- Imperative control structures can be formulated as higher order functions.
- Example: while loop

```
while (cond) (body)  =
   if (cond ()) {
      body () ; while (cond) (body)
   } else {
      ()
   }
```

- Usage:

```
while (| i < N & !found) (| found := f (i) ; i := next (i))
```

- Exercise: Write functions that implement repeat and for loops.

Stateful Objects

- An object with *methods* $m_1,\ldots,m_n$ and *instance variables* $x_1,\ldots,x_k$ can be expressed such :

def this.m_1 & state $(x_1,\ldots,x_k)$ = ... ; state $(y_1,\ldots,y_k)$,
:
:
this.m_n & state $(x_1,\ldots,x_k)$ = ... ; state $(z_1,\ldots,z_k)$;

this & state $(init_1,\ldots, init_k)$

« Result » « initial state »

- The encoding enforces *mutual exclusion*, makes the object into a *monitor*.

Object Identity

- One often characterizes objects as having "state, behavior and identity".
- We model state with instance variables and behavior with methods, but what about identity?
- Question: Can we define an operation == such that for objects X, Y, X == Y is true iff X and Y are the same object (i.e. have been created by the same operation)?
- Need cooperation of the object.

Objects with Identity

- We want to define a method eq with one parameter, so that A == B can be implemented as A.eq(B).
- Idea: Make use of a boolean instance variable which is normally set to false. Then eq can be implemented by setting the variable to true and testing whether the *other* object's variable is also true.

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

      ... (other definitions) ...

  this  &  flag false
}
```

- Does this work in a setting where several threads run concurrently?

Synchronization

- Functional nets are very good at expressing many process synchronization techniques.
- Example: A semaphore (or: lock) offers two operations, getLock and releaseLock, which bracket a region which should be executed atomically.
- The getLock operation blocks until the lock is available. The releaseLock operation is asynchronous.
- This is implemented as follows:

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


Using Semaphores

- Semaphores can be used as follows:

```
lock = newLock
client1 = {
   ...
   lock.getLock ; ... /* critical region */ ... ; lock.releaseLock
   ...
}
client2 = {
   ...
   lock.getLock ; ... /* critical region */ ... ; lock.releaseLock
   ...
}
client1 & client2
```

- Problem: It's easy to forget a getLock or releaseLock operation ina client. Can you design a solution which passes a critical region to a single higher order function, sync?

Monitors

- A monitor is an object in which only one method can execute at any one time.
- This is easy to model as a functional net: Simply add an asynchronous function turn, which is consumed at each call and which is re-called after a method has executed:

```
def f  & turn =  ... ; turn
    g  & turn =  ... ; turn
```


Exercise: Bounded Buffer

- Let's implement a bounded buffer as a function net.
- Without taking overflow/underflow or concurrency into account, such a buffer could be written as follows:

```
def newBuffer (N) = {
  val elems = Array.new (N)
  var in := 0; var out := 0;
  def put (x) = {
    elems.put (in, x) ; in := (in + 1) % N ;
  }
  def get = {
    val x = elems.get (out) ; out := (out + 1) % N
    x
  }
}
```

- The parameter N indicates the buffer's size.

- This assumes arrays which are created with

  Array.new

  and which offer operations:

  get (index)
  put (index, value)

- Question: How can we modify newBuffer, so that
  - a buffer can be accessed by several processes running concurrently
  - A put operation blocks as long as the buffer is full.
  - A get operation blocks as long as the buffer is empty.

?


```
def newBuffer (N) = {
   val elems = Array.new (N)
   var in := 0; var out := 0; var n := 0
   def put (x)
      elems.put (in, x) ; in := (in + 1) % N

   }
   def get
      val x = elems.get (out) ; out := (out + 1) % N
      x
    }


}
```

Readers/Writers Synchronization.

- Readers/writers is a more refined synchronization technique.
- Specification: Implement operations startRead, startWrite, endRead, endWrite such that:
  - there can be multiple concurrent reads,
  - there can be only one write at one time,
  - reads and writes are mutually exclusive,
  - pending write requests have priority over pending reads, but don't preempt ongoing reads.

First Version

- Introduce two auxiliary state functions
  readers n - the number of *active* reads
  writers n - the number of *pending* writes
- Equations:

```
def startRead  & writers 0   =  startRead1,
    startRead1 & readers n   =  () & writers 0 & readers (n+1),

    startWrite  & writers n  =  startWrite1 & writers (n+1),
    startWrite1 & readers 0  =  (),

    endRead  & readers n     =  readers (n-1),
    endWrite & writers n     =  writers (n-1) & readers 0

readers 0 & writers 0
```

- Note the almost-symmetry between startRead and startWrite, which reflects the different priorities of readers and writers.

Final program

- The previous program was is not yet legal Funnel since it contained numeric patterns.
- We can get rid of value patterns by partitioning state functions.

```
def startRead  & noWriters    =  startRead1,
    startRead1 & noReaders    =  () & noWriters & readers 1,
    startRead1 & readers n    =  () & noWriters & readers (n+1),

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

    endRead   & readers n     =  if (n == 1) noReaders else (readers (n-1)),
    endWrite  & writers n     =  noReaders &
                                 ( if (n == 1) noWriters else writers (n-1) )
noWriters & noReaders
```


Summary : Concurrency

- Functional nets support an event-based model of concurrency.
- Channel based formalisms such as CCS, CSP or π - Calculus can be easily encoded.
- High-level synchronization à la Petri-nets.
- Takes work to map to instructions of hardware machines.
- Options:
  - Search patterns linearly for a matching one,
  - Construct finite state machine that recognizes patterns,
  - others?

Foundations

- We now develop a formal model of functional nets.
- The model is based on an adaptation of join calculus (Fournet & Gonthier 96)
- Two stages: sequential, concurrent.

A Calculus for Functions and Objects

- Name-passing, continuation passing calculus.
- Closely resembles intermediate language of FPL compilers.

Syntax:

Names	x, y, z		
Identifiers	i, j, k	::=	x | i.x
Terms	M, N	::=	i j | **def** D ; M
Definitions	D	::=	L = M | D, D | 0
Left-hand Sides	L	::=	i x

Reduction:

def D, i x = M ; ... i j ... → **def** D, i x = M ; ... [j/x] M ...

A Calculus for Functions and Objects

- The ... ... dots are made precise by a *reduction context*.
- Same as Felleisen's evaluation contexts but there's no evaluation here.

Syntax:

Names	x, y, z		
Identifiers	i, j, k	::=	x | i.x
Terms	M, N	::=	i j | **def** D ; M
Definitions	D	::=	L = M | D, D | 0
Left-hand Sides	L	::=	i x
Reduction Contexts	R	::=	[ ] | **def** D ; R

Reduction:

def D, i x = M ; R[i j] → **def** D, i x = M ; R[[j/x]M]

Structural Equivalence

- Alpha renaming: Local names may be consistently renamed as long as this does not introduce variable clashes.
- Comma is associative and commutative, with the empty definition 0 as identity

$$D_1, D_2 \equiv D_2, D_1$$
$$D_1, (D_2, D_3) \equiv (D_1, D_2), D_3$$
$$0, D \equiv D$$

Properties

- Name-passing calculus - every value is a (qualified) name.
- Contrast to lambda calculus, where values are lambda abstractions.
- Mutually recursive definitions are built in.
- Functions with results are encoded via a CPS transform (see paper).
- Value definitions can be encoded:

  val x = M ; N ≡ **def** k x = N ; k M

- Tuples can be encoded:

  f (i, j) ≡ (**def** ij.fst () = i, ij.snd () = j ; f ij)

  f (x, y) = M ≡ f xy = (**val** x = xy.fst () ; **val** y = xy.snd () ; M)

A Calculus for Functions, Objects and Concurrency

Syntax:

Names	x, y, z		
Identifiers	i, j, k	::=	x | i.x
Terms	M, N	::=	i j | **def** D ; M | M & M
Definitions	D	::=	L = M | D, D | 0
Left-hand Sides	L	::=	i x | L & L
Reduction Contexts	R	::=	[ ] | **def** D ; R | R & M | M & R

Reduction:

def D, i_1 x_1 & ... & i_n x_n = M ; R [i_1 j_1 & ... & i_n j_n]

→ **def** D, i_1 x_1 & ... & i_n x_n = M ; R [[j_1/x_1,...j_n/x_n] M]

Structural Equivalence

- Alpha renaming
- Comma is AC, with the empty definition 0 as identity:
- & is AC:

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

- Scope Extrusion:

$$(\textbf{def}\ D\ ;\ M)\ \&\ N \equiv \textbf{def}\ D\ ;\ M\ \&\ N$$

Relation to Join Calculus

- Strong connections to join calculus.
  - \- Polyadic functions
  - \+ Records, via qualified definitions and accesses.
- Formulated here as a rewrite system, whereas original join uses a reflexive CHAM.
- The two formulations are equivalent.

Continuation Passing Style

- Note that there is no term form which can represent a value. Hence, nothing can ever be returned from a join calculus expression.
- Instead, every "value-returning" function f is passed another function k as a parameter. k is called a *continuation* for f. The result of f is passed as a parameter to k.
- That is, instead of

```
def f () = 1 ; ... print (f ())
```

  one writes

```
def f (k) = k 1 ; ... f (print)
```

- This is called *continuation passing style* (in contrast to *direct style*).

One-Place Buffer in Continuation Passing Style

- Here is the one-place buffer in continuation passing style

```
def newBuffer k1 = (
  def this.put (x, k2) & empty  =  k2 ()  & full x ,
      this.get k3      & full x  =  k3 x  & empty ;
  k1 this  &  empty
)
```

- This formulation fits our syntax for object-based join calculus.
- Note that only functions which were synchronous in direct style get continuation parameters; asynchronous functions stay as they were.
- In a sense, the continuation argument represents a function's return address.

The Continuation Passing Transform

- Is it possible to map from direct style to continuation passing style?
- This is the task of a *continuation passing transform*.
- The transform takes programs written in direct style and maps them into equivalent programs written in continuation passing style.

Conclusions

- Functional nets provide a simple, intuitive way to think about functional, imperative, and concurrent programs.
- They are based on join calculus.
- Mix-and-match approach: functions (+objects) (+concurrency).
- Close connections to
  - sequential FP (a subset),
  - Petri-nets (another subset),
  - π-Calculus (can be encoded easily).
- Functional nets admit a simple expression of object-oriented concepts.

State ofWork

Done :

- Design of Funnel,
- experimental Hindley/Miler style type system,
- First, dynamically typed, implementation (available from `http://lampwww.epfl.ch`).

• Current :

- More powerful type System,
- Efficient compilation strategies,
- Encoding of objects
- Funnel as a composition language in a Java environment.

- Collaborators : Philippe Altherr, Matthias Zenger, Christoph Zenger (EPFL)
  Stewart Itzstein (Uni South Australia).