(basic)

Programming in π, Pict, and others

In TCS theme π-calculus
February 3, 2003
Claus Reinke

π: a claim to fame?

- *λ-calculus* (role-model)
  - formalises names (placeholder variables) and application of functional abstractions
  - basis for study of "sequential" languages (..)
- *π-calculus*
  - formalises names (channels/references) and communication between concurrent processes
  - basis for study of concurrent languages?

π-calculus: recap

$P = 0 \mid \Sigma_i P_i \mid \|_i P_i \mid !P \mid \nu(x).P \mid x(y).P \mid \underline{x}\langle y\rangle.P$

Σ,|| comm., assoc., with neutral 0

_(x).P binds x in P (capture-avoiding substitution, renaming,..)

ν(x).P ≡ P, if x∉ fn(P)

C[ν(x).P] ≡ ν(x).C[P], if x∉ fn(C)

$\Sigma\, x(y).R \parallel \Sigma\, \underline{x}\langle z\rangle.S \rightarrow R\{z/y\} \parallel S$

λ vs π

both: lexical scoping, capture-avoiding substitution

λ: *non-deterministic reduction, confluent*
- implicit communication channels
- implicit roles (function/parameter position)
- explicit term structure

π: *non-deterministic reaction, not confluent*
- explicit communication channels
- explicit roles (receiver/sender annotation)
- implicit term structure

λ → π (1)

$C[(\lambda y.R\ S)] \rightarrow C[R\{S/y\}]$

Explicitly label application sender/receiver channels

$C[({}_x\lambda y.R\ {}_x S)] \rightarrow C[R\{S/y\}]$

Scope channel names, separate send/receive

$C[\nu(x).(x\lambda y.R\ \underline{x}\langle S\rangle)] \rightarrow C[R\{S/y\}]$

Ignore now unused applicative structure

$C_{\nu(x)}[(x\lambda y.R \parallel \underline{x}\langle S\rangle)] \rightarrow C_{\nu(x)}[R\{S/y\}]$ (mod. congruence)

–explicit roles/channels
–implicit structure
–not confluent (multiple s/r)
–higher-order communication,
–no sender continuation
–no choice

π*: blue calculus

At this stage, we are close to Boudol's π* calculus

POPL'97: "The π-calculus in direct style", Gérard Boudol
- π*: includes π and λ as sub-calculi
- π* also distinguishes simple*"channel passing"* and higher-order *"resource fetching"*, as well as *single* and *replicated resources*.
- cf. earlier work by Boudol on *asynchronicity in π* and on *λ with resources* and work by Silvano Dal-Zilio on *typing and oop in π** (INRIA and MSR Cambridge)

Why is π not *"in direct style"*?

- so far, we have just relaxed restrictions of λ, enriching the calculus (we could go on by introducing sender continuations and choice)
- but π also introduces a new restriction of its own:

Restriction: send channels, but not processes

can still encode sending of processes in π:

- *process send*: replicate process, guarded by "trigger", then send trigger instead of process
- *process receive*: receive trigger, trigger process copies

x<P>.S ν(p).(x<p>.S || !p(t).P)
|| → ||
x(P).R[P] ν(t).(x(p).R[pt])

λ → π (2)

Embedding λ now requires CPS-style transformation

translation parameter is a context- (or continuation-)channel along which the translated "term" communicates with the context it is "embedded" in

$[[x]]_c = x\langle c\rangle$
$[[\lambda y.M]]_c = c(yc').[[M]]_{c'}$
$[[(R\ S)]]_c = \nu(rs).(r\langle sc\rangle \parallel [[R]]_r \parallel !s(c).[[S]]_c)$

Programming in Pict

Standard ML, Haskell, .. = ?

λ π

Core Pict, overview

- choice-free, asynchronous π-calculus plus some primitives, in ascii-fied syntax (slightly volatile..)
  - *booleans, characters, strings, integers*
  - *tuples, records, pattern-matching*
  - *asynchronous*: no continuation to output prefix
  - *choice-free*: but input-guarded choice as library
  - *replicated input* instead of free replication
  - no τ, no channel matching
- for simplicity, we ignore type system(s)

asynchrony

c<x>.S || c(y).R → S || {x/y}R *sync.comm.*
ν(s).(c<s> || s(r).(r<x> || S)) *async.*
|| c(s). ν(r).(s<r> || r(y).R) *handshake*

- from **Boudol, "Asynchrony and the π-calculus"**
- **but** see also **Catuscia Palamidessi, "Comparing the Expressive Power of the Synchronous and the Asynchronous pi-calculus"**

Core Pict, syntax

Val = Id | `[`Label Val .. `]` | `[`Val ..`]`
  | String | Char | Int | Bool
Pat = _ | Id | Id `@` Pat | `[`Label Pat ..`]` | `[`Pat ..`]`
Abs = Pat `=` Proc
Dec = `new` Id
Proc = Val `!` Val | Val `?` Abs | Val `?*` Abs
  | (Proc `|` Proc) | (Dec Proc)
  | `if` Val `then` Proc `else` Proc

Core Pict, semantics

congruence (comm/assoc/extrusion) as usual

reduction

pattern-matching, communication of built-in types

x!v|x?p = e → σe ,σp=v

x!v|x?*p = e → σe|x?*p = e ,σp=v

reduction under declarations and parallel composition, reduction modulo congruence, rules for conditional and operations on built-in types, as well as other built-in channels (print, ..)

core and derived syntax

```
cons?*[hd tl r] = (new l (r!l | l?*[n c] = c![hd tl]))
nil?*[r] = (new l (r!l | l?*[n c] = n![]))
(new r1 (nil![r1] | r1?e =
   (new r2 (cons![33 e r2] | r2?l = ..))))
(new n (new c (l![n c] | n?[] = e | c?[hd tl] = f)))
---------------------------------------------------------------
def cons (hd tl) = \[n c] = c![hd tl]
def nil () = \[n c] = n![]
val l = (cons 22 (cons 33 (cons 44 (nil))))
```

Pict, derived forms (1)

- nested declarations, run
  - $(d_1..d_n\ e) \Rightarrow (d_1..(d_n\ e)..)$
  - $(\text{run}\ e_1\ e_2) \Rightarrow (e_1 \mid e_2)$
- process abstractions
  - $(\text{def}\ x\ p = e_1\ e_2) \Rightarrow (\text{new}\ x\ (x?*p = e_1 \mid e_2))$
  - instantiated as (x!v)
  - also groups of mutually recursive process abstractions, separated by and

Pict, derived forms (2)

- complex values, via a CPS translation:
  - $[x \to c] = c!x$
  - $[k \to c] = c!k$
  - $[(d\ v) \to c] = (d\ [v \to c])$
  - $[[l_1\ v_1..] \to c] = (\text{new}\ c_1\ ([v_1 \to c_1] \mid c_1?x_1 = ..c![[l_1\ x_1..]]))..))$
  - $v_1!v_2 \Rightarrow (\text{new}\ c\ ([v \to c] \mid c?x = [v_2 \to x]))$
  - $v?a \Rightarrow (\text{new}\ c\ ([v \to c] \mid c?x = x?a))$
  - $v?*a \Rightarrow (\text{new}\ c\ ([v \to c] \mid c?x = x?*a))$
  - $(\text{val}\ p=v\ e) \Rightarrow (\text{new}\ c\ ([v \to c] \mid c?p = e)$

Pict, derived forms (3)

- application

  $[(v\ v_1..) \to c] = (\text{new}\ c'\ ([v \to c'] \mid c'?x = .. (\text{new}\ c_1\ ([v_1 \to c_1] \mid c_1?x_1 = ..\ x![x_1..c]))..))$
- abstractions
  - $(p_1..\ p_n) = v \Rightarrow [p_1..\ p_n\ r] = r!v$
  - \a => (def x a x)
  - example:

    def twice (f x) = (f (f x))

    val t = (twice \(x) = (+ x 1) 3)

Objects with locks (ref 1)

```
ref?*[init res] = (new contents new s new g
   (contents!init
   | (s?*[v c] = contents?_ = contents!v | c![])
   | (g?*[r] = contents?x = contents!x | r!x
   | res![set=s get=g] )))

(new r (ref![3 r] | r?myRef = ..))
```

Objects with locks (ref 2)

```
def ref [init res] =
   new contents
   run contents!init
   def s [v c] = contents?_ = contents!v | c![]
   def g [r] = contents?x = contents!x | r!x
   res![set=s get=g]
```

Objects with locks (ref 3)

```
def ref [init res] =
   new contents
   run contents!init
   res![set= \[v c] = contents?_ = contents!v | c![]
         get= \[r] = contents?x = contents!x | r!x
        ]
```

Objects with locks (ref 4)

```
def ref (init) =
   new contents
   run contents!init
   [set= \[v c] = contents?_ = contents!v | c![]
    get= \[r] = contents?x = contents!x | r!x
   ]

(val [get set]  = (ref 0) (val v1 = (get) (val _ = (set 5)
 print!(r.get) )))
```

Objects with locks (nullRef)

```
def nullRef () =
   new contents
   new set
   run set?[v c] = contents!v | c![]
   run set?*[v c] = contents?_ = contents!v | c![]
   [set=set
    get= \[r] = contents?x = contents!x | r!x
   ]
```

Objects with locks (clearRef)

```
def clearRef () = new contents new clear new init
   def server [] =
         init?[v c] = contents!v | c![]
        |clear?[c] = contents?_ = c![] | server![]
   run server![]
   [set=\[v c] = contents?_ = contents!v | c![]
    get= \[r] = contents?x = contents!x | r!x
    clear=clear
    init=init
   ]
```

Ref with choice

```
def ref (init) =
   new set new get
   def server x =
        sync!(
              (get => \[r] = r!x | server!x)
        $     (set => \[v c] = c![] | server!v)
              )
   run server!init
   [set= set    get= get]
```

Implementing choice

```
def newLock () = new lock
   run lock?[r] = r!true | (lock?*[r] = r!false)
   lock
def ($)(e1 e2) = \lock = e1!lock | e2!lock
def sync e= e!(newLock)
def (=>)(c receiver) =
   \lock = c?v = if (lock)
                 then receiver!v
                 else c!v
```

Multi-State Objects (clearRef 2)

```
def clearRef () = new set get clear init
   def empty [] =
       sync!( init => \[v c] = full!v | c![] )
   and full x =
       sync!( set => \[v c] = full!v | c![]
              $ get => \[r] = full!x | r!x
              $ clear => \[c] = empty![] | c![]
              )
   run empty![]
   [set=set get=get clear=clear init=init]
```

π/Pict for programming?

- π
  - concurrent machine-code
  - could be the basis for different high-level PLs, but some encodings are indirect
  - lacks support for distributed programming → variants
- Pict
  - mixture of programming with functions and concurrent objects (groups of agents that collaborate to provide set of services to outside world, jointly maintaining consistency of shared data)
  - substantial libraries; type system
  - fp efficiency slightly behind cps-style compilers (SML)
  - communication efficiency slightly ahead (CML)

main references

- Pierce and Turner, *"Concurrent Objects in a Process Calculus"* (LNCS 907, 1995)
- Pierce and Turner, *"Pict: A Programming Language Based on the Pi-Calculus"* (Indiana TR, March 1997)
- Pict home page
  http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html
- Robin Milner, *"communicating and mobile systems: the π-calculus"* (CUP 1999)

What else?

- type systems for π/Pict [various]
- work by Boudol (including *blue calculus, atomic actions, asynchrony, resources*) [INRIA]
- POPL'96: *"The reflexive CHAM and the join-calculus"*, Cédric Fournet and Georges Gonthier [INRIA]
- nomadic Pict, ambient calculus, ..
- my favourite: *linear logic, concurrent calculi and Petri nets* [various]

λ → π (2)

Restriction: send channels, but not processes

Embedding λ now requires CPS-style transformation

- "put aside" complex expressions, replacing (representing) them by channels along which they will send channels representing their results
- abstractions get parameter-channel in addition to result-channel, and are replicated to account for non-linear substitutions

$[[(R\ S)]]_c = \nu(rs).(r\lambda(r).\underline{r}\langle sc\rangle \parallel [[R]]_r \parallel [[S]]_s)$

$[[\lambda y.M]]_c = \nu(x).(\underline{c}x \parallel !x\lambda(yr).[[M]]_r)$

$[[x]]_c = \underline{c}x$