Implementing Choice Using Transactional Memory

Neil Brown

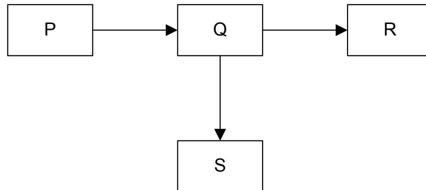
Computing Laboratory
University of Kent
UK

17 October 2008

University of Kent | Computing

Process-Oriented Programming

- No shared mutable data
- Processes interact via synchronous point-to-point communication channels



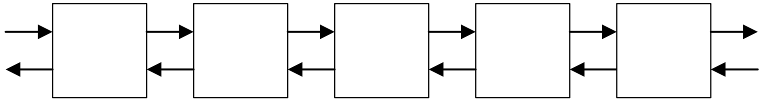
- Inherited from CSP and occam

University of Kent | Computing

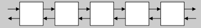
Outline

1. Different ways of waiting for a combination of events:
   - Disjunction (a OR b)
   - Conjunction (a AND b)
   - Disjunction of conjunctions (a OR (b AND c))
2. Their uses
3. How to implement them

1D Example

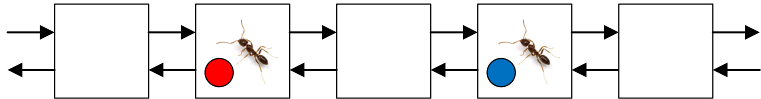


1D Example

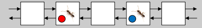


We can represent discrete space (here: one dimensional) by having active processes as space cells, each connected to their neighbours via a pair of channels.

1D Example

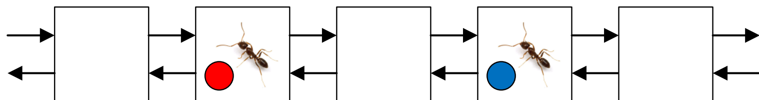


University of Kent | Computing



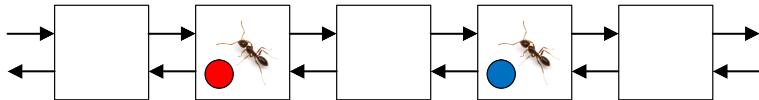
We can then add agents to wander around this space. In this example our agents (ants) are not processes, merely pieces of data.

1D Ants Example



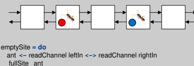
```
fullSite  ant = do
  dir <- pickRandomDirection
  case dir of
    Left  -> writeChannel leftOut ant
    Right -> writeChannel rightOut ant
  emptySite
```

1D Ants Example



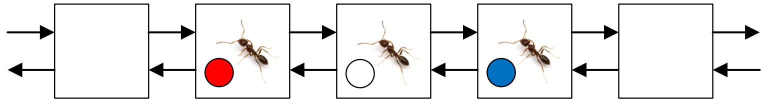
```
emptySite = do
   ant <- readChannel leftIn <-> readChannel rightIn
    fullSite   ant
```

University of Kent | Computing



The empty site performs a choice (the `<->` operator) between reading an ant from the left channel or the right channel. It will wait until it performs exactly one of these actions; we only allow one ant per cell, so it is important we do not accidentally read two ants at the same time.

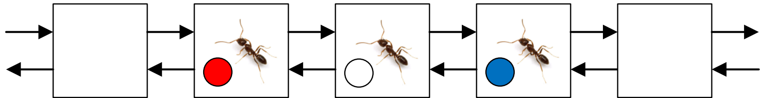
1D Ants Example – Swapping



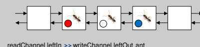
University of Kent | Computing

With several ants, we will begin to have problems because ants cannot move past each other. If two ants try to move into each other, they will deadlock! So we want to allow a swap to take place if ants do try to move into each other, both removing the deadlock and allowing them to pass each other. But how to implement this?

1D Ants Example – Swapping

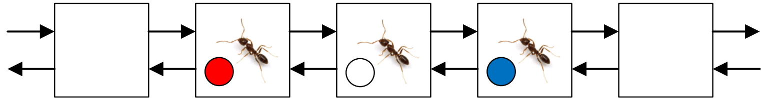


readChannel leftIn >> writeChannel leftOut ant



This sequence (read in, then send out) will not work. Consider if the space with the blue ant reads in the white ant and then tries to send out the blue ant; in the mean-time, the red ant could have moved into the middle space, and thus one cell then contains two ants.

1D Ants Example – Swapping

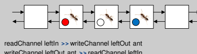


readChannel leftIn >> writeChannel leftOut ant

writeChannel leftOut ant >> readChannel leftIn

University of Kent | Computing

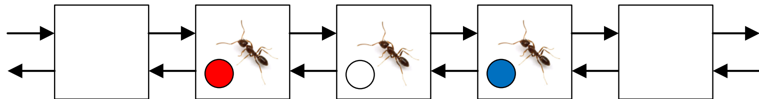
1D Ants Example – Swapping



readChannel leftIn >> writeChannel leftOut ant
writeChannel leftOut ant >> readChannel leftIn

We cannot use this sequence either (send then read) because if both sides first send then they will deadlock waiting for the read (this also applied to our first sequence, in fact).

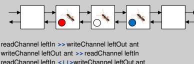
1D Ants Example – Swapping



```
readChannel leftIn >> writeChannel leftOut ant
writeChannel leftOut ant >> readChannel leftIn
readChannel leftIn <||>writeChannel leftOut ant
```

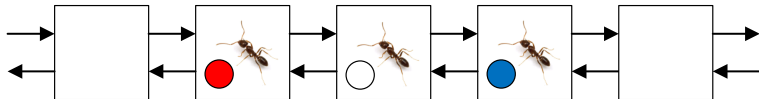
University of Kent | Computing

1D Ants Example – Swapping



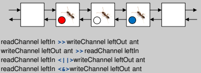
Placing the write and read in parallel solves the deadlock, but still does not prevent the first problem of the read ant “stealing in”.

1D Ants Example – Swapping



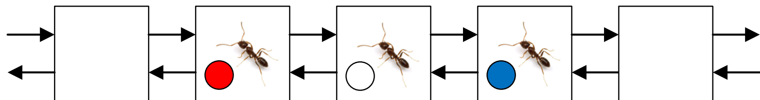
```
readChannel leftIn >> writeChannel leftOut ant
writeChannel leftOut ant >> readChannel leftIn
readChannel leftIn <||>writeChannel leftOut ant
readChannel leftIn <&>writeChannel leftOut ant
```

University of Kent | Computing



We must use conjunction. This operator performs both events as an indivisible action; the process will only perform all the conjoined events together, never just one of them. So the swap will happen as one action, without deadlock and without the possibility of the red ant moving in the middle of the swap.

1D Ants Example



```
fullSite  ant = do
  dir  <- pickRandomDirection
  maybeAnt <- case dir of
     Left  -> (writeChannel leftOut ant >> return Nothing)
                ...
    ...
  maybe emptySite fullSite maybeAnt
```

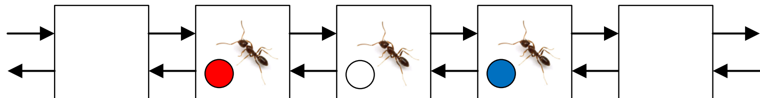
University of Kent | Computing

1D Ants Example

```
fullSite  ant = do
  dir  <- pickRandomDirection
  maybeAnt <- case dir of
    Left  -> (writeChannel leftOut ant >> return Nothing)
          ...
    ...
  maybe emptySite fullSite maybeAnt
```

We must still offer the basic send – we do not want to only swap, as there may not be an ant in the space. So we offer to just write (to move into an empty space) or to swap (see the next slide). These choices overlap, but that is fine; the choice is determined by our neighbour.

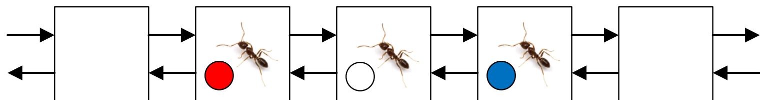
1D Ants Example



```
fullSite  ant  = do
  dir  <- pickRandomDirection
  maybeAnt <- case dir of
     Left  -> (writeChannel leftOut ant >> return Nothing)
                <-> (f  (readChannel leftIn <&> writeChannel leftOut ant))
    . . .
  maybe emptySite fullSite maybeAnt
  where f =  liftM  (Just  .  fst )
```

University of Kent | Computing

1D Ants Example



```
fullSite   ant  = do
  dir  <- pickRandomDirection
  maybeAnt <- case dir of
     Left  -> (writeChannel leftOut  ant  >> return Nothing)
              <-> (f  (readChannel leftIn  <&> writeChannel leftOut ant))
     Right -> (writeChannel rightOut ant >> return Nothing)
              <-> (f  (readChannel rightIn <&> writeChannel rightOut ant))
  maybe emptySite fullSite maybeAnt
  where f =  liftM  (Just  .  fst )
```

University of Kent | Computing

Choice

Each event has an enrollment count, n of processes; all n must synchronise together. Each process can offer:

- Disjunction of events: a OR b OR c
  - Wait for any event to be ready, then engage in exactly one
- Conjunction of events: a AND b AND c
  - Wait for all of the events to be ready, then engage in all of them
- Disjunction of conjunctions: a OR (b AND c)
  - (disjunctive normal form)

Dining Philosophers

Dining Philosophers

```
philosopher = forever ( do
   randomDelay -- Thinking
   writeChannel leftFork () <&> writeChannel rightFork ()
   randomDelay -- Eating
   writeChannel leftFork () <||> writeChannel rightFork () )
```

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

You can see that to resolve person 1's choice, we must examine person 2's choices, which then depend on person 3. So to resolve for person 1, we must examine person 3, even though person 1 and person 3 do not share any events.

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

Person 2: $(a \wedge z) \vee (b \wedge y) \vee (c \wedge x) \vee (d \wedge w)$

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

Person 2: $(a \wedge z) \vee (b \wedge y) \vee (c \wedge x) \vee (d \wedge w)$

You can see that to resolve person 1's choice, we must examine person 2's choices, which then depend on person 3. So to resolve for person 1, we must examine person 3, even though person 1 and person 3 do not share any events.

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

Person 2: $(a \wedge z) \vee (b \wedge y) \vee (c \wedge x) \vee (d \wedge w)$

Person 3: $j \vee q \vee x \vee z$

Resolving Choice

Person 1: $(a \wedge e) \vee b \vee c \vee d \vee f$

Person 2: $(a \wedge z) \vee (b \wedge y) \vee (c \wedge x) \vee (d \wedge w)$

Person 3: $j \vee q \vee x \vee z$

You can see that to resolve person 1's choice, we must examine person 2's choices, which then depend on person 3. So to resolve for person 1, we must examine person 3, even though person 1 and person 3 do not share any events.

Implementing Choice

- Previous solution used a global data structure with One Big Lock*TM*
- We instead use Transactional Memory
  - One transactional variable per event/channel
- No busy-waiting, no active search – sleep while waiting

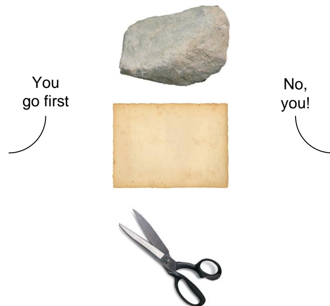
Implementing Choice

- Can’t just use one transaction each

University of Kent Computing

Implementing Choice

- Can't just use one transaction each

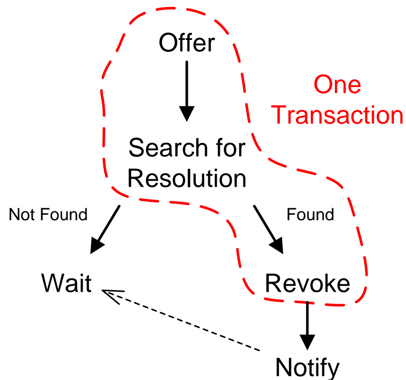


Implementing Choice

- Can't just use one transaction each

Our choice bears some relation to transactions, in that transactions can express conjunction (by putting multiple things in a transaction) and disjunction (by using choice between transactions). But we have a rock-paper-scissors problem; everyone's choice depends on everyone else's, so they all wait for the others to decide first. Therefore we need more than one transaction.

Implementing Choice



University of Kent | Computing

Implementing Choice

Meat Counter

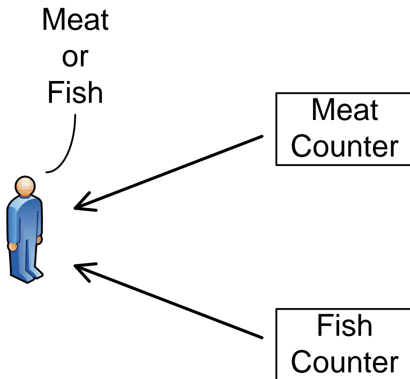
Fish Counter

University of Kent | Computing

Implementing Choice

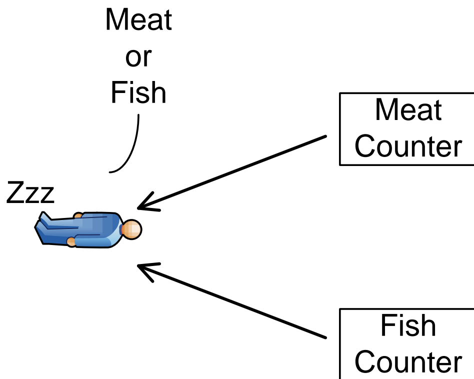
The reason for our revoke step being inside the transaction is to avoid this meat-fish problem. A customer places an order for meat or fish, but there is a race hazard of the two chefs both delivering to the customer. There are several ways to prevent this, but we make the person completing an event (the meat chef) revoke the offers of everyone involved (the customer) during the transaction.

Implementing Choice



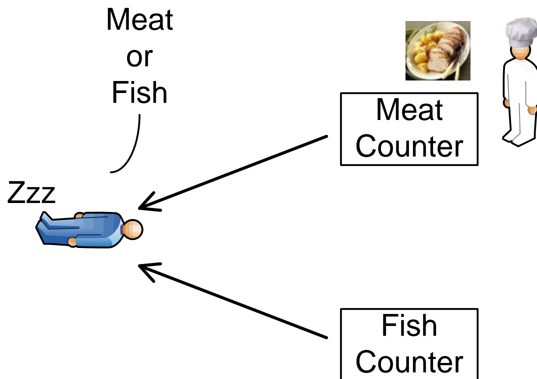
University of Kent | Computing

Implementing Choice



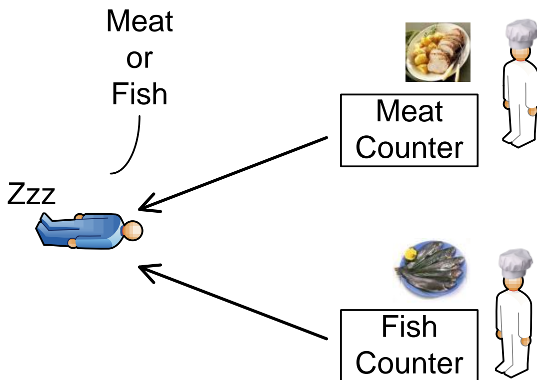
University of Kent | Computing

Implementing Choice



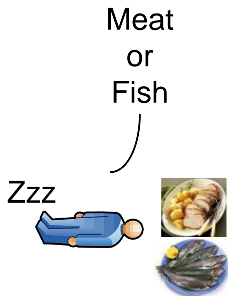
University of Kent | Computing

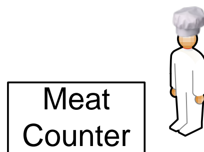
Implementing Choice

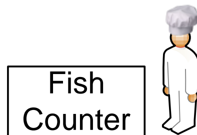


University of Kent | Computing

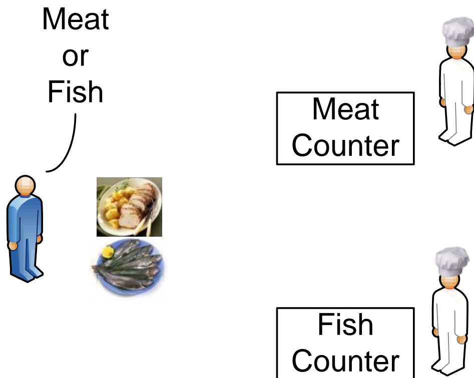
Implementing Choice





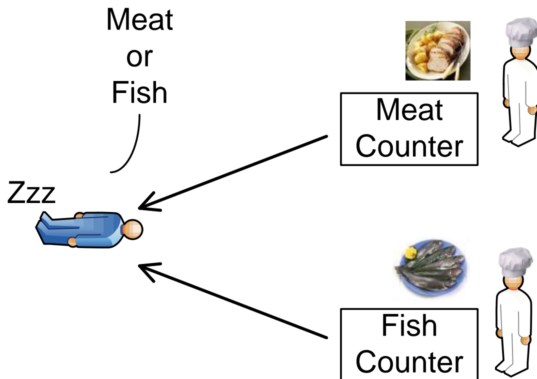


Implementing Choice



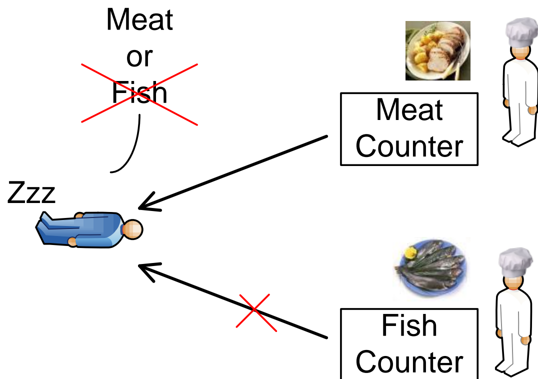
University of Kent | Computing

Implementing Choice



University of Kent | Computing

Implementing Choice



University of Kent | Computing

Implementation Features

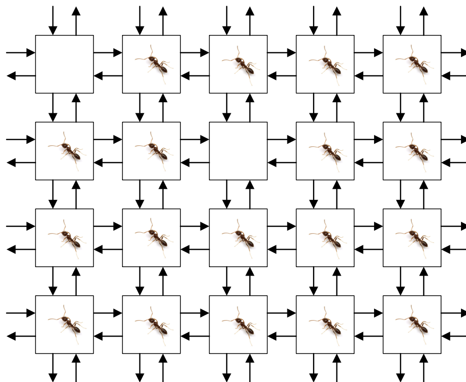
- The algorithm is sequential, bundled into a transaction
- No need to consider race hazards
- No *retry*, or *orElse*
- Various optimisations for search, not mentioned here

Scalability

University of Kent | Computing

There are several factors involved in how the algorithm scales. One is the complexity of the offers, the other is how well it scales to larger numbers of processors.

Scalability



University of Kent Computing

Cycling

Ant wants to go left; allow direct swap:

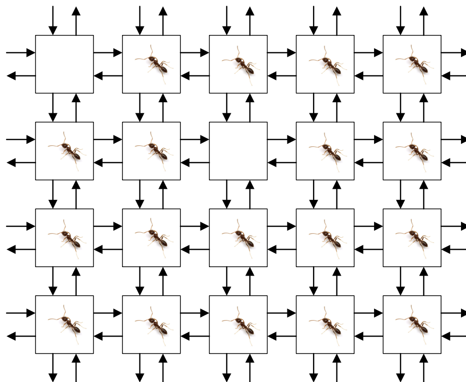
```
writeChannel leftOut ant <&> readChannel leftIn
```

Cycling

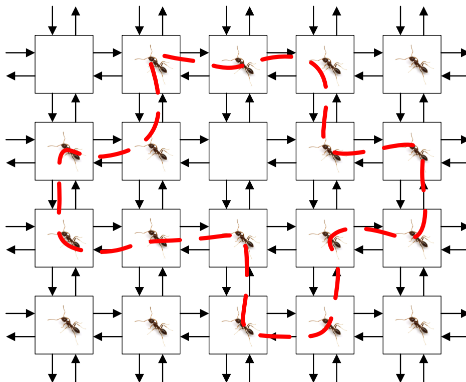
Ant wants to go left; allow cycling:

```
(writeChannel leftOut ant <&> readChannel leftIn)
<->
(writeChannel leftOut ant <&> readChannel rightIn)
<->
(writeChannel leftOut ant <&> readChannel upIn)
<->
(writeChannel leftOut ant <&> readChannel downIn)
```

Scalability



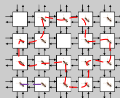
Scalability



University of Kent | Computing

Scalability

University of Kent | Computing



The algorithm will be slow for resolve this complicated cycle, yet (depending on the implementation of Transactional Memory) it should allow two disjoint event sets to be processed on different cores at the same time.

Summary

- Waiting for a disjunction or conjunction of events is useful
- We can use Transactional Memory to implement this
- Transactional Memory simplifies the algorithm
- Future work: priority, guards

Available in latest version of the Communicating Haskell Processes library; also an unpublished paper.

Distributivity

$a \wedge (b \vee c)$ is the same as $(a \wedge b) \vee (a \wedge c)$

$a \vee (b \wedge c)$ is NOT the same as $(a \vee b) \wedge (a \vee c)$

- RHS allows a and c to complete together, LHS does not

Back