Haskell: Compiler as Theorem-Prover

Greg Price (`price`)

2007 Nov 19

code samples: `http://cluedumps.mit.edu/wiki/2007/11-19`

Software Transactional Memory

- Concurrency: locking

Software Transactional Memory

- Concurrency: locking costly, deadlocks, bugs.
- Optimistic transactions, restarting

Software Transactional Memory

- Concurrency: locking costly, deadlocks, bugs.
- Optimistic transactions, restarting
- Worse bugs:

```
void f() {
  begin_transaction();
  if (x != y)
    launch_missiles();
  end_transaction();
}
```

```
void g() {
  begin_transaction();
  x++;
  y++;
  end_transaction();
}
```

Software Transactional Memory

- Concurrency: locking costly, deadlocks, bugs.
- Optimistic transactions, restarting
- Worse bugs:

```
void f() {
  begin_transaction();
  if (x != y)
    launch_missiles();
  end_transaction();
}
```

```
void g() {
  begin_transaction();
  x++;
  y++;
  end_transaction();
}
```

Restart side effects?

Software Transactional Memory

- Concurrency: locking costly, deadlocks, bugs.
- Optimistic transactions, restarting
- Worse bugs:

```
void f() {
  begin_transaction();
  if (x != y)
    launch_missiles();
  end_transaction();
}
```

```
void g() {
  begin_transaction();
  x++;
  y++;
  end_transaction();
}
```

Restart side effects?

- & all the old bugs too

Software Transactional Memory

- Solution:

```
f = atomically $
    do xv <- readTVar x
       yv <- readTVar y
       if xv /= yv then launch_missiles_soon
                   else return ()
g = atomically $
    do xv <- readTVar x; writeTVar x (xv+1)
       yv <- readTVar y; writeTVar y (yv+1)
```

(see example STMExample)

Software Transactional Memory

- Solution:

```
f = atomically $
      do xv <- readTVar x
         yv <- readTVar y
         if xv /= yv then launch_missiles_soon
                     else return ()
g = atomically $
      do xv <- readTVar x; writeTVar x (xv+1)
         yv <- readTVar y; writeTVar y (yv+1)
```

  (see example STMExample)
- can't have (non-transactional) side effects

Software Transactional Memory

- Solution:

```
f = atomically $
    do xv <- readTVar x
       yv <- readTVar y
       if xv /= yv then launch_missiles_soon
                   else return ()
g = atomically $
    do xv <- readTVar x; writeTVar x (xv+1)
       yv <- readTVar y; writeTVar y (yv+1)
```

  (see example STMExample)
- can't have (non-transactional) side effects
- no special compiler support (except runtime)

Software Transactional Memory

- Solution:

```
f = atomically $
    do xv <- readTVar x
       yv <- readTVar y
       if xv /= yv then launch_missiles_soon
                   else return ()
g = atomically $
    do xv <- readTVar x; writeTVar x (xv+1)
       yv <- readTVar y; writeTVar y (yv+1)
```

  (see example `STMExample`)
- can't have (non-transactional) side effects
- no special compiler support (except runtime)
- other bugs ruled out too

STM: Guaranteeing No Side Effects

- pure

STM: Guaranteeing No Side Effects

- pure
- `putStr "hello" :: IO ()`

STM: Guaranteeing No Side Effects

- pure
- `putStr "hello" :: IO ()` an IO action

STM: Guaranteeing No Side Effects

- pure
- `putStr "hello" :: IO ()`   an IO action
- sequenced: `do { ...; f :: IO a; ... }`

STM: Guaranteeing No Side Effects

- pure
- `putStr "hello" :: IO ()` an IO action
- sequenced: `do { ...; f :: IO a; ... }`
- executed only through `main`:

```
main :: IO ()
main = do putStr "Hello world!\n"
          launch_missiles
```

STM: Guaranteeing No Side Effects

- pure
- `putStr "hello" :: IO ()` an IO action
- sequenced: `do { ...; f :: IO a; ... }`
- executed only through `main`:

```
main :: IO ()
main = do putStr "Hello world!\n"
          launch_missiles
```

⇒ side effects only through type `IO a`

STM: Guaranteeing No Side Effects

- side effects only through type `IO a`

STM: Guaranteeing No Side Effects

- side effects only through type `IO a`
- `atomically :: STM a -> IO a`

STM: Guaranteeing No Side Effects

- side effects only through type `IO a`
- `atomically :: STM a -> IO a`
- ```
  newTVar   :: a -> STM (TVar a)
  readTVar  :: TVar a -> STM a
  writeTVar :: TVar a -> a -> STM ()
  ```
  ```

STM: Guaranteeing No Side Effects

- side effects only through type `IO a`
- `atomically :: STM a -> IO a`
- ```
  newTVar   :: a -> STM (TVar a)
  readTVar  :: TVar a -> STM a
  writeTVar :: TVar a -> a -> STM ()
  ```
- `do { ...; f :: STM a; ... }`   (same)
  ```

Protocol Types

- `spec :: Spec ((Snd Int :+: Snd String) :->: End) IOChan`
  a protocol spec

Protocol Types

- $\texttt{spec :: Spec } \underbrace{\texttt{((Snd Int :+: Snd String) :->: End)}}_{s} \texttt{ IOChan}$

  a protocol spec

Protocol Types

- $\texttt{spec :: Spec } \underbrace{\texttt{((Snd Int :+: Snd String) :->: End)}}_{s} \texttt{ IOChan}$

  a protocol spec
- `accept spec`
- `request spec`

Protocol Types

- $\texttt{spec :: Spec } \underbrace{\texttt{((Snd Int :+: Snd String) :->: End)}}_{s} \texttt{ IOChan}$

  a protocol spec

- ```
  accept spec  :: (Extend M (ChanCap c s) e e' n) =>
                  LinearT IO e e' (LVar n)
  ```

- ```
  request spec :: (Dual s s',
                   Extend M (ChanCap c s') e e' n) =>
                  LinearT IO e e' (LVar n)
  ```

Protocol Types

- $\texttt{spec :: Spec } \underbrace{\texttt{((Snd Int :+: Snd String) :->: End)}}_{s} \texttt{ IOChan}$

  a protocol spec

- ```
  accept spec  :: (Extend M (ChanCap c s) e e' n) =>
                  LinearT IO e e' (LVar n)
  ```

- ```
  request spec :: (Dual s s',
                   Extend M (ChanCap c s') e e' n) =>
                  LinearT IO e e' (LVar n)
  ```

- `runLinearT (accept spec >>>= ...) :: IO a`
  executes protocol *exactly*

Protocol Types: Means of Proof

- `runLinearT :: LinearT IO Empty Empty a -> IO a`

Protocol Types: Means of Proof

- `runLinearT :: LinearT IO Empty Empty a -> IO a`
- environments of capabilities

Protocol Types: Means of Proof

- `runLinearT :: LinearT IO Empty Empty a -> IO a`
- environments of capabilities
-
```
send :: (Evolve n c (Snd a :->: x) e x e') =>
        LVar n -> a -> LinearT IO e e' ()
recv :: (Evolve n c (Rcv a :->: x) e x e') =>
        LVar n -> LinearT IO e e' a
```

Protocol Types: Means of Proof

- `runLinearT :: LinearT IO Empty Empty a -> IO a`
- environments of capabilities
- ```
  send :: (Evolve n c (Snd a :->: x) e x e') =>
          LVar n -> a -> LinearT IO e e' ()
  recv :: (Evolve n c (Rcv a :->: x) e x e') =>
          LVar n -> LinearT IO e e' a
  ```
- ```
  sel1 :: (Evolve n c ((x1:+:x2):->:y) e (x1:->:y) e') =>
          LVar n -> LinearT IO e e' ()
  ```

Protocol Types: Generic Building Blocks

```
data T                  class Prop a
data F                  instance Prop T
                        instance Prop F
```

Protocol Types: Generic Building Blocks

```
data T
data F
```

```
class Prop a
instance Prop T
instance Prop F
```

```
class Prop b => Equal x y b | x y -> b
```

Protocol Types: Generic Building Blocks

```haskell
data T                  class Prop a
data F                  instance Prop T
                        instance Prop F

class Prop b => Equal x y b | x y -> b

data Z                  class Nat a
data S x                instance Nat Z
                        instance Nat n => Nat (S n)
```

Protocol Types: Generic Building Blocks

```
data T                 class Prop a
data F                 instance Prop T
                       instance Prop F
```

```
class Prop b => Equal x y b | x y -> b
```

```
data Z                 class Nat a
data S x               instance Nat Z
                       instance Nat n => Nat (S n)
```

```
instance Equal Z Z T
instance Nat n => Equal (S n) Z F
instance Nat n => Equal Z (S n) F
instance (Nat n1, Nat n2, Equal n1 n2 b)
         => Equal (S n1) (S n2) b
```

Protocol Types: Generic Building Blocks

```
data T                class Prop a
data F                instance Prop T
                      instance Prop F
```

```
class Prop b => Equal x y b | x y -> b
```

```
data Z                class Nat a
data S x              instance Nat Z
                      instance Nat n => Nat (S n)
```

```
instance Equal Z Z T
instance Nat n => Equal (S n) Z F
instance Nat n => Equal Z (S n) F
instance (Nat n1, Nat n2, Equal n1 n2 b)
         => Equal (S n1) (S n2) b
```

also lists, environments, many other things

Other popular theorems

- type-checked physical dimensions:

```
newton = kg <*> m </> s </> s
thrust = dm 12537.2 <*> newton
dm 1 <*> m   <+>  dm 1 <*> m</>s  -- error!
```

  (see example `Dimensional`)

Other popular theorems

- type-checked physical dimensions:
```
newton = kg <*> m </> s </> s
thrust = dm 12537.2 <*> newton
dm 1 <*> m   <+>  dm 1 <*> m</>s   -- error!
```
  (see example `Dimensional`)
- mutable state on a leash:
```
runST :: (forall s. ST s a) -> a
```

Other popular theorems

- type-checked physical dimensions:
```
newton = kg <*> m </> s </> s
thrust = dm 12537.2 <*> newton
dm 1 <*> m   <+>  dm 1 <*> m</>s  -- error!
```
  (see example `Dimensional`)
- mutable state on a leash:
```
runST :: (forall s. ST s a) -> a
```
- "theorems for free":
```
if    maybemap :: (a -> b) -> [a] -> [b]
then  maybemap f == maybemap id . map f
```

A Proof

$$A \wedge B \Rightarrow B \wedge A$$

A Proof

$$\{\} \vdash A \wedge B \Rightarrow B \wedge A$$

A Proof

$$\frac{\{A \wedge B\} \vdash B \wedge A}{\{\} \vdash A \wedge B \Rightarrow B \wedge A}$$

A Proof

$$\dfrac{\dfrac{\dfrac{}{\{A \wedge B\} \vdash A \wedge B}}{\{A \wedge B\} \vdash B} \qquad \dfrac{\dfrac{}{\{A \wedge B\} \vdash A \wedge B}}{\{A \wedge B\} \vdash A}}{\dfrac{\{A \wedge B\} \vdash B \wedge A}{\{\} \vdash A \wedge B \Rightarrow B \wedge A}}$$

A Proof

$$\dfrac{\dfrac{\dfrac{}{\{A \wedge B\} \vdash A \wedge B}\,Id}{\{A \wedge B\} \vdash B}\,\wedge_{E2} \qquad \dfrac{\dfrac{}{\{A \wedge B\} \vdash A \wedge B}\,Id}{\{A \wedge B\} \vdash A}\,\wedge_{E1}}{\dfrac{\{A \wedge B\} \vdash B \wedge A}{\{\} \vdash A \wedge B \Rightarrow B \wedge A}\,\Rightarrow_I}\,\wedge_I$$

A Proof (the same one)

$$\lambda x.\ \texttt{pair}\,(\texttt{snd}\,x)\,(\texttt{fst}\,x)$$

A Proof (the same one)

$$
\dfrac{
  \dfrac{\dfrac{}{\{x : A \times B\} \vdash x : A \times B}\,Id}{\{x : A \times B\} \vdash (snd\ x) : B}\,\times_{E2}
  \qquad
  \dfrac{\dfrac{}{\{x : A \times B\} \vdash x : A \times B}\,Id}{\{x : A \times B\} \vdash (fst\ x) : A}\,\times_{E1}
}{
  \dfrac{\{x : A \times B\} \vdash (pair\ (snd\ x)(fst\ x)) : B \times A}{\{\} \vdash (\lambda x.\ \ pair\ (snd\ x)(fst\ x)) : A \times B \rightarrow B \times A}\,\rightarrow_I
}\,\times_I
$$

References

- all at `http://cluedumps.mit.edu/wiki/2007/11-19`
- STM: Harris, Marlow, Peyton Jones, Herlihy 2005; Peyton Jones "Beautiful Concurrency" for intro
- protocol types: Jesse Tov, unpublished. Some of the ideas in Oleg Kiselyov's HList.
- "theorems for free": Phil Wadler, 1989. Now $\exists$ a web app.