Measuring Software Complexity by Types

PÁLI Gábor János   KOZSIK Tamás

Department of Programming Languages and Compilers,
Eötvös Loránd University, Budapest

Chalmers Functional Programming Workshop 2010
Gothenburg, Sweden
June 2, 2010

Why?

- Types are present in most programming languages and they are already exploited in many ways. *Interesting.*
  - *abstraction* $\rightarrow$ thinking in higher-level terms
  - *safety* $\rightarrow$ more consistent software
  - *optimization* $\rightarrow$ more fine-tuned software
  - *documentation* $\rightarrow$ description of the software
- Other researches (Van Den Berg (1995), Ryder (2004), Király (2010)) work with call graphs and flowgraphs. *Strange.*
- It would be nice if we can control the (ab)use of abstractions, a factor of software quality – but "You cannot control what you cannot measure."[1] *Clear.*

So why not?

[1] Tom DeMarco

Featuring: A Star[2]

```haskell
import Control.Monad (guard, liftM2)
import Data.List (elemIndex)
import qualified Data.Set as S
import qualified Data.Map as M
import qualified PriorityQueue as Q

type Point = (Int, Int)

type Map = [[Char]]

main :: IO ()
main = interact doit
heuristic :: Point -> Point -> Int
heuristic (x, y) (u, v) = abs (x - u) `max` abs (y - v)

astar :: (Ord a, Ord b, Num b) =>
  a -> (a -> [a]) -> (a -> Bool)
    -> (a -> b) -> (a -> b) -> [a]
astar s succ end cost heur
 = astar' (S.singleton s) (Q.singleton (heur s) [s])
 where
   astar' seen q
     | Q.null q   = error "No Solution."
     | end n      = next
     | otherwise = astar' seen' q'
    where
     ((c,next), dq) = Q.deleteFindMin q
     n      = head next
     succs = filter (`S.notMember` seen) $ succ n
     calc  = (+ c) . (subtract $ heur n) .
             liftM2 (+) cost heur
     costs = map calc succs
     nexts = map (: next) succs
     czs   = Q.fromList (zip costs nexts)
     q'    = dq `Q.union` czs
     seen' = seen `S.union` S.fromList succs
```

```haskell
find :: Char -> Map -> Point
find c m = find' 0 m
 where find' _ [] = error "Cannot find tile."
       find' y (h:t)
        | Just x <- elemIndex c h = (y, x)
        | otherwise = find' (y + 1) t

successor :: Map -> Point -> [Point]
successor m (x,y)
 = do u <- [x + 1, x, x - 1]
      v <- [y + 1, y, y - 1]
      guard (0 <= u && u < length m)
      guard (0 <= v && v < length (head m))
      guard (u /= x || y /= v)
      guard (m !! u !! v /= '~')
      return (u, v)
path :: Map -> [Point] -> Map
path m l = iterY m l 0
 where
  iterY [] _ _       = []
  iterY (h:t) l n    = iterX h l n 0 : iterY t l (n + 1)
  iterX [] _ _ _     = []
  iterX (h:t) l n m = pick : iterX t l n (m + 1)
    where pick = if (n,m) `elem` l then '#' else h)
doit :: String -> String
doit s = unlines . path m $ astar start succ (== end) cost h
 where
  m            = lines s
  start        = find '@' m
  end          = find 'X' m
  succ         = successor m
  h            = heuristic end
  cost (x, y) = costsM M.! (m !! x !! y)
  costsM       = M.fromList [('@',1),('x',1),('X',1)
                            ,('.',1),('*',2),('^',3)]
```

[2] http://www.haskell.org/haskellwiki/Haskell_Quiz/Astar

Recipe for Measuring Types

- Pick all the types for each identifier in the program.

```
main :: IO ()
heuristic :: Point -> Point -> Int
find :: Char -> Map -> Point
...
```

- Break types for functions into pieces.

```
heuristic :: Point -> Point -> Int
```
$\rightarrow$ `Point`, `Point`, `Int`

```
...
```

- Put them in a bag.
- Boil down some metrics from them.

Mmm, delicious.

Recurring: A Star

```haskell
import Control.Monad (guard, liftM2)
import Data.List (elemIndex)
import qualified Data.Set as S
import qualified Data.Map as M
import qualified PriorityQueue as Q
type Point = (Int, Int)
type Map = [[Char]]
main :: IO ()
main = interact doit
heuristic :: Point -> Point -> Int
heuristic (x, y) (u, v) = abs (x - u) `max` abs (y - v)
astar :: (Ord a, Ord b, Num b) =>
  a -> (a -> [a]) -> (a -> Bool)
    -> (a -> b) -> (a -> b) -> [a]
astar s succ end cost heur
 = astar' (S.singleton s) (Q.singleton (heur s) [s])
 where
   astar' seen q
     | Q.null q  = error "No Solution."
     | end n     = next
     | otherwise = astar' seen' q'
    where
     ((c,next), dq) = Q.deleteFindMin q
     n     = head next
     succs = filter (`S.notMember` seen) $ succ n
     calc  = (+ c) . (subtract $ heur n) .
             liftM2 (+) cost heur
     costs = map calc succs
     nexts = map (: next) succs
     czs   = Q.fromList (zip costs nexts)
     q'    = dq `Q.union` czs
     seen' = seen `S.union` S.fromList succs
```

```haskell
find :: Char -> Map -> Point
find c m = find' 0 m
 where find' _ [] = error "Cannot find tile."
       find' y (h:t)
        | Just x <- elemIndex c h = (y, x)
        | otherwise = find' (y + 1) t
successor :: Map -> Point -> [Point]
successor m (x,y)
 = do u <- [x + 1, x, x - 1]
      v <- [y + 1, y, y - 1]
      guard (0 <= u && u < length m)
      guard (0 <= v && v < length (head m))
      guard (u /= x || y /= v)
      guard (m !! u !! v /= '~')
      return (u, v)
path :: Map -> [Point] -> Map
path m l = iterY m l 0
 where
  iterY [] _ _      = []
  iterY (h:t) l n   = iterX h l n 0 : iterY t l (n + 1)
  iterX [] _ _ _    = []
  iterX (h:t) l n m = pick : iterX t l n (m + 1)
    where pick = if (n,m) `elem` l then '#' else h)
doit :: String -> String
doit s = unlines . path m $ astar start succ (== end) cost h
 where
  m           = lines s
  start       = find '@' m
  end         = find 'X' m
  succ        = successor m
  h           = heuristic end
  cost (x, y) = costsM M.! (m !! x !! y)
  costsM      = M.fromList [('@',1),('x',1),('X',1)
                           ,('.',1),('*',2),('^',3)]
```

Contents of the Bag

α	17	$\alpha^{MonadPlus}$	4
Point	16	α^{Ord}	4
Map	15	$\alpha^{Ord,Num}$	4
$\alpha^{Num,Eq,Show}$	13	((α^{Ord},[β]),PriorityQueue α^{Ord} [β])	4
Bool	11	String	3
[α]	11	Maybe α	2
[Int]	10	[α^{Ord}]	2
α^{Num}	10	Map α	2
[Point]	7	α^{Monad}	2
Set α	6	$\alpha^{Eq,Ord}$	1
PriorityQueue α^{Ord} β	6	Char	1
[String]	5	[(α,β)]	1
Int	5	IO α	1

Metrics #1: Complexity

α	17	$\alpha^{MonadPlus}$	4
Point	16	α^{Ord}	4
Map	15	$\alpha^{Ord,Num}$	4
$\alpha^{Num,Eq,Show}$	13	$((\alpha^{Ord},[\beta]),\text{PriorityQueue } \alpha^{Ord}\ [\beta])$	4
Bool	11	String	3
$[\alpha]$	11	Maybe α	2
[Int]	10	$[\alpha^{Ord}]$	2
α^{Num}	10	Map α	2
[Point]	7	α^{Monad}	2
Set α	6	$\alpha^{Eq,Ord}$	1
PriorityQueue α^{Ord} β	6	Char	1
[String]	5	$[(\alpha,\beta)]$	1
Int	5	IO α	1

The complexity is 26 *(ELOC*[3]*: 64). It means approx. 0.4 type introduced per line.*

[3] Effective Lines of Code.

Metrics #2: Distribution of Types

α	0.104	$\alpha^{MonadPlus}$	0.026
Point	0.098	α^{Ord}	0.026
Map	0.092	$\alpha^{Ord,Num}$	0.026
$\alpha^{Num,Eq,Show}$	0.079	$((\alpha^{Ord},[\beta]),\text{PriorityQueue } \alpha^{Ord}\ [\beta])$	0.026
Bool	0.067	String	0.018
$[\alpha]$	0.067	Maybe α	0.012
[Int]	0.061	$[\alpha^{Ord}]$	0.012
α^{Num}	0.061	Map α	0.012
[Point]	0.043	α^{Monad}	0.012
Set α	0.037	$\alpha^{Eq,Ord}$	0.006
PriorityQueue α^{Ord} β	0.037	Char	0.006
[String]	0.030	$[(\alpha,\beta)]$	0.006
Int	0.030	IO α	0.006

This program uses its own data types nicely, it also contains some generic components, and mostly works with lists and Boolean expressions.

Metrics #3: Important Types

α	0.104	$\alpha^{MonadPlus}$	0.026
Point	0.098	α^{Ord}	0.026
Map	0.092	$\alpha^{Ord,Num}$	0.026
$\alpha^{Num,Eq,Show}$	0.079	$((\alpha^{Ord}, [\beta]), \text{PriorityQueue } \alpha^{Ord}\ [\beta])$	0.026
Bool	0.067	String	0.018
$[\alpha]$	0.067	Maybe α	0.012
[Int]	0.061	$[\alpha^{Ord}]$	0.012
α^{Num}	0.061	Map α	0.012
[Point]	0.043	α^{Monad}	0.012
Set α	0.037	$\alpha^{Eq,Ord}$	0.006
PriorityQueue $\alpha^{Ord}\ \beta$	0.037	Char	0.006
[String]	0.030	$[(\alpha,\beta)]$	0.006
Int	0.030	IO α	0.006

Most used types: `Point`, `Map`, `Bool`, α, $[\alpha]$, [`Int`].

In Retrospect: A Star

```haskell
import Control.Monad (guard, liftM2)
import Data.List (elemIndex)
import qualified Data.Set as S
import qualified Data.Map as M
import qualified PriorityQueue as Q

type Point = (Int, Int)
type Map = [[Char]]

main :: IO ()
main = interact doit
heuristic :: Point -> Point -> Int
heuristic (x, y) (u, v) = abs (x - u) `max` abs (y - v)

astar :: (Ord a, Ord b, Num b) =>
  a -> (a -> [a]) -> (a -> Bool)
  -> (a -> b) -> (a -> b) -> [a]
astar s succ end cost heur
 = astar' (S.singleton s) (Q.singleton (heur s) [s])
 where
   astar' seen q
     | Q.null q  = error "No Solution."
     | end n     = next
     | otherwise = astar' seen' q'
    where
     ((c,next), dq) = Q.deleteFindMin q
     n     = head next
     succs = filter (`S.notMember` seen) $ succ n
     calc  = (+ c) . (subtract $ heur n) .
             liftM2 (+) cost heur
     costs = map calc succs
     nexts = map (: next) succs
     czs   = Q.fromList (zip costs nexts)
     q'    = dq `Q.union` czs
     seen' = seen `S.union` S.fromList succs
```

```haskell
find :: Char -> Map -> Point
find c m = find' 0 m
 where find' _ [] = error "Cannot find tile."
       find' y (h:t)
        | Just x <- elemIndex c h = (y, x)
        | otherwise = find' (y + 1) t
successor :: Map -> Point -> [Point]
successor m (x,y)
 = do u <- [x + 1, x, x - 1]
      v <- [y + 1, y, y - 1]
      guard (0 <= u && u < length m)
      guard (0 <= v && v < length (head m))
      guard (u /= x || y /= v)
      guard (m !! u !! v /= '~')
      return (u, v)
path :: Map -> [Point] -> Map
path m l = iterY m l 0
 where
  iterY [] _ _     = []
  iterY (h:t) l n  = iterX h l n 0 : iterY t l (n + 1)
  iterX [] _ _ _   = []
  iterX (h:t) l n m = pick : iterX t l n (m + 1)
    where pick = if (n,m) `elem` l then '#' else h
doit :: String -> String
doit s = unlines . path m $ astar start succ (== end) cost h
 where
  m           = lines s
  start       = find '@' m
  end         = find 'X' m
  succ        = successor m
  h           = heuristic end
  cost (x, y) = costsM M.! (m !! x !! y)
  costsM      = M.fromList [('@',1),('x',1),('X',1)
                           ,('.',1),('*',2),('^',3)]
```

- The complexity is 26 (ELOC: 64). It means approx. 0.4 type introduced per line.
- This program uses its own data types nicely, it also contains some generic components, and mostly works with lists and Boolean expressions.
- Most used types: `Point`, `Map`, `Bool`, α, $[\alpha]$, `[Int]`.

Thank you for your attention!

feel free to ask questions or comment