# Ants

## Amazing ants!

Martijn Fleuren Marinus Oosters Carlos Tomé Cortiñas
Matthew Swart

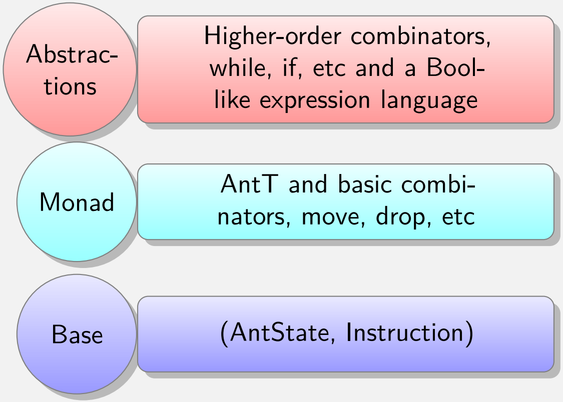
Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

## Overview

- Architecture
- Genetic Strategy search
- Optimizations
- Testing

# Architecture of the EDSL


Abstrac-tions
Higher-order combinators, while, if, etc and a Bool-like expression language
Monad
AntT and basic combi-nators, 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)

```
-- | Look until we can find what we are looking for.
-- leave a mark on the way.
search :: (MonadFix m, Label l)
       => Maybe Marker
       -> Condition
       -> AntT m l ()
search m cond = do
  while (Not (ahead :=: cond     :|:
              leftAhead :=: cond :|:
              rightAhead :=: cond)) $ do
    try 2 move turnRandom
    maybe (return ()) mark m
    flip_ 15 turnRandom
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

## Genetic (I) (Genetic/Evolve.hs)

- Instead of thinking deep about how to write a strategy, let a computer do the searching for you.
- How to generate random programs?

## Genetic (II) Meat of the search

- QuickCheck has `generate :: Gen a -> IO a` to transfer random samples to `IO`
- Use some kind of 'max' function and then use it in a fold

```
evalP (p1, f1) p2 =
  fitness p2 >>=
    \f2 -> return $ if f1 < f2 then (p2, f2)
      else (p1, f1)
search n = do
  prog1     <- newProgram
  fit1      <- fitness prog1
  xs        <- generate n - 1 programs
  (best,_)  <- foldM evalP (prog1, fit1) xs
  return best
```

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

## Genetic (III) Results

- Benchmark against the winner of ICFP2004 lightning division
- None of the programs obtained a score $>0$
- There is no meaningful way to tune a random program
- Possible solution: write small programs and compose them randomly (attempted, not finished)
- Brute force random search is *not a good idea*

## 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 ...

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

## 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!*

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

## 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)

```haskell
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)
```

## Conclusions

- Making the EDSL was a lot of fun
- But making good strategies even with a nice EDSL is hard
- The Tardis monad is a very powerful abstraction for writing assembly-like code

**Thank you for your attention!**

**Any Questions?**