Parallelism and Concurrency

Patrik Jansson
~~Koen Lindström Claessen~~
Chalmers University
Gothenburg, Sweden

Expressing Parallelism

- In a pure, lazy language
  - Evaluation is done when needed
  - Evaluation order does not affect meaning
  - Many sub-expr. could be eval. in parallel
  - But how can we express that?

Two primitives

```
pseq :: a -> b -> b
-- denotational semantics:
pseq _|_ y = _|_
pseq _   y = y
```

pseq x y:
Evaluate *first* x,
and then y

```
par :: a -> b -> b
-- denotational semantics:
par thread main = main
```

par thread main:
Evaluate thread *in parallel*, and
immediately return
main

Example

```
normal, paraNormal :: X -> Y -> N
paraNormal x y = x `par` y `par` normal x y
```

Idea: Write "normal" program first, then add parallelism to speed it up

Example: QuickSort

```
qsort :: (Ord a) => [a] -> [a]
qsort []        = []
qsort [x]       = [x]
qsort (x:xs) =
    losort `par` hisort `par` losort ++ (x:hisort)
  where
    losort = qsort [y | y <- xs, y < x ]
    hisort = qsort [y | y <- xs, y >= x ]
```

QuickSort (II)

```
qsort :: (Ord a) => [a] -> [a]
qsort []        = []
qsort [x]       = [x]
qsort (x:xs) =
    force losort `par` force hisort `par`
      losort ++ (x:hisort)
  where
    losort = qsort [y | y <- xs, y < x ]
    hisort = qsort [y | y <- xs, y >= x ]
```

```
force :: [a] -> ()
force []      = ()
force (x:xs) = x `pseq` force xs
```

Example: Parallel Map

```
pmap :: (a -> b) -> [a] -> [b]
pmap f []       = []
pmap f (x:xs) = fx  `par`  fxs  `par`  fx:fxs
   where
      fx  = f x
      fxs = pmap f xs
```

Evaluation Strategies

```
-- From module Control.Parallel.Strategies (v1)
type Done            =  ()
type Strategy a      =  a -> Done
```

```
using :: a -> Strategy a -> a
a `using` strat = strat a  `pseq`  a
```

Evaluation Strategies (II)

```
rwhnf :: Strategy a -- Called rseq in later versions
class NFData a where
  rnf :: Strategy a   -- Evaluate to normal form
```

```
parList :: Strategy a -> Strategy [a]
parList strat  []          = ()
parList strat  (x:xs)   = strat x  `par`  parList strat xs
```

Parallel Evaluation Strategies

```
pmap :: Strategy b -> (a -> b) -> [a] -> [b]
pmap strat f xs = map f xs `using` parList strat
```

More ...

- Implemented in GHC -- hackage `parallel`
  - Control.Parallel (par, pseq)
  - Control.Parallel.Strategies
- Also look at:
  - Control.Concurrent (`ghc -threaded`)
  - Control.Monad.STM
- RWH: Ch. 24 and Ch. 28

Concurrent Programming

- Processes
  - Concurrency
  - Parallelism
- Shared resources
  - Communication
  - Locks
  - Blocking

Concurrent Haskell

`Control.Concurrent`

```
fork    :: IO a -> IO Pid
kill    :: Pid -> IO ()
```

starting/killing processes

```
type MVar a
```

a shared resource

```
newEmptyMVar    :: IO (MVar a)
takeMVar        :: MVar a -> IO a
putMVar         :: MVar a -> a -> IO ()
```

blocking actions

Concurrent Haskell

`Control.Concurrent.Chan`

```
type Chan a

newChan    :: IO (Chan a)
readChan   :: Chan a -> IO a
writeChan  :: Chan a -> a -> IO ()
```

an unbounded channel

write returns immediately

Typical Concurrent Programming Today

- Use MVars (or similar concepts) to implement ”locks”
  - Grab the lock
    - Block if someone else has it
  - Do your thing
  - Release the lock

Problems With Locking

- Races
  - Forgotten lock
- Deadlock
  - Grabbing/releasing locks in wrong order
- Error recovery
  - Invariants
  - Locks

The Biggest Problem

- Locks are **not compositional**!
- Compositional = build a working system from working pieces

```
action1 = withdraw a 100
```

```
action2 = deposit b 100
```

```
action3 =
  do withdraw a 100
     deposit b 100
```

Inconsistent state

Solution (?)

- Expose the locks

```
action3 =
  do lock a
     lock b
     withdraw a 100
     deposit b 100
     release a
     release b
```

Danger of deadlock!

– better but error-prone
if a < b then do lock a; lock b
else do lock b; lock a

More Problems

```
action4 =
  do ...
```

```
action5 =
  do ...
```

```
action6 =
  action4 AND action5
```

Impossible!

Need to keep track of
all locks of an action,
and compose these

Conclusion

- Programming with explicit locks is
  - Not compositional
  - Not scalable (to many cores / threads)
  - Gives you a headache
  - Leads to code with errors
  - ...
- A new concurrent programming paradigm is sorely needed

Idea behind STM

- Borrow ideas from database people
  - Transactions
- Add ideas from functional programming
  - Computations are first-class values
  - What side-effects can happen where is controlled
- Et voila!

Software Transactional Memory (STM)

- First ideas in 1993
- New developments in 2005
  - Simon Peyton Jones
  - Simon Marlow
  - Tim Harris
  - Maurice Herlihy

Atomic Transactions

```
action3 =
  atomically $ do
    withdraw a 100
    deposit b 100
```

”write sequential code, and wrap atomically around it”

How Does It Work?

```
action3 =
  atomically $ do
    withdraw a 100
    deposit b 100
```

- Execute body without locks
- Each memory access is **logged**
- No actual update is performed
- At the end, we try to **commit** the log to memory
- Commit may fail, then we **retry** the whole atomic block

Transactional Memory

- No locks, so no race conditions
- No locks, so no deadlocks
- Error recovery is easy; an exception aborts the whole block and retries
- Simple code, and scalable

Caveats

- Absolutely forbidden:
  - To read a transaction variable outside an atomic block
  - To write to a transaction variable outside an atomic block
  - Side-effects inside an atomic block...

Simon's Missile Program

```
action3 =
  atomically $ do
     withdraw a 100
     launchNuclearMissiles
     deposit b 100

launchNuclearMissiles :: IO ()
```

No side effects allowed!
(type error)

STM Haskell

`Control.Concurrent.STM`

- First fully-fledged implementation of STM
- Impl.s for C++, Java, C# available
  - But it is difficult to solve the problems
- In Haskell, it is easy!
  - Controlled side-effects

STM Haskell

`Control.Concurrent.STM`

```
type STM a
instance Monad STM
```

```
type TVar a
newTVar      :: a -> STM (TVar a)
readTVar     :: TVar a -> STM a
writeTVar    :: TVar a -> a -> STM ()
```

```
atomically   :: STM a -> IO a  -- run function
```

Example

```
type Account = TVar Int
```

```
deposit :: Account -> Int -> STM ()
deposit r i = do        v <- readTVar r
                        writeTVar r (v+i)
```

```
main = do ... atomically (deposit r 13) ...
```

Example

```
retry    :: STM a
```

```
withdraw :: Account -> Int -> STM ()
withdraw r i = do     v <- readTVar r
                      if v < i then retry
                               else writeTVar r (v-i)
```

```
main = do   ...   atomically (do     withdraw r1 4
                                     deposit   r2 4  )   ...
```

Retrying

- An atomic block is retried when
  - the programmer says so, or
  - the commit at the end fails.
- *Before retrying, the STM implementation waits until one of the variables used in the atomic block is changed*
  - Why?

Referential transparency!

Compositional Choice

```
orElse :: STM a -> STM a -> STM a
```

```
main = do ... atomically (    withdraw r1 4
                            `orElse`
                              withdraw r2 4) ...
```

```
instance MonadPlus STM where
  mzero = retry
  mplus = orElse
-- Laws
m1 `orElse` (m2 `orElse` m3) = (m1 `orElse` m2) `orElse` m3
                    retry `orElse` m = m
                    m `orElse` retry = m
```

Blocking or not?

```
nonBlockWithdraw :: Account -> Int -> STM Bool
nonBlockWithdraw r i =
    do withdraw r i
       return True
  `orElse`
    return False
```

Choice of blocking / non-blocking is up to the *caller*, not the method (here ”withdraw”) itself

Example: MVars

- MVars can be implemented using TVars
- type MVar a = TVar (Maybe a)

(Demo if time permits.)

STM in Haskell summary

- Safe transactions through type safety
  - Degenerate ”IO-like” monad STM
    - We can only access TVars
    - TVars can only be accessed in STM monad
  - Referential transparency
- Explicit retry -- expressiveness
- Compositional choice -- expressiveness

Problems in C++ / Java / C#

- Retry semantics
- IO in atomic blocks
- Access of transaction variables outside of atomic blocks
- Access to regular variables inside of atomic blocks

STM Haskell

`Control.Concurrent.STM`

```
type STM a
instance Monad STM
```

```
type TVar a
newTVar    :: a -> STM (TVar a)
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM ()
```

```
atomically :: STM a -> IO a
retry      :: STM a
orElse     :: STM a -> STM a -> STM a
```