Functional Nets

Martin ODERSKY
EPFL
LAMP/DI

Ways to think about

Computers and Programs

Von Neumann Machines	…manipulate a store.
Calculators	…evaluate functions.
Rule Bases	…answer queries.

Problems :

- None of these covers all aspects of modern programming.
- None of these addresses *concurrency* well.

What about Object-Oriented Programming ?

- OOP is about structuring programs and about reuse.
  - Key-words: encapsulation, inheritance, subtyping.
  - Typically about larger, compound structures.
- It is orthogonal to the basic computation paradigm.
- It does not address by itself the issue of concurrency.
  - Concurrency handled by additional constructs such as monitors (e.g. Java) or rendez-vous (e.g. active objects).

Thesis of this Talk

Functional nets are a simple, intuitive model

of { imperative / functional / concurrent } programming

Functional nets combine well with OOP

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 define more than one function.
- & means parallel composition.

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 completely block until they form part of a set which does match.
- Example: Given a one place buffer:

```
put 10      &     get     &     empty
                    ⇓
  ()        &     get     &     full 10
                    ⇓
                  10      &     empty
```

Functional Programming

- FP is a special case, where & is not used.
- Example :

```
def map (f, xs) =
     if isEmpty xs then Nil
     else Cons (f (head xs), map (f, tail xs))
```

- Evaluation is call by value, as in Scheme or ML, unlike in Haskell.

Imperative Programming

- A variable (or reference cell) with functions

```
read, write   (external)
state         (internal)
```

  is defined by the following :

```
def read    &  state x  =  x  &  state x,
    write y &  state x  =  () &  state y
```

- This can be packaged in a function which creates a new variable every time it is called :

```
def newref init = {
    def read    & state x   =  x & state x,
        write y & state x   =  () & state y;
    (read, write) & state init
}
```

- (Note that definitions can nest)

Encoding Objects

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

```
def   m1 & state (x1,...,xn)  =  ... ; state (y1,...,yn),
                                 .
                                 .
      mn & state (x1,...,xn)  =  ... ; state (z1,...,zn);

(m1,...,mn) & state (init1,..., initn)
```

« Result » « initial state »

- Here ; is sequencing:

```
A;B   =      def k () = B ; k (A)
```

- The encoding enforces *mutual exclusion.*

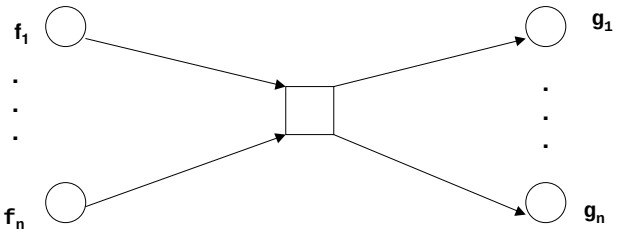
Dealing with Concurrency

- Principle :
  - Calls model *events.*
  - & means *conjunction* of events.
  - = means left-to-right rewriting.
- Analogy to Petri-Nets :

call	≈	place
equation	≈	transition

f_1 & ... & f_n = g_1 & ... & g_n

corresponds to



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

Example: Readers/Writers Synchronization

- 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 read.

First Version

```
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
```

- Not yet executable, because there are numeric patterns embedded in left-hand sides.

Final program

- Split state functions into interesting subsets :

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

    endRead     & readers n  =  if (n == 1) then noReaders
                                else (readers (n-1)),
    startWrite  & noWriters  =  startWrite1 & writers 1,
    startWrite  & writers n  =  startWrite1 & writers (n+1),
    startWrite1 & noReaders  =  (),

    endWrite    & writers n  =  noReaders &
                                (if (n == 1) then 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 current 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)

Sequential Subset:

Expression E = def D ; E ¦ $x(y_1,\ldots,y_n)$

Definition D = L = E

Left-hand side L = $x(y_1,\ldots,y_n)$

Reduction

def L = E ; … $\ominus$ L … $\rightarrow$ def L = E ; … $\ominus$ E …

where $\ominus$ maps the formal parameters of L to actual names.

- Example:

```
def f x = h (x, x) ; f 1
```

→

```
def f x = h (x, x) ; h (1, 1)
```

- More precise formulation of reduction :

Reduction context R = [ ] ¦ def D ; R

Reduction

def L = E ; R [⊖L] → def L = E ; R [⊖E]

Full Calculus with Concurrency

Expression	E	=	def D ; E ¦ $x(y_1,\ldots,y_n)$ ¦ E & E
Definition	D	=	L = E ¦ D, D
Left-hand side	L	=	$x(y_1,\ldots,y_n)$ ¦ L & L

Reduction

def D, L = E ; … ⊖ L … → def D, L = E ; … ⊖ E …

- Example:

```
def f x & g y = h (x, y) ; f 1 & g 2
```

→

```
def f x & g y = h (x, y) ; h (1, 2)
```

- More precise formulation of reduction :

Reduction context R = [ ] ¦ def D ; R ¦ R & E ¦ E & R

Reduction

def D, L = E ; R [⊖L] → def D, L = E ; R [⊖E]

Structural Equivalence

Some terms are considered equivalent (≡).

1. Consistent renaming of local variables yields an equivalent term (α- renaming)
2. & Over expressions is associative and commutative:

$$E_1 \,\&\, (E_2 \,\&\, E_3) \equiv (E_1 \,\&\, E_2) \,\&\, E_3$$
$$E_1 \,\&\, E_2 \equiv E_2 \,\&\, E_1$$

3. , over definitions is associative and commutative:

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

4. Scope extrusion law:

$$E_1 \,\&\, (\text{def } D ; E_2) \equiv \text{def } D ; (E_1 \,\&\, E_2)$$

if D does not define variables used in E_1.

Derived Constructs

- Join is a minimal calculus, but higher-level concepts can be *encoded.*
- Function-results are via a continuation passing transform. E.g.

```
def get & state x = x & state x
```

becomes

```
def get k & state x = k x & state x
```

- k is called a « continuation function ».
- Concept is well known from functional programming (e.g. Plotkin 76).

More Derived Constructs

- Local value definitions:

```
val x = f (y) ; g x
```

becomes

```
def k x = g x ; f (y, k)
```

- Nested Function Application:

```
f (g x)      becomes      val y = g x ; f y
```

- Sequence:

```
f ; g        becomes      val x = f ; g
```

etc.

Modelling Objects

- So far we have modelled objects with *tuples* of methods.
- This gets unwieldy as the number of methods increases.
- Needed: a lightweight way to represent an object with a *single* identifier.
- *Idea* : Introduce *qualified* names.
- Keep the principle that every term is represented by a (qualified) name. No anonymous values !

Object-Based Join Calculus

Field-Label	f, g		
Qualified Name	m , n	=	x ¦ m.f
Expression	E	=	m (m_1,…,m_n) ¦ def D ; E ¦ E & E
Definition	D	=	L = E ¦ D , D
Left-hand Side	L	=	m (x_1,…,x_n) ¦ L & L

Reduction

def D , L = E ; … ΘL… → def D, L = E ; … ΘE …

Example Rewriting

```
def newref (init) = {
    def r.get & state x   = x & state x,
        r.put y & state x = () & state y;
    r & state init
};
val R = newref 7 ; R.put 8 ; R.get
```

→... →

```
def newref (init) = ...
def r.get & state x   = x & state x,
    r.put y & state x = () & state y;
(r.put 8 ; r.get) & state 7
```

→

```
def...
def...
r.get & state 8
```

→

```
8 & state 8
```


Syntactic Sugar

Introduce a *record constructor* :

Definition D ::= … ¦ val y = D

To explain what this is, introduce qualified definitions, x.D, and left-hand sides, x.L and rewrite :

val x = D	⇒	x.D
x. (D_1, D_2)	⇒	x.D_1, x.D_2
x. (L = E)	⇒	(x.L) = E
x. (L_1 & L_2)	⇒	x.L_1 & x.L_2
x.(f(x_1,…,x_n))	⇒	x.f (x_1,…,x_n)

Example : Lists and Visitors

- We model lists as records with a single match method.
- match takes a ListVisitor as argument.
- A ListVisitor is a record with two cases, caseNil and caseCons.
- Example use :

```
def concat (xs, ys) = xs.match {
   def caseNil = ys
   def caseCons (x, xsl) =
           List.Cons (x, concat(xsl, ys))
}
```

- Analogous to algebraic datatypes in FP.

What 's needed to make this work ?

- A type for lists :

```
type List a = {
     match (v: ListVisitor a b)
}
```

- and one for List-Visitors :

```
type ListVisitor a b = {
     caseNil: b
     caseCons (x: a, xs: List a): b
}
```

The « List » Module

- is written as a record.
- is separated from the List type,

```
val List = {
   def Nil: List a = {
       def match v = v.caseNil
   }
   def Cons (x;xs): List a = {
       def match v = v.caseCons (x, xs)
   }
}
```


Conclusions

- Functional nets provide a simple, intuitive way to think about functional, imperative, and concurrent programs.
- Functional nets capture a single concept, they are not an amalgamation of several concepts.
- Close connections to
  - sequential FP (is a subset),
  - Petri-nets (are another subset),
  - π-Calculus (can be encoded easily).
- Functional nets admit a simple extension to OOP.

Finished and Current-Work

- Done :
  - Reduction Semantics
  - Hindley-Milner style type system
- Current :
  - Object-based extension of join calculus
  - More powerful type system
  - Compilation strategies
  - Implementation of a kernel language
  - Extension of Java with Join synchronisation

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