Parallel and Concurrent Programming in Haskell

An overview

Eva Burrows

BLDL-Talks
Department of Informatics, University of Bergen, Norway

March 02, 2010

WHAT IS HASKELL LIKE?

- purely functional - no side effects
- higher-order
- strongly typed, to the very extent
- general purpose
- good support for domain spesific languages
- LAZY!

WHY SO SUCCESSFUL?

- on the language front:
  - a committee of 15 people started to work seriously and enthusiastically on it (1990 - fisrt specification)
  - functional by default, imperativ on demand without ruining its “purity” (monads)
  - “the world’s finest imperative programming language”
  - general purpose
  - versatile: DSL, hardware design, etc
  - expressive power and elegancy
- on the implementation front:
  - fifths, latest standard: Haskell 98, with subsequent implementations, compilers. Still in use (Haskell-prime - the new revised language to come)
  - open source
  - well documented
  - well supported
  - raw computer power growth: expressiveness cost less
  - wiki website
  - scalable support fot concurrency and parallelism (past few years)
- impact:
  - its influence on other languages (ex. Python, C#, etc) made it more visible
  - evergrowing user group: academia first (lazyness, teaching language), later industry (Bluespec, ABN AMRO, and others)

EXPRESSIVE POWER

HASKELL CODE:

```
quicksort    :: [a] -> [a]
quicksort [] = []
 quicksort (x:xs) = quicksort small ++ (x : quicksort large)
   where small = [y | y <- xs, y <= x]
         large = [y | y <- xs, y > x]
```

C CODE:

```
void qsort(int a[], int lo, int hi) {
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);
    a[hi] = a[l];
    a[l] = p;
    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

ARE FUNCTIONAL PROGRAMMING LANGUAGES PARALLEL?

- there is a problem here...
- they state *what* to compute, in a pure way, but do not tell *how* to compute:
  - partial ordering on program execution
  - mapping a program onto a processor
  - distributing data

  unless there exists a *coordination language* to specify the *how*

STEPS TOWARDS A CONCURRENT AND A DATA PARALLEL HASKELL

- 1991: Paul Hudak proposes an annotation based meta-language to:
  - control evaluation order of expressions
  - map expressions to certain processors
  - parallelize lists
- 1996: *Concurrent Haskell*: language extension (4 primitives) to create explicit lightweight Haskell threads and manage communication (ex.: forkIO, MVars)
- 1998: *Algorithm+Strategies = Parallelism* introduces sparks and evaluation strategies (semi explicit parallelism)
- 2005: a new Haskell concurrency model based on *Transactional Memories*: they introduce constructs like *blocking*, *sequential composition* and *choice* over the traditional TM model
- 2006: *Data Parallel Haskell*: implementing nested data parallelism
- since 2001: *distributed Haskell*: conservative extension of both Concurrent Haskell and GpH
- all these well-supported and well-documented

Sparks and Evaluation Strategies

- *strategy*: annotation on code to hint where parallelism is useful, or to force that an expression should be evaluated first wrt to another
- *sparks*: result of parallel strategies, that can be turned into threads by the runtime system
- no restriction on what you can annotate
- deterministic without explicit threads, locks, or communication
- *ThreadsScope* profiler: allows to debug the parallel performance of Haskell programs
- two evalution strategies:
  - `par::  a -> b -> b`
  - `pseq::  a -> b -> b`
- ex.: `f 'par' e 'pseq' f+e`
- implemented as functions in the `Control.Parallel` module

Thread Programming in Shared Memory

- comes as a collection of library functions, not as an extra syntax
- idea: user creates explicit threads (forkIO) that can communicate and synchronize via “mutable variables”, boxes – like shared memory, or channels like message passing
- threads are then run concurrently by the Haskell runtime system
- non deterministic scheduling of the threads
- they are very cheap: couple of hundred bytes in memory, it is reasonable for a program to spawn thousands of them

SIMPLE EXAMPLE

Definition of the thread issuing function:

```
forkIO :: IO a -> IO ThreadId
```

Issuing a concrete thread:

```
main :: IO ()
main = do { forkIO (hPutStr stdout "Hello")
          ; hPutStr stdout " world\n" }
```

Communicating via MVar - a box:

```
communicate =
  do {m <- newEmptyMVar
     ; forkIO (
       do {
           v <- takeMVar m
          ; putStrLn ("received " ++ show v)})
     ;  putStrLn "sending"
     ;  putMVar m "wake up!"}
```

COMMUNICATING VIA STM IS SAFER

- MVar may lead to deadlocks: when a thread is waiting for a value from an other thread that will never appear
- or may lead to race conditions due to forgotten locks
- MVar – not composable: cannot glue correct subproblems into a big one
- lock-free synchronization via STM
- higher level than MVar, safer, composable, though slower

Basic Idea of Software Transactional Memory

- a concurrency mechanism borrowed from database transactions for controlling shared memory access by concurrent threads
- a transaction is a block of code with memory reads and writes executed by a single thread with the properties: *atomic*, *isolated*:
  - atomic: either all operations goes forward, or - if one fails - then all fails
  - isolated: intermediate states inside a transaction are not visible to other threads; the overall effect of the whole transaction becomes visible to all other threads at once; the execution of a block is unaffected by other threads
- has an *optimistic execution model*:
  - assuming no conflicts, transactions inside atomic blocks execute in a local thread transaction log, not actual memory
  - if the execution fails at some point, it rolls back everything, and eventually rerun the whole transaction,
  - in the end, the runtime system validates the log against real memory. In case of conflicts, rolls back everything, and rerun the transaction. If validation succeeds, the effect of the transaction is committed to real memory.
  - requires special control of side effects: things that cannot be undone, cannot be part of an atomic block

HASKELL'S CONTRIBUTION TO THIS IDEA

- *composable transactions*: by introducing some primitives
- wrap two operations into one - a new atomic transaction
- *blocking* transactions with *retry*: if a condition fails, abandon the current transaction and start it again when the transaction variable is updated (as an effect of an other thread)
- *orelse*: composition of alternatives. Tries two alternative paths, if both fail, rerun the whole transaction, when one of the transaction variables is updated (again as an effect of an other thread)
- the runtime system is responsable for checking the state of the transaction variables using the thread logs. When a condition variables have been updated, reruns the transaction.

SIMPLE EXAMPLE FOR A BANK TRANSACTION

The type of STM:

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

Defining a money transfer from one account into an other:

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer acc1 acc2 amount
         = atomically (do { deposit acc1 amount
                          ; withdraw acc2 amount })


withdraw :: Account -> Int -> STM ()
withdraw acc amount
   = do { bal <- readTVar acc
        ; if amount > bal then retry
          else writeTVar acc (bal - amount) }
```

CONCURRENCY VS. PARALLELISM (IN HASKELL)

- concurrency is a software structuring technique: to model computations as hypothetical independent activities (each having its own PC), that communicate and synchronise
- a parallel program looks for performance: by exploiting the potential of a real parallel computing resource:
  - task parallelism: concurrent execution of threads created for independent tasks
  - data parallelism: do the same thing in parallel over a large amount of data in parallel (no explicit threads, clear cost model, good locality)

INTRODUCING EXTRA SYNTAX FOR DATA PARALLELISM

- comes as part of `Data.Array.Parallel` module
- fundamental data structure is parallel array: `[:a:]`
- with associated parallel combinators:
  ex.: `mapP :: (a -> b) -> [:a:] -> [:b:]`, etc.
- with synstactic sugar: parallel array comprehension
- it is like a large array programming, with a big for loop
- it is too “flat”, so they say
- nested data parallelism is more intresting. . .

Nested Data Parallelism

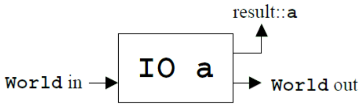
- 1990: Guy Blelloch described *nested* data-parallel programming:
- gives more flexibility to the programmer by opening a wider range of applications: divide and conquer, sparse matrix computations, graph algorithms
- harder to implement it: following Blelloch's idea, it is based on a systematic flattening transformation of any nested data parallel program into a flat one.

A Taste of Monads

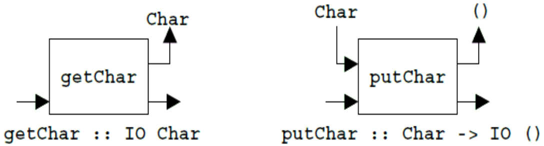
- motivated by simple I/O actions: since if no side effects, no I/O - this was embarassing
- monads are structures to supplement pure computations with features like state, common environment or I/O
- they simulate side effects

Monadic I/O

- `type IO a = World -> (a, World).`
  A value of type `IO a` is an I/O action.

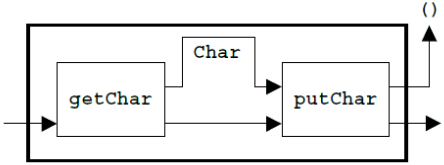


- ex.1: `getChar :: IO Char`
- ex.2: `putChar :: Char -> IO ()`



Monadic I/O cont.

- combining actions by sequential composition:
  `(» =) ::  IO a -> (a -> IO b) -> IO b`



getChar >>= putChar

- complete Haskell program defines a single big I/O action, called `main`, of type `IO ()`

```
main :: IO ()
main = getChar >>= \c -> putChar c
```

- there are defined several “gluing” combinators, and other mechanism to “fish out” the result of an action in a variable

MONADIC COMPUTATIONS SIMPLE: DO

```
getTwoChars :: IO (Char,Char)
getTwoChars = do { c1 <- getChar ;
                   c2 <- getChar ;
                   return (c1,c2)
                 }
```

- where `return` is also a combinator: `return ::  a -> IO a`.
  It has no side effects.

OTHER USES

- imperative control structures can also be implemented as actions:

```
repeatN :: Int -> IO a -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

- references: to model mutable variables

```
data IORef a -- An abstract type
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`

So What Is a Monad?

- An *abstract data type* with:
  - a type constructor $\mathscr{M}$
  - two functions:

$$\texttt{return} :: \forall\alpha.\alpha \to \mathscr{M}\,\alpha$$

$$\texttt{>>==}:: \forall\alpha\beta.\mathscr{M}\,\alpha \to (\alpha \to \mathscr{M}\,\beta) \to \mathscr{M}\,\beta$$

  - satisfying certain algebraic laws

$$\boxed{\begin{array}{c} \texttt{return}\,x \texttt{ >>= } f \quad = \quad f\,x \qquad (LUNIT) \\[2ex] m \texttt{ >>= return} \quad = \quad m \qquad (RUNIT) \\[2ex] \dfrac{x \text{ does not apprear free in } m_3}{m_1 \texttt{ >>= } (\lambda x.m_2 \texttt{ >>= } (\lambda y.m_3)) \quad = \quad (m_1 \texttt{ >>= } (\lambda x.m_2)) \texttt{ >>= } (\lambda y.m_3)}\ (BIND) \end{array}}$$

MAIN REFERENCES

- *Beautiful Concurrency*, Peyton Jones, O'Reilly 2007
- *Harnessing the Multicores: Nested Data Parallelism in Haskell*, Peyton Jones, Leshchinkskiy, Keller, Chakravarty, 2008.
- *A Tutorial on Parallel and Concurrent Programming in Haskell*, Peyton Jones and Singh. 2008
- *Real World Haskell*, O'Sullivan, Goerzon, Stewart. O'Reilly 2008
- *A History of Haskell: being lazy with class*, Hudak, Hughes, Peyton Jones, Wadler, The Third ACM SIGPLAN History of Programming Languages Conference, 2007.
- *Composable Memory Transactions*, Harris, Marlow, Peyton Jones, Herlihy. PPoPP'05
- *A Gentle Introduction to Haskell 98*, Hudak, Peterson, Fasel, 1999
- *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, Peyton Jones, 2009