Work In Progress: Building Yoyak Engine

Jaeho Shin

2010-11-12

ROPAS Show&Tell
Contents

1. Scalability with Multiprocess Model
2. Simplicity with Haskell Type Classes
3. Transparency with Haskell Interactive
Scalability with Multiprocess Model

(working with Wonchan)
Our Analyzers

- Engine
  - Worklist
  - Graph

- Semantics
  - Table

A

F

Mairac
Our Analyzers

Engine

Semantics

Worklist

Graph

Table

Process Boundaries
External Memory

Engine

Semantics

Filesystem

Graph

Worklist

Table
External Memory

Engine

Semantics

Filesystem

Graph

Worklist

Table
External Memory

Graph

Worklist

Table

Engine

Semantics

A

F
Splitting Processes

Engine

Semantics

Graph

Table

Worklist
Into Many Processes

Engine

Semantics

Semantics

Semantics

Graph

Worklist

Table
A Multiprocess Model

Engine

Master

Worker

Worker

Worker

Semantics

Graph

Worklist

Table
Control & Data Flow

- Master
- Worker
- Worklist
- Graph
- Table
Master & Worker Roles

Master

Worker

control
data
Master & Worker Roles

1. Choose work from Worklist
Master & Worker Roles

1M. Choose work from Worklist

2M. Load partial control-flow info
Master & Worker Roles

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work
Master & Worker Roles

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work
4W. Load input states
Master & Worker Roles

1M. Choose work from Worklist

2M. Load partial control-flow info

3M. Assign Work

4W. Load input states

5W. Compute node semantics

control data
Master & Worker Roles

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work
4W. Load input states
5W. Compute node semantics
6W. Widen & check order
Master & Worker Roles

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work

4W. Load input states
5W. Compute node semantics
6W. Widen & check order
7W. Store output states
Master & Worker Roles

**Master**

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work

**Worker**

4W. Load input states
5W. Compute node semantics
6W. Widen & check order
7W. Store output states
8W. Return Result

---

**Worklist**

---

control
data
Master & Worker Roles

Master

1M. Choose work from Worklist
2M. Load partial control-flow info
3M. Assign Work
9M. Bookkeep Graph & Worklist

Worker

4W. Load input states
5W. Compute node semantics
6W. Widen & check order
7W. Store output states
8W. Return Result

control
data
Going Multiprocess
For Scalability!
Simplicity with Haskell Type Classes
Haskell Type Classes

A clean way to do
ad-hoc polymorphism
Haskell Type Classes

A clean way to do operator overloading
Operations for Abstract Domain

- bottom
- join or lub
- order check
- meet or glb
- widening
- narrowing
Operations for Abstract Domain

- bottom
- join or lub
- order check
- meet or glb

AbsDom

- widening
- narrowing

Widen Narrow
Operations for Numerical Values

- +
- ×
- negate
- fromInteger
- /
- fromRational
Operations for Numerical Values

- +
- ×
- negate
- fromInteger
- / fromRational

Num Fractional
AbsDom Class

```haskell
-- We'll need operations on complete partial order domains with
-- join-completion for convenience.

class Eq a => AbsDom a where
  bottom :: a
  top :: a
  infixr 5 <=^ 
  (<=^) :: a -> a -> Maybe Bool  -- partial order
  infixr 6 ++^ 
  (++^) :: a -> a -> a  -- join
  infixr 7 **^ 
  (**^) :: a -> a -> a  -- meet
  lub :: [a] -> a
  lub = foldr (++^) bottom
  glb :: [a] -> a
  glb [] = bottom
  glb as = foldr1 (**^) as

instance (AbsDom a, AbsDom b) => AbsDom (a, b) where
  bottom = (bottom, bottom)
  top = (top, top)
  (a, b) <=^ (c, d) = do p <- a <=^ c
                         q <- b <=^ d
```

"Yoyak/Domain/Base.hs" 136 lines --8%--
Interval Data Type

We first need a data type with omegas on the positive and negative sides. We will make it an instance of Num and Fractional for ease of use.

```haskell
data Omega a = InfNeg | Num a | InfPos | NaN
    deriving (Show, Read, Eq, Ord)
```

Using Omega, we will now define the Interval data type. Interval will also be an instance of Num and Fractional for code simplicity.

```haskell
data Interval a = Interval { lo :: Omega a, hi :: Omega a }
                   | Empty
    deriving (Show, Read, Eq, Ord)
```

It may be helpful to have a function which finds the upper and lower bound of given list of numbers.

```haskell
interval0f [] = Empty
interval0f zs = Interval { lo = minimum zs, hi = maximum zs }
```

"Yoyak/Interval/Domain.hs" 157 lines --53%-- 84,1 11%
Interval as an AbsDom Instance

```haskell
instance Ord a => AbsDom (Interval a) where
  bottom = Empty
  top = Interval InfNeg InfPos
  Empty \<^\ y = Just True
  Empty \<^\ Empty = Just False
  x \<^\ y | lo y \<^\ lo x && hi x \<^\ hi y = Just True
           | lo x < lo y && hi y < hi x = Just False
           | otherwise = Nothing
  Empty \++^\ y = y
  x \++^\ Empty = x
  x \++^\ y = validate $ Interval (min (lo x) (lo y)) (max (hi x) (hi y))
  Empty \++^\ y = Empty
  x \++^\ Empty = Empty
  x \++^\ y = validate $ Interval (max (lo x) (lo y)) (min (hi x) (hi y))
  lub [] = bottom
  lub xs = validate $ Interval l h where
    l = (minimum . map lo . filter (bottom/=)) $ xs
    h = (maximum . map hi . filter (bottom/=)) $ xs
  glb [] = bottom
  glb xs | bottom `elem` xs = bottom
          | otherwise = validate $ Interval (maximum . map lo $ xs) (maximum
```
Interval as a Num Instance

instance (Ord a, Num a) ⇒ Num (Interval a) where
  Empty + _ = Empty
  _ + Empty = Empty
  x + y = Interval { lo=lo x + lo y, hi=hi x + hi y }
  Empty * _ = Empty
  _ * Empty = Empty
  x * y = interval0f [ a * b | a <- [lo x, hi x], b <- [lo y, hi y] ]
  negate Empty = Empty
  negate x = Interval { lo=negate (hi x), hi=negate (lo x) }
  abs Empty = Empty
  abs x = interval0f [ abs (lo x), abs (hi x) ]
  signum Empty = Empty
  signum x = Interval { lo=signum (lo x), hi=signum (hi x) }
  fromInteger i = Interval { lo=n, hi=n } where n = fromInteger i

instance (Ord a, Fractional a) ⇒ Fractional (Interval a) where
  Empty / _ = Empty
  _ / Empty = Empty
  x / y | lo y <= 0 && 0 <= hi y = Interval { lo=InfNeg, hi=InfPos }
  | otherwise = interval0f zs
    where zs = [ a / b | a <- [lo x, hi x], b <- [lo y, hi y] ]
  fromRational r = Interval { lo=fromRational r, hi=fromRational r }

"Yoyak/Interval/Domain.hs" 157 lines --64%-- 102,0-1 64%
Simplicity

```erlang
*Yoyak.Interval.Domain> 1 :: Interval Int
Interval {lo = Num 1, hi = Num 1}
*Yoyak.Interval.Domain> let a = 1 :: Interval Int
*Yoyak.Interval.Domain> a + 1
Interval {lo = Num 2, hi = Num 2}
*Yoyak.Interval.Domain> 2*a + 10
Interval {lo = Num 12, hi = Num 12}
*Yoyak.Interval.Domain> (a + 1) +/- (2*a + 10)
Interval {lo = Num 2, hi = Num 12}
*Yoyak.Interval.Domain> let b = ((a + 1) +/- (2*a + 10)) - 5
*Yoyak.Interval.Domain> b
Interval {lo = Num (-3), hi = Num 7}
*Yoyak.Interval.Domain> b - b*b
Interval {lo = Num (-52), hi = Num 28}
*Yoyak.Interval.Domain> let c = b - b*b; d = b*b - b
*Yoyak.Interval.Domain> d
Interval {lo = Num (-28), hi = Num 52}
*Yoyak.Interval.Domain> c ** d
Interval {lo = Num (-28), hi = Num 28}
*Yoyak.Interval.Domain> d - 100
Interval {lo = Num (-128), hi = Num (-48)}
*Yoyak.Interval.Domain> ```
Simplicity

*Yoyak.Interval.Domain> a
Interval {lo = Num 1, hi = Num 1}
*Yoyak.Interval.Domain> b
Interval {lo = Num (-3), hi = Num 7}
*Yoyak.Interval.Domain> c
Interval {lo = Num (-52), hi = Num 28}
*Yoyak.Interval.Domain> d
Interval {lo = Num (-28), hi = Num 52}
*Yoyak.Interval.Domain> a <=^ b
Just True
*Yoyak.Interval.Domain> c <=^ b
Just False
*Yoyak.Interval.Domain> c <=^ d
Nothing
*Yoyak.Interval.Domain> c <=^ c++^d
Just True
*Yoyak.Interval.Domain> d <=^ c++^d
Just True
*Yoyak.Interval.Domain> c**^d <=^ c
Just True
*Yoyak.Interval.Domain> c**^d <=^ d
Just True
*Yoyak.Interval.Domain> ❋
Value as an AbsDom

data Value = Value { numValue :: Interval Float, addrValue :: [Address] }
  deriving (Show, Read, Eq)
data Address = VarAddr Var
  | CellAddr Region Index
  | FieldAddr Region FieldName
  deriving (Show, Read, Eq, Ord)

instance AbsDom Value where
  bottom = Value bottom bottom
  top = Value top top
  Value n1 a1 <=^ Value n2 a2 = (n1, a1) <=^ (n2, a2)
  v1 ++^ v2 = Value (numValue v1 ++^ numValue v2)
  (addrValue v1 ++^ addrValue v2)
  v1 **^ v2 = Value (numValue v1 **^ numValue v2)
  (addrValue v1 **^ addrValue v2)

instance AbsDom State where
  "Yoyak/Domain.hs" 80 lines --33%-- 27,0-1 69%
Simplicity
Transparency with Haskell Interactive
Observing Airac’s Behavior

- Hard to use ocamlmktop
- Hand-crafted pretty printer for each data type
  - tostring
- Ad-hoc “printf debugging”
  - prerr_endline
  - string_of_*
  - ^
- No easy way to input values for testing
Observing Yoyak’s Behavior

• Show, Read classes
• With a Haskell Interactive, such as ghci
• And the HumanFriendly class
Haskell’s Support

• Generates instances of **Show & Read**
  
  • data Interval a = ...
    deriving (Show, Read, ...)

• Easy to observe and test parts
  
  • (even outside the interactive environment)
import Yoyak.Syntax.HumanFriendly

use :: HumanFriendly a => String -> a
use = either (error.show) id . parse parser ""

see :: HumanFriendly a => a -> IO ()
see = putStrLn . present

class HumanFriendly a where
    present :: a -> String
    present a = presentPrec 0 a ""
    presentPrec :: Int -> a -> ShowS
    presentPrec _ a = (present a ++)
Transparency with Read & Show
Transparency with HumanFriendly
Summary

• I’m working on building Yoyak Engine

• into Multiple Programs
  • Master + Workers
  • for Scalability, i.e. more resources = more output per time

• leveraging Haskell’s features, for
  • Simplicity, i.e. shorter code to read and write
  • Transparency, i.e. easier debugging, testing
Thank you

Q & A
References

- Process (computing) - Wikipedia
- Scalability - Wikipedia
- A Gentle Introduction to Haskell, Version 98
  - 5. Type Classes and Overloading
  - 8.3. The Read and Show Classes
  - 10. Numbers
- Rule of Transparency - Basics of the Unix Philosophy