Communicating Haskell Processes

Neil Brown

Computing Laboratory
University of Kent
UK

9 September 2008

University of Kent | Computing

< 30 mins

University of Kent | Computing

Communicating Haskell Processes (CHP)

Haskell library for process-oriented programming, in the tradition of libraries such as:

- CTJ/JCSP (1997)
- CCSP (1999)
- C++CSP (2003)
- CSP for .NET (2006)
- PyCSP (2007)
- CSO (2008)

Relevant Haskell Features

- Pure language
  - No hidden side-effects, referentially transparent
- Functional
- Imperative
- Statically typed

University of Kent | Computing

Basic CHP Features

- Channels, usable via reading/writing ends
  - Shared channels with explicitly claimed ends
- Barriers, with dynamic enrollment
- Poison on channels and barriers
- Concurrency
- User-threads, load balanced across kernel threads by the Haskell run-time

Identity Process

```
idProcess :: forall a. Chanin a -> Chanout a -> CHP ()
```

For any *type* a, idProcess takes an incoming channel carrying a, an outgoing channel carrying type a, and yields a CHP process that returns nothing.

Identity Process

```
idProcess :: forall a. Chanin a -> Chanout a -> CHP ()

idProcess input output
  = forever  (do x <- readChannel input
                 writeChannel output x)
```

University of Kent | Computing

Delta2 Process

```
delta2Process ::
   forall a. Chanin a -> Chanout a -> Chanout a -> CHP ()

delta2Process input output0 output1
   = forever (do x <- readChannel input
                 writeChannel output0 x <||> writeChannel output1 x)
```

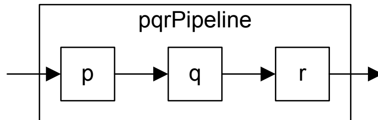
Delta Process

```
deltaProcess :: forall a. Chanin a -> [Chanout a] -> CHP ()

deltaProcess input outputs
  = forever (do x <- readChannel input
                runParallel [writeChannel c x | c ∈ outputs])
```

$<$ 25 mins

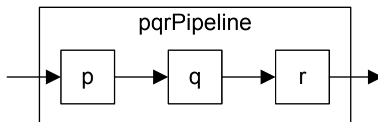
Process Pipelines

```
type InOut a = Chanin a -> Chanout a -> CHP ()

p, q, r, pqrPipeline :: InOut Int
```



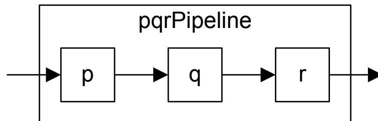
Process Pipelines

```
pqrPipeline :: InOut Int

pqrPipeline input output
  = do c <- oneToOneChannel
       d <- oneToOneChannel
       p input (writer c)
         <||> q (reader c) (writer d)
         <||> r (reader d) output
```



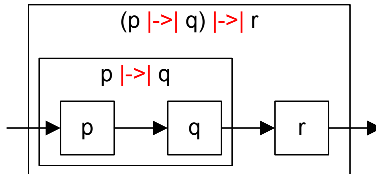
Process Pipelines

```
pqrPipeline :: InOut Int

pqrPipeline = p |->| q |->| r
```



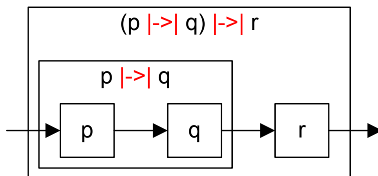
Process Pipelines

```
pqrPipeline :: InOut Int

pqrPipeline = p |->| q |->| r
```



Process Pipelines

```
(|->|) :: forall a. InOut a -> InOut a -> InOut a

(|->|) p q = λ input output ->
                do c <- oneToOneChannel
                   p input (writer c) <||> q (reader c) output
```



Process Pipelines

```
type InOut' a b = Chanin a -> Chanout b -> CHP ()

(|->|) :: forall a b c. InOut' a b -> InOut' b c -> InOut' a c

(|->|) p q = λ input output ->
                 do c <- oneToOneChannel
                    p input (writer c) <||> q (reader c) output
```

University of Kent | Computing

Haskell Higher-Order Functions

```
map :: forall a b. (a -> b) -> [a] -> [b]
filter :: forall a. (a -> Bool) -> [a] -> [a]

map negate (filter isEven [0,1,2,3,4]) == [0,-2,-4]
```

University of Kent | Computing

Haskell Higher-Order Functions

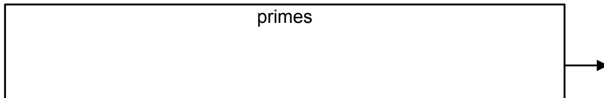
```
mapProcess ::
  forall a b. (a -> b) -> Chanin a -> Chanout b -> CHP ()
filterProcess ::
  forall a. (a -> Bool) -> Chanin a -> Chanout a -> CHP ()

filterProcess isEven |->| mapProcess negate
```

University of Kent Computing

< 20 mins

Example – Primes using Trial Division

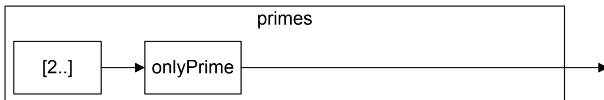
```
primes :: Chanout Integer -> CHP ()
```



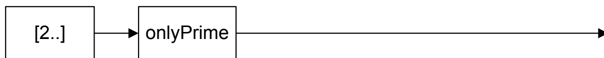
University of Kent | Computing

Example – Primes using Trial Division

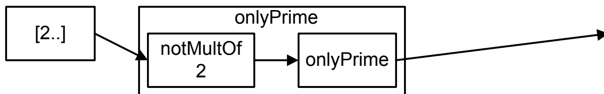
```
primes :: Chanout Integer -> CHP ()
primes output
  = do c <- oneToOneChannel
       writeNums outCh 2
         <||> onlyPrime (reader c) output
  where
    writeNums outCh n = do writeChannel outCh n
                           writeNums outCh (n + 1)
```



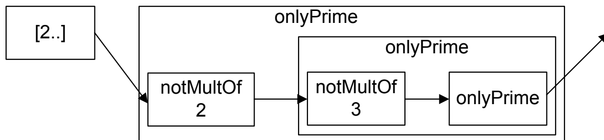
Example – Primes using Trial Division



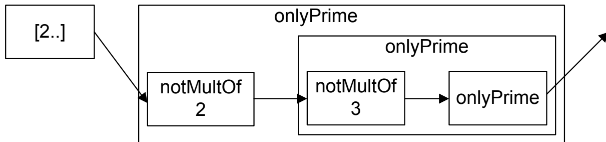
Example – Primes using Trial Division



Example – Primes using Trial Division



Example – Primes using Trial Division




```
onlyPrime input output
  = do x <- readChannel input
       writeChannel output x
       c <- oneToOneChannel
       filterProcess (notMultOf x) input (writer c)
         <||> onlyPrime (reader c) output
```

University of Kent | Computing

Choice

- There is choice on:
  1. Channel reads (input guards)
  2. Channel writes (output guards)
  3. Barrier synchronisations (alting barriers)
  4. SKIP (always-ready), STOP (never-ready)

Choice

- There is choice on:
  1. Channel reads (input guards)
  2. Channel writes (output guards)
  3. Barrier synchronisations (alting barriers)
  4. SKIP (always-ready), STOP (never-ready)
  5. Conjunctions of any of 1–4

Choice

- There is choice on:
  1. Channel reads (input guards)
  2. Channel writes (output guards)
  3. Barrier synchronisations (alting barriers)
  4. SKIP (always-ready), STOP (never-ready)
  5. Conjunctions of any of 1–4
  6. Timeouts
  7. Choices (nested alts)

Choice

- There is choice on:
  1. Channel reads (input guards)
  2. Channel writes (output guards)
  3. Barrier synchronisations (alting barriers)
  4. SKIP (always-ready), STOP (never-ready)
  5. Conjunctions of any of 1–4
  6. Timeouts
  7. Choices (nested alts)
- But no priority between 1–3 and 5

$<$ 15 mins

Example Use – Overwriting Buffer

- An overwriting buffer process

```
overwritingOnePlaceBuffer :: forall a. InOut a
overwritingOnePlaceBuffer input output = readNew
  where
    readNew = do x <- readChannel input
                 full  x

    full  x = ???
```

University of Kent | Computing

Example Use – Overwriting Buffer

- An overwriting buffer process

```
overwritingOnePlaceBuffer :: forall a. InOut a
overwritingOnePlaceBuffer input output = read
  where
    readNew = do x <- readChannel input
                 full x

     full x = alt [do y <- readChannel input
                      full y
                  ,do writeChannel output x
                      readNew]
```

University of Kent | Computing

Example Use – Overwriting Buffer

- An overwriting buffer process

```
overwritingOnePlaceBuffer :: forall a. InOut a
overwritingOnePlaceBuffer input output = read
  where
    readNew = do x <- readChannel input
                 full  x

     full  x =  alt  [do y <- readChannel input
                         full  y
                     ,do writeChannel output x
                         readNew]
```

Example Use – Overwriting Buffer

- An overwriting buffer process

```
overwritingOnePlaceBuffer :: forall a. InOut a
overwritingOnePlaceBuffer input output = read
  where
    readNew = do x <- readChannel input
                 full  x

     full  x = alt  [readNew
                    ,do writeChannel output x
                        readNew]
```

University of Kent Computing

Example Use – Overwriting Buffer

- An overwriting buffer process

```
overwritingOnePlaceBuffer :: forall a. InOut a
overwritingOnePlaceBuffer input output = read
  where
    readNew = do x <- readChannel input
                 full  x

    full  x =  alt  [readNew
                    ,do writeChannel output x
                        readNew]
```

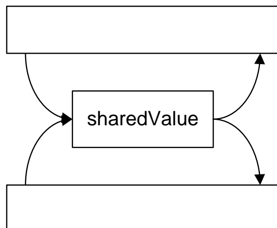
University of Kent Computing

Example Use – Shared Variables

```
valueStore :: forall a. InOut a
valueStore input output = readNew
  where
    readNew = do x <- readChannel input
                 full x

     full x = alt [readNew
                  ,do writeChannel output x
                      full x]
```

University of Kent | Computing

Shared Variables Gone Bad




```
update = do x <– claim input readChannel
            claim output ( flip  writeChannel (x + 2))

update <| |> update
```

Example Use – Shared Variables

```
data Update a = NewValue a | ModifyWith (a -> a)

valueStore' :: forall a. Chanin (Update a) -> Chanout a -> CHP ()
valueStore' input output
  = do NewValue x <- readChannel input
       full x
  where
    full x = alt [do req <- readChannel input
                     case req of
                       NewValue y -> full y
                       ModifyWith f -> full (f y)
                 ,do writeChannel output x
                     full x]
```

University of Kent Computing

Other Features

- Broadcast and gather channels
- Conjunction (from fringe presentation)
- Tracing (from yesterday)

Implementation

Channels and barriers are based on an “event” type:

- Events support poisoning and choice
- Event synchronisations are done using an oracle-like system (but with added support for conjunction)...
- ...which is built using Software Transactional Memory (STM)

Paper available on request

Conclusions

CHP shows us that:

- Recursion can be useful in process-oriented programming
- Output guards open up some new, easy design patterns
- Higher-order processes make wiring and utility processes easy

< 10 mins

($<$ 5 mins if I over-ran. . .)

University of Kent | Computing