Haskell: Compiler as Theorem-Prover

Greg Price (price)

2007 Nov 19

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

Greg Price (price) ()

Haskell: Compiler as Theorem-Prover

2007 Nov 19 1 / 26



2 Protocol Types





Greg Price (price) ()



• Concurrency: locking

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

• 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();
}
```

• Concurrency: locking costly, deadlocks, bugs.

- Optimistic transactions, restarting
- Worse bugs:

```
void f() { void g() {
    begin_transaction(); begin_transaction();
    if (x != y) x++;
    launch_missiles(); y++;
    end_transaction(); end_transaction();
}
Restart side effects?
```

• Concurrency: locking costly, deadlocks, bugs.

- Optimistic transactions, restarting
- Worse bugs:

```
void f() { void g() {
    begin_transaction(); begin_transaction();
    if (x != y) x++;
    launch_missiles(); y++;
    end_transaction();
    }
    Restart side effects?
```

& all the old bugs too

• can't have (non-transactional) side effects

```
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)</pre>
```

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

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

STM: Guaranteeing No Side Effects

o pure

< 🗇 🕨 < 🖃 🕨

STM: Guaranteeing No Side Effects

o pure

• putStr "hello" :: IO ()

.

STM: Guaranteeing No Side Effects

o pure

• putStr "hello" :: IO () an IO action

- **4 ∃ ≻** 4

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

- 4 ⊒ →

- o 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

```
o 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
- \Rightarrow side effects only through type IO a

• side effects only through type IO a

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

- 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 ()

- 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)



2 Protocol Types





Greg Price (price) ()



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

► < ∃ ►</p>

• spec :: Spec ((Snd Int :+: Snd String) :->: End) s protocol spec

► < ∃ ►</p>

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

- accept spec
- request spec

- ∢ ∃ ▶

• spec :: Spec	((Snd Int :+: Snd String) :->: End) IOChan
a protocol spec	5
• accept spec	<pre>:: (Extend M (ChanCap c s) e e' n) => LinearT IO e e' (LVar n)</pre>
• request spec	<pre>:: (Dual s s', Extend M (ChanCap c s') e e' n) => LinearT IO e e' (LVar n)</pre>

メロト メポト メヨト メヨト

• spec :: Spec	((Snd Int :+: Snd String) :->: End) IOChan
a protocol spec	s
• accept spec	<pre>:: (Extend M (ChanCap c s) e e' n) => LinearT IO e e' (LVar n)</pre>
• request spec	<pre>:: (Dual s s', Extend M (ChanCap c s') e e' n) => LinearT IO e e' (LVar n)</pre>
• runLinearT (a	ccept spec >>>=) :: IO a

executes protocol exactly

Image: A match a ma

4

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

- ∢ ∃ ▶

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

- 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

- 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' ()

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

< 一型

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

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

► < ∃ ►</p>

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)

data T class Prop a
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
```

ъ

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



2 Protocol Types





Greg Price (price) ()



A Proof

$A \land B \Rightarrow B \land A$

∃ 990

・ロト ・ 日 ト ・ ヨ ト ・ ヨ ト

A Proof

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

Greg Price (price) ()

Haskell: Compiler as Theorem-Prover

 ▶
 ▲
 ■
 ∽
 Q
 Q

 2007 Nov 19
 21 / 20

・ロト ・ 日 ト ・ ヨ ト ・ ヨ ト

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

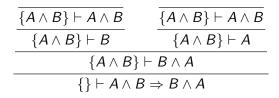
Greg Price (price) ()

Haskell: Compiler as Theorem-Prover

2007 Nov 19 21 / 26

イロト イヨト イヨト イヨト

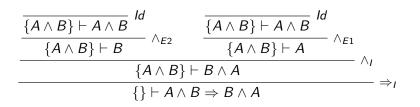
-2



Greg Price (price) ()

3 2007 Nov 19 21 / 26

-



▶ < ≣ > ঊ ∕ ৭ে 2007 Nov 19 21 / 26

► < ∃ ►</p>

A Proof (the same one)

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

A Proof (the same one)

$$\frac{\overline{\{x:A \times B\} \vdash x:A \times B} \ ^{ld}}{\{x:A \times B\} \vdash (\mathsf{snd} \ x):B} \times_{E2}} \qquad \frac{\overline{\{x:A \times B\} \vdash x:A \times B} \ ^{ld}}{\{x:A \times B\} \vdash (\mathsf{fst} \ x):A} \times_{E1}}{\{x:A \times B\} \vdash (\mathsf{pair} \ (\mathsf{snd} \ x)(\mathsf{fst} \ x)):B \times A} \times_{I}}{\{\} \vdash (\lambda x. \ \mathsf{pair} \ (\mathsf{snd} \ x)(\mathsf{fst} \ x)):A \times B \to B \times A} \rightarrow$$

Greg Price (price) ()

Image: A match a ma

- 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.