Ants

Amazing ants!

Martijn Fleuren Marinus Oosters Carlos Tomé Cortiñas
Matthew Swart

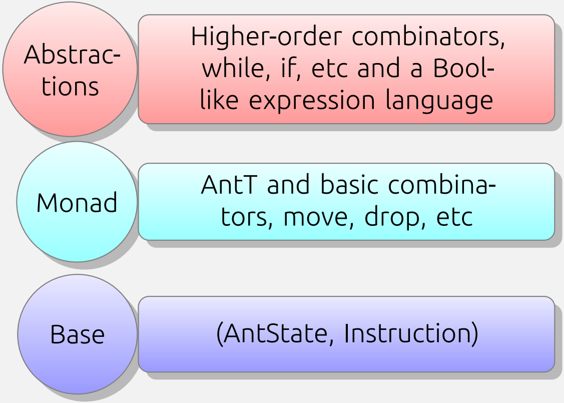
Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Overview

- Architecture
- Optimizations
- Testing

Architecture of the EDSL



Abstractions

Higher-order combinators, while, if, etc and a Bool-like expression language

Monad

AntT and basic combinators, move, drop, etc

Base

(AntState, Instruction)

Internals (I) (Ant/Monad.hs)

```
import              Control.Monad.Tardis

newtype AntT m l a =
  AntT { runAnt :: TardisT (Program l) l m a }
```

> *A Tardis is the combination of the State monad transformer and the Reverse State monad transformer.*

```
instance MonadFix m => Monad (TardisT bw fw m)
  ...
```

Internals (II) (Ant/Monad.hs)

```
data Program l =
  Program { _entry      :: l
          , _commands   :: Map l (Command l) }

goto :: MonadFix m  => l -> AntT m l ()
goto = ...

label :: MonadFix m => AntT m l l
label = ...
```

Example

```haskell
{-# LANGUAGE RecursiveDo #-}

prog1 :: (MonadFix m, Label l)
      => AntT m l ()
prog1 = mdo
  l <- label
  turn left
  goto l
```

Abstractions over AntT (Abstractions.hs)

```
loop :: MonadFix m
     => (AntT m l () -> AntT m l () -> AntT m l a)
     -> AntT m l ()
loop cmds = mdo
    cont <- label
    cmds (goto cont) (goto brk)
    goto cont
    brk <- label
    return ()
```

Genetic (I) (Genetic/Evolve.hs)

Optimizations (I) (Ant/Optimizations.hs)

The size of generated programs is huge, consider:

```
move (goto p_label) (goto p_label)
p_label <- label <* p
```

versus

```
move p p
```

Optimizations (II) (Ant/Optimizations.hs)

We define a optimization as:

```
newtype Opt l1 l2 =
  Opt { unOpt :: Program l1 -> Program l2 }

type Optimization l = Opt l l

instance C.Category Opt where

applyOpt :: Optimization l -> Program l -> Program l
```

. . . And we have implemented a couple of them:

```
unreachableOpt   :: Label l => Optimization l
duplicateCodeOpt :: Label l => Optimization l
```

Testing (I) (test/Spec.hs)

How do we know that the optimizations do not change the `intended` behaviour of a program?

At least the `optimised` program should be valid ...

Testing (I) (test/Spec.hs)

How do we know that the optimizations do not change the `intended` behaviour of a program?

At least the `optimised` program should be valid ...

QuickCheck!

Testing (I) (test/Spec.hs)

How do we know that the optimizations do not change the `intended` behaviour of a program?

At least the `optimised` program should be valid …

QuickCheck!

```
testOptimization :: Op -> AntMTest L -> Bool
testOptimization opt antm =
  valid $ applyOpt (toOptimization opt)
                   (compileProg (toAntM antm))
```

Testing (I) (test/Spec.hs)

How do we know that the optimizations do not change the `intended` behaviour of a program?

At least the `optimised` program should be valid …

QuickCheck!

```
testOptimization :: Op -> AntMTest L -> Bool
testOptimization opt antm =
  valid $ applyOpt (toOptimization opt)
                   (compileProg (toAntM antm))
```

But we could do better, and we do!

Testing (I) (test/Spec.hs)

```
test :: Int -> Int -> AntMTest L -> Op -> Property
test r seed cprog opt = do
  gs1  <- run $
    initGameState seed tinyWorld
        (toCmds cprog)
        blackInstr
    >>= runNRounds r
  gs2  <- run $
    initGameState seed tinyWorld
        (toCmds $ applyOpt
                  (toOptimization opt)
                  cprog)
        blackInstr
    >>= runNRounds r
  run (gs1 =~= gs2)
```

Questions

Thank **you** for your **attention**!

Any Questions?