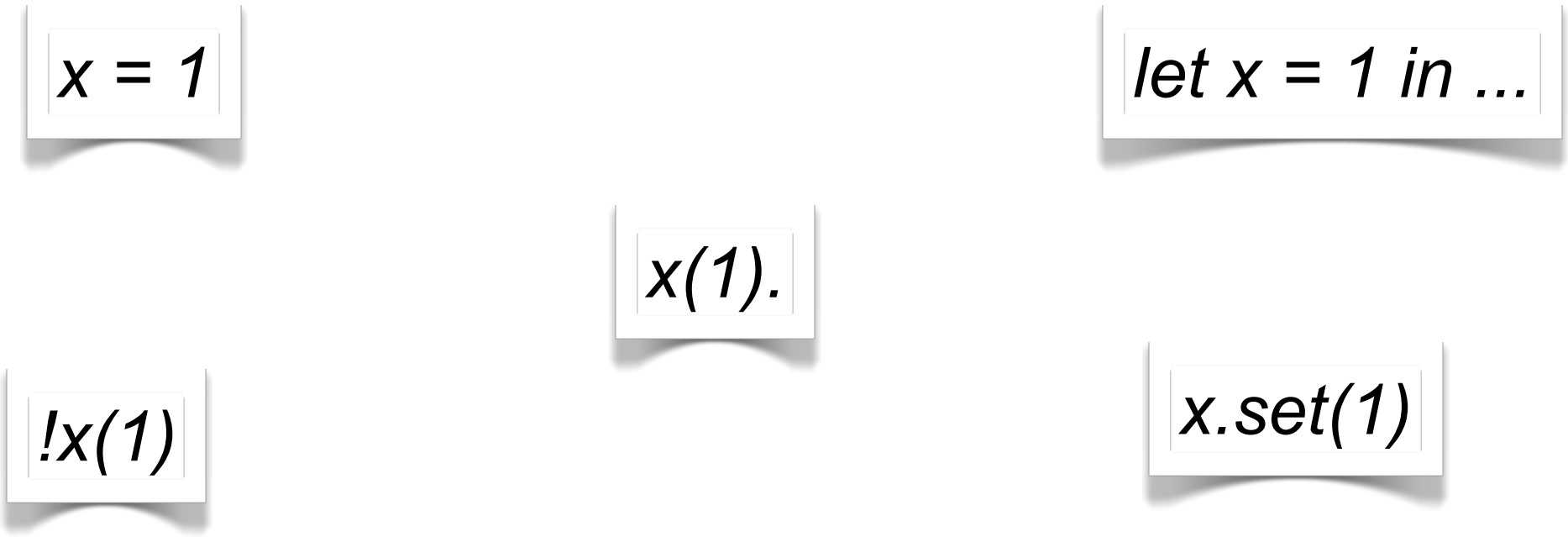


Programming Language Theory

Preparation for the Final

Ralf Lämmel

Lectures covered

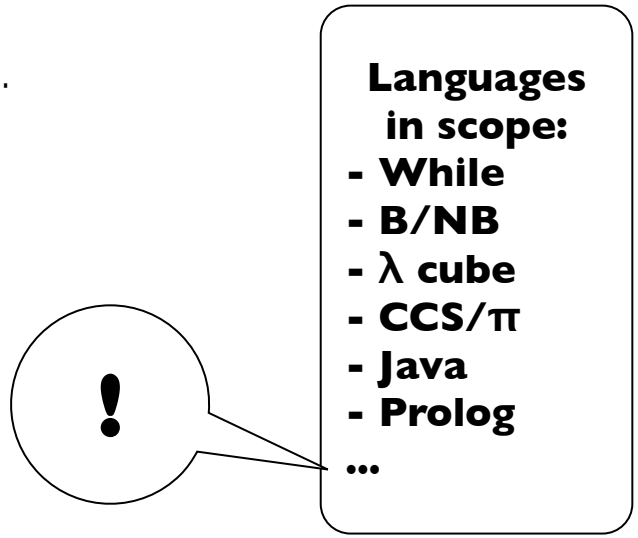
- **Introduction to Haskell**
- **Denotational Semantics**
- **Program Analysis**
- **Monads**

Underlying principles

- Heavily based on sketches in **Haskell**.
  - ✦ "No text", "No Multiple Choice"
- Based on subjects/skills covered by assignments.
- Many concepts and intuitions from lecture needed.

Categories of questions for ***final***
(**0-2** questions per category, 9 questions in total)

1. Infer the Haskell **type** of the given expression.
2. Define a **list-processing function** as described.
3. Define **type-class instances** for the given problem.
4. Transform the given function into **fixed point form**.
5. Represent the **abstract syntax** of given constructs in Haskell.
6. Define a **denotational semantics** for given constructs in Haskell.
7. Transform the given function into **continuation style**.
8. Define a **program analysis** for the given problem.
9. Solve a **semantics riddle** with a succinct argument.



1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-
13	-
14	-
15	4,0
16	3,7
17	3,7
18	3,3
19	3,3
20	3,0
21	2,7
22	2,7
23	2,3
24	2,3
25	2,0
26	1,7
27	1,7
28	1,3
29	1,3
30	1,0
31	1,0
32	1,0

Grading rules (midterm+final=resit)

- One final grade
- 0-2 points per question
  - 0 "missing or mental assault"
  - 1 "the beginning of an idea"
  - 2 "nearly or fully complete/correct"
- 1 possible extra point per exam
  - for an "outstanding solution"
- 6 questions for midterm (12 points, 40 %)
- 9 questions for final (18 points, 60 %)
- 30 points in total + 2 extra points

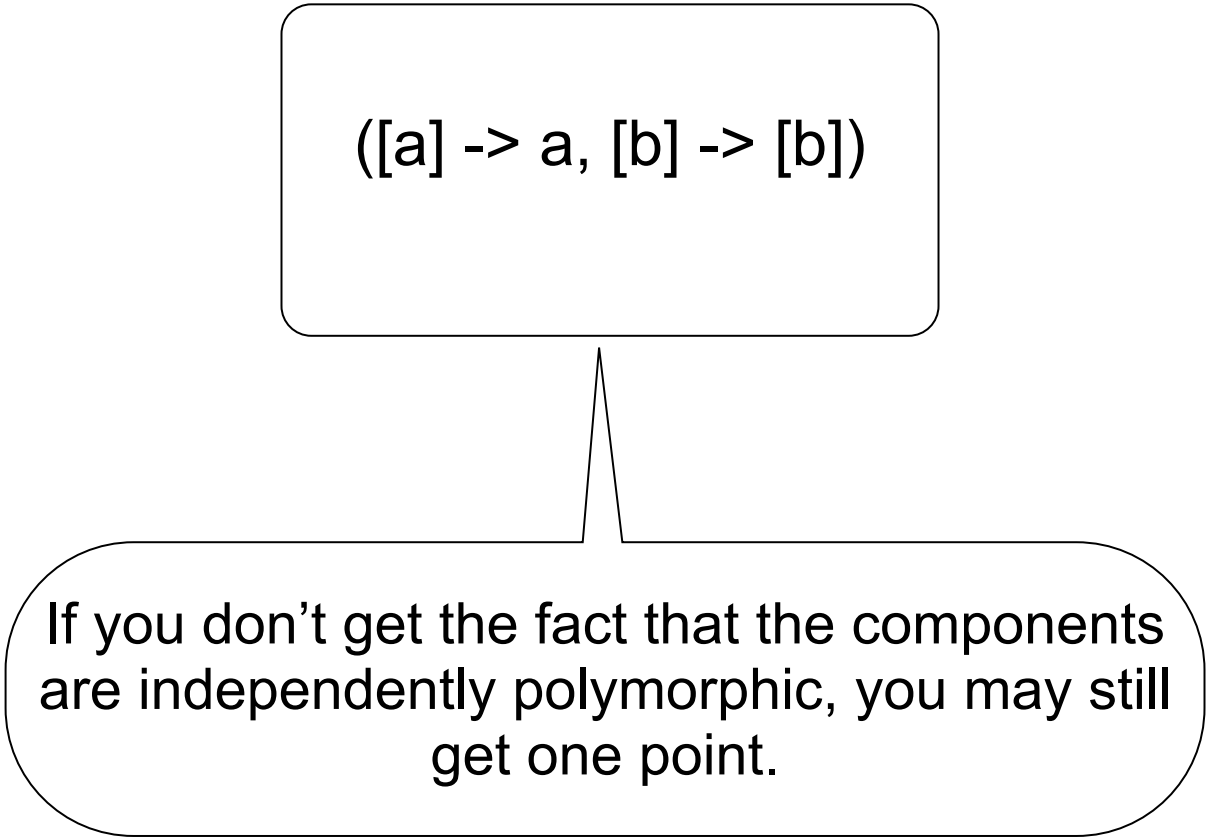
Samples questions and answers

Infer the Haskell type of the given expression.

Infer the Haskell type of the given expression.

(head, tail)

Solution



Infer the Haskell type of the given expression.

```
head . filter fst
```

Solution

head :: [a] -> a

filter :: (b -> Bool) -> [b] -> [b]

fst :: (c,d) -> c

(.) :: (g -> f) -> (e -> g) -> e -> f

(.) head :: (e -> [a]) -> e -> a

because g -> f = [a] -> a and hence g = [a], f = a

filter fst :: [(Bool,d)] -> [(Bool,d)]

because (b -> Bool) = (c,d) -> c and hence c = Bool and b = (Bool,d)

(.) head (filter fst) :: [(Bool,d)] -> (Bool,d)

because (e -> [a]) = [(Bool,d)] -> [(Bool,d)]

and hence e = [(Bool,d)] and a = (Bool,d)

Define a list-processing function as described.

Define a list-processing function as described.

Define a predicate for membership test using **foldr**. Given a value *x* and a list *l* of values, the predicate determines whether *x* appears in *l*.

Simple solution

```
member :: Eq a => a -> [a] -> Bool
member x l = foldr f False l
 where
  f y r = x == y || r
```

Another solution

```
member :: Eq a => a -> [a] -> Bool
member x = or . map (x==)
```

map is defined is based on foldr.

Define a list-processing function as described.

Provide a function *skippy* that takes a list and returns a list with the even indexes of the list (starting to count at 0).

For example:

skippy ["a","b","c","d"] = [“a”,”c”]

You can define the function any way you like.

Solution

```
skippy [] = []

skippy (x0:[]) = [x0]

skippy (x0:x1:xs) = x0:skippy xs
```

Define type-class instances for the given problem.

Define type-class instances for the given problem.

This type class models the extraction of all ints from a given value. **Add instances for lists and pairs.**

```
class ToInts x
 where
  toInts :: x -> [Int]

instance ToInts Int
 where
  toInts i = [i]

instance ToInts Bool
 where
  toInts = const []
```

Solution

```
instance ToInts a => ToInts [a]
 where
  toInts = concat . map toInts

instance (ToInts a, ToInts b) => ToInts (a,b)
 where
  toInts (a,b) = toInts a ++ toInts b
```

Make sure you understand type-class instances for polymorphic or recursive types, and the need for instance constraints in many such cases.

Define type-class instances for the given problem.

```
class Size x
 where
  size :: x -> Int

instance Size Bool
 where
  size = const 1

instance (Size a, Size b) => Size (a,b)
 where
  size (a,b) = size a + size b + 1
```

This type class counts constructors in terms. We assume that primitive values count as 1. **Define instances for Int and lists.**

Solution

```
instance Size Int
 where
  size = const 1

instance Size a => Size [a]
 where
  size = (+) 1 . sum . map size
```

Instead (+1), we may also count length + 1 for the cons'es.

Less point-free code is also acceptable, but you may need to remember sum, map, and friends in order to quickly right down the solution.

Transform the given function into fixed point form.

Transform the given function into fixed point form.

Rephrase the following definition of append (which uses direct recursion) such that it uses Haskell's fixed point combinator (also shown below):

```
append :: [a] -> [a] -> [a]
append [] l = l
append (h:t) l = h : append t l
```

```
fix :: (t -> t) -> t
fix f = f (fix f)
```

Solution

```
append l1 l2 = fix append' l1
 where
  append' _ []    = l2
  append' f (h:t) = h : f t
```

This is already a complicated example because it involves a function with two parameters with some tricky order. You should count on something more straightforward.

Represent the abstract syntax of given constructs in Haskell.

Define a denotational semantics for given constructs in Haskell.

Define a denotational semantics for given constructs in Haskell.

Consider the following abstract syntax of a simple state machine (think of Java byte code):

```
data Code = Push Int       -- push an element onto the stack
          | Add            -- replace topmost elements by sum
          | Seq Code Code  -- left-to-right composition
          | While2 Code    -- loop until stack size < 2
```

Define a denotational semantics so that the following main program would print a stack with the single element 10. Stacks are represented as lists.

```
main =
 do
    print $
     exec (Seq (Push 1)
          (Seq (Push 2)
          (Seq (Push 3)
          (Seq (Push 4)
               (While2 Add))))) []
```

Solution

```
exec :: Code -> [Int] -> [Int]
exec (Push i)    = (i:)
exec Add         = \(i1:i2:s) -> (i1+i2:s)
exec (Seq c1 c2) = exec c2 . exec c1
exec (While2 c)  = fix f
 where
  f g s = if length s < 2
            then s
            else g (exec c s)

fix f = f (fix f)
```

Correct handling of while may be enough for an excellent solution.

In the actual exam, you do not have to write so much code necessarily. Some parts may be given with elisions indicated.

Transform the given function into continuation style.

Continuations from a ***programming*** perspective

- "*Evolution of a Haskell program*"

  for solving $x^2 + p\,x + q = 0$

  - ✦ Solve the equation; don't care about negative discriminant.
  - ✦ Anticipate failure in discriminant's computation and its consumer.
  - ✦ Localize error handling in the discriminant's computation.

Solve the equation; don't care about negative discriminant.

```haskell
-- Solve quadratic equation
qequation :: Double -> Double -> (Double, Double)
qequation p q = (x1,x2)
 where
  d = discriminant p q
  x1 = fst'solution p d
  x2 = snd'solution p d

-- Compute discriminant
discriminant :: Double -> Double -> Double
discriminant p q = sqrt $ p * p / 4 - q

-- Map p and discriminant to first solution
fst'solution :: Double -> Double -> Double
fst'solution p d = (-p) / 2 + d

-- Map p and discriminant to first solution
snd'solution :: Double -> Double -> Double
snd'solution p d = (-p) / 2 - d

-- Time to test
main = do
          print $ qequation 2 (-8)
          print $ qequation 2 2
```

Sqrt operation may be undefined.

```
>main
(2.0,-4.0)
(NaN,NaN)
```

- $x^2 + p\,x + q = 0$

Anticipate failure in discriminant's computation and its consumer.

```haskell
-- Solve quadratic equation
qequation :: Double -> Double -> Maybe (Double, Double)
qequation p q =
 case discriminant p q of
  Nothing -> Nothing
  Just d' -> Just (x1 d', x2 d')
 where
  x1 = fst'solution p
  x2 = snd'solution p

-- Compute discriminant
discriminant :: Double -> Double -> Maybe Double
discriminant p q =
 if t < 0
  then Nothing
  else Just (sqrt t)
 where t = p * p / 4 - q
```

Failure is now at the top of the result.

Error handling

Error handling

```
>main
Just (2.0,-4.0)
Nothing
```

Localize error handling in the discriminant's computation.

```haskell
-- Solve quadratic equation
qequation :: Double -> Double -> Maybe (Double, Double)
qequation p q = discriminant p q (\d -> (x1 d, x2 d))
 where
  x1 = fst'solution p
  x2 = snd'solution p

-- Compute discriminant
discriminant :: Double -> Double -> (Double -> r) -> Maybe r
discriminant p q k =
 if t < 0
  then Nothing
  else Just $ k (sqrt t)
 where t = p * p / 4 - q
```

Pass continuation

Error handling

Invoke continuation under normal circumstances

```
>main
Just (2.0,-4.0)
Nothing
```

Pythagoras in direct style

```
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

> pythagoras 3 4
25

Pythagoras in continuation (passing) style

```
add'cps :: Int -> Int -> (Int -> r) -> r
add'cps x y k = k (add x y)

square'cps :: Int -> (Int -> r) -> r
square'cps x k = k (square x)

pythagoras'cps :: Int -> Int -> (Int -> r) -> r
pythagoras'cps x y k =
 square'cps x $ \x'squared ->
 square'cps y $ \y'squared ->
 add'cps x'squared y'squared $ \sum'of'squares ->
 k sum'of'squares
```

> pythagoras'cps 3 4 print
25

Transform the given function into continuation style.

Rephrase the following find function so that it takes a continuation to process the Int and to use an arbitrary result type.

```
find :: [(String,Int)] -> String -> Maybe Int
find [] s = Nothing
find ((k,v):l) s = if k==s then Just v else find l s
```

Solution

```
find' :: [(String, Int)] -> String -> (Int -> r) -> Maybe r
find' [] x f = Nothing
find' ((x',v):t) x f = if x'==x then Just (f v) else find' t x f
```

```
> find' [("x",42),("y",88)] "x" id
Just 42
> find' [("x",42),("y",88)] "x" (+1)
Just 43
> find' [("x",42),("y",88)] "z" (+1)
Nothing
```

Demo

In the actual exam, you are likely to get a bit more help (by means of a more detailed explanation, the type of find' etc.)

Transform the given function into continuation style.

Consider the following functional interpreter for a tiny functional language. It exposes one bug: "Return i" does not properly terminate the computation. Fix this problem by migrating to continuation style.

```
data Exp = Id | Plus1 | Dot Exp Exp | Return Int

eval :: Exp -> Int -> Int
eval Id                          = id
eval Plus1                       = (+1)
eval (Dot f g)           = eval f . eval g
eval (Return i)          = const i

> eval (Dot Plus1 (Dot Plus1 (Return 1))) 42
3
```

Wrong result!
We want "1" instead.

Solution

```
eval' :: Exp -> (Int -> Int) -> Int -> Int
eval' Id                     = id
eval' Plus1                  = flip (.) (+1)
eval' (Dot f g)              = eval' g . eval' f
eval' (Return i)             = const $ const i

> eval' (Dot Plus1 (Dot Plus1 (Return 1))) id 42
1
```

In the actual exam, you are likely to get a bit more help (by means of a more detailed explanation, the type of eval' etc.)

Define a program analysis for the given problem.

Define a program analysis for the given problem.

For a simple imperative syntax, as shown, please check whether programs *may* loop in the sense that they contain while loops with bodies lacking any assignments. The test for a statement to lack assignments is given already. Implement the function mayLoop.

```
data Stm = Skip
         | Assign String Exp
         | Seq Stm Stm
         | If Exp Stm Stm
         | While Exp Stm

data Exp = ...
```

```
lacksAssign :: Stm -> Bool
mayLoop :: Stm -> Bool

lacksAssign Skip         = True
lacksAssign (Assign _ _) = False
lacksAssign (Seq s1 s2)  = lacksAssign s1 && lacksAssign s2
lacksAssign (If _ s1 s2) = lacksAssign s1 && lacksAssign s2
lacksAssign (While _ s)  = lacksAssign s
```

Solution

```
mayLoop Skip          = False
mayLoop (Assign _ _) = False
mayLoop (Seq s1 s2)  = mayLoop s1 || mayLoop s2
mayLoop (If _ s1 s2) = mayLoop s1 || mayLoop s2
mayLoop (While _ s)  = lacksAssign s || mayLoop s
```

In the actual exam, you should typically write less code than shown above. To this end, some trivial cases may be prepared for you.

Define a program analysis for the given problem.

Rather than defining a complete analysis, the assignment may also be concerned with an important building block of an analysis -- specifically a **complete lattice**.

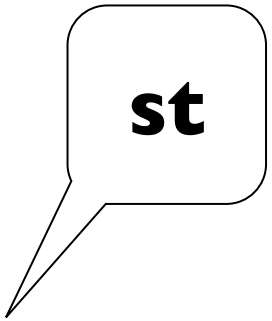
For instance, define a partial order, leq, for the complete lattice of special Booleans with least element Bottom, greatest element Top and incomparable values in between : True' and False'.

```
data Bool' = Bottom | True' | False' | Top
```

Solution

```
leq :: Bool' -> Bool' -> Bool
leq Bottom _       = True
leq _ Top          = True
leq True' True'    = True
leq False' False'  = True
leq _ _            = False
```

"Solve a semantics riddle with a succinct argument."



Logistics

- *10am-12pm, 8 Feb 2012, Room E114.*
- Two rounds; same way as last time.
- No phones, computers, electronics, books, notes, etc.
- You must bring your student ID.
- No need to formally register / deregister.
- You can choose to do the exam for real or for fun.
- Reference solution will be published.
- Teaching assistant will organize exam access.

All the best for the exam.

Make sure to talk to me about research projects.