Pies, Tins, and Calculi

Clarissa Littler

March 17, 2016

Outline

This Talk

- A brief introduction to the π calculus

This Talk

- A brief introduction to the π calculus
- Milner's translation of λ into π

This Talk

- A brief introduction to the π calculus
- Milner's translation of λ into π
- An enriched π calculus

This Talk

- A brief introduction to the π calculus
- Milner's translation of λ into π
- An enriched π calculus
- The Tin language

This Talk

- A brief introduction to the π calculus
- Milner's translation of λ into π
- An enriched π calculus
- The Tin language
- A translation from Tin to our enriched π

A Universal Model of Computation

- Computation is independent of any description language

A Universal Model of Computation

- Computation is independent of any description language
- Turing complete

A Universal Model of Computation

- Computation is independent of any description language
- Turing complete
- λ calculus

A Universal Model of Computation

- Computation is independent of any description language
- Turing complete
- λ calculus
- Real world has concurrency

A Universal Model of Computation

- Computation is independent of any description language
- Turing complete
- λ calculus
- Real world has concurrency
- Turing machines and λ calculi don't

Pi Calculus

- The π calculus was introduced by Milner et al. in '92

Pi Calculus

- The π calculus was introduced by Milner et al. in '92
- Calculus of communicating systems

Pi Calculus

- The π calculus was introduced by Milner et al. in '92
- Calculus of communicating systems
- Processes and channels, not functions and arguments

Pi Calculus

- The π calculus was introduced by Milner et al. in '92
- Calculus of communicating systems
- Processes and channels, not functions and arguments
- Simple syntax

Pi Calculus

$$
\begin{aligned}
P :=& \\
&(\nu x.P) \\
&P|Q \\
&x(y).P \\
&\overline{x}(y).P \\
&\perp
\end{aligned}
$$

Pure Pi Calculus

- Only data are channels

Pure Pi Calculus

- Only data are channels
- Synchronous communication

Pure Pi Calculus

- Only data are channels
- Synchronous communication
- A universal model of computation?

Pure Pi Calculus

- Only data are channels
- Synchronous communication
- A universal model of computation?
- Milner provided an embedding of λ

Lambda to Pi

The basic idea is that

- Variables send the result along the channel named by the variable
- Abstractions
  - receive the argument
  - receive the destination to send the result
  - run the body of the function with those connections
- Application
  - Runs the argument and function in parallel
  - Connects them with fresh channels
  - Replicates the argument in case it's used in multiple places

Lambda To Pi

$$
\begin{aligned}
[x]u &= \overline{x}(u).\ \perp \\
[\lambda x.M]u &= u(x).u(f).[M](f) \\
[MN](u) &= \nu c.\nu d.([M](c)|\overline{c}(d).\overline{c}(u)|!d(v).[N](v))
\end{aligned}
$$

In This Project

- `PureLam.hs`
- `PurePi.hs`
- `LamToPi.hs`

Lambda Calculus

```
data Lam = Abs Name Lam
 | App Lam Lam
 | Var Name
 | Print Name

data Val = VUnit
 | VAbs Name Lam
```

Pi Calculus

```
type Name = String

type Value = Name

data Proc = Receive Name Name Proc
  | Send Name Name Proc
  | Par Proc Proc
  | Nu Name Proc
  | Serv Proc
  | Print Name
  | Terminate
```

Pi Calculus

```
interpProc :: Proc -> Interp ()
interpProc (Print n) = putText n
interpProc Terminate = return ()
interpProc (Par p1 p2) = do
  forkM $ interpProc p1
  interpProc p2
interpProc (Serv p) = do
  forkM $ interpProc p
  interpProc (Serv p)
interpProc (Nu n p) = do
  m <- liftIO newEmptyMVar
  withChan n m $ interpProc p
```

Pi Calculus

```
interpProc (Send x y p) = do
  env <- asks fst
  case lookup x env of
    Nothing -> error "channel doesn't exist"
    Just m -> (liftIO $ putMVar m y) >> interpProc p
interpProc (Receive x y p) = do
  ec <- asks fst
  case lookup x ec of
    Nothing -> error "channel doesn't exist"
    Just m -> do
      v <- liftIO $ takeMVar m
      interpProc $ substName y v p
```

Lambda To Pi

```
transLam (L.Print n) _ = return $ P.Print n
transLam (L.Var x) n = return $ P.Send x n P.Terminate
transLam (L.Abs x b) n = do
  u <- fresh
  b' <- transLam b u
  return $ P.Receive n x $ P.Receive n u $ b'
transLam (L.App f a) n = do
  c <- fresh
  d <- fresh
  v <- fresh
  f' <- transLam f c
  a' <- transLam a v
  return $ P.Nu c $ P.Nu d $ P.Par (P.Par f'
       (P.Send c d $ P.Send c n $ P.Terminate))
       (P.Serv $ P.Receive d v a')
```

Richer Languages

- Both λ and π calculus are sparse

Richer Languages

- Both λ and π calculus are sparse
- Hard to program in

Richer Languages

- Both λ and π calculus are sparse
- Hard to program in
- Wanted to write something higher level

Common Expression Language

```
data Exp = EBinOp Exp Op Exp
  | EUnOp Op Exp
  | EInt Int
  | EBool Bool
  | EString String
  | EVar Var
  | EUnit
  | EPrint Exp
  | EName Name

data Val = VInt Int
  | VString String
  | VBool Bool
  | VUnit
  | VName Name
```

Enriched Pi Calculus

- Ordinary π calculus is sparse

Enriched Pi Calculus

- Ordinary π calculus is sparse
- Enriched with expressions and data

Enriched Pi Calculus

```
data Proc = Receive Exp Name Proc
  | Send Exp Exp Proc
  | Par Proc Proc
  | Nu Name Proc
  | Serv Proc
  | If Exp Proc Proc
  | Terminate
```

Tin: An Imperative Concurrent Language

- Slightly higher level

Tin: An Imperative Concurrent Language

- Slightly higher level
- Imperative language

Tin: An Imperative Concurrent Language

- Slightly higher level
- Imperative language
  - while loops
  - if statements
  - sequenced code

Tin: Concurrency Model

- Each declaration is a process

Tin: Concurrency Model

- Each declaration is a process
- Each process runs in parallel

Tin: Concurrency Model

- Each declaration is a process
- Each process runs in parallel
- Each process can receive or send messages

Tin: Concurrency Model

- Each declaration is a process
- Each process runs in parallel
- Each process can receive or send messages
- Sending and receiving is a blocking action

Tin: Concurrency Model

- Each declaration is a process
- Each process runs in parallel
- Each process can receive or send messages
- Sending and receiving is a blocking action
- No primitive locking

Tin AST

```
data Stmt = SExp Exp
  | SReceive [Var]
  | SSend Exp [Exp]
  | SWhile Exp [Stmt]
  | SIf Exp [Stmt] [Stmt]

data Decl = Decl Name [Stmt]

type Inbox = Chan Val

data InterpEnv = IE { inboxes :: NEnv Inbox, -- inboxes
      venv :: VEnv Val, -- value env
      outc :: Chan String, -- output queue
      self :: Name}

type Interp = StateT InterpEnv IO
```

Fibonacci in Tin

```
f := {
       send @h (1)
       while (true) do {
       receive (x)
       send @h (x)
       }
      }

g := {
       send @h (1)
       while (true) do {
       receive (x)
       send @h (x)
       }
      }
```

```
h := {
       receive (x,y)
       while ((x < 100000)) do
        print ((x + y))
        send @f (y)
        send @g ((x + y))
        receive (x,y)
       }
      }
```

Tin to Enriched Pi

```
progToPi :: [T.Decl] -> P.Proc

blockToProc :: String -> [T.Stmt] -> P.Proc -> Fresher P.Proc

stmtToProc :: String -> T.Stmt -> P.Proc -> Fresher P.Proc
stmtToProc n (T.SWhile e ss) p = do
  sp <- blockToProc n ss P.Terminate
  conn <- fresh
  dummy <- fresh
  return $ P.Par (P.Serv $ P.If e sp
    (P.Send (EName conn) EUnit P.Terminate))
    (P.Receive (EName conn) dummy p)
stmtToProc n (T.SIf e sts sfs) p = do
  tp <- blockToProc n sts p
  fp <- blockToProc n sfs p
  return $ P.If e tp fp
```

Questions?