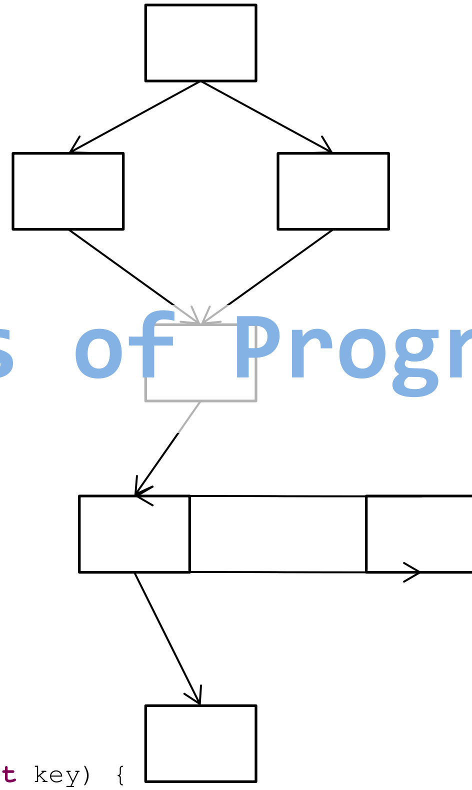
Foundations of Program Analysis

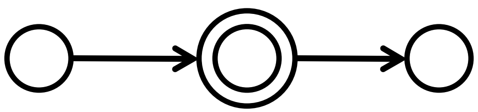
$$\frac{\{A \wedge b\}\ c\ \{A\}}{\{A\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{A \wedge !b\}}$$

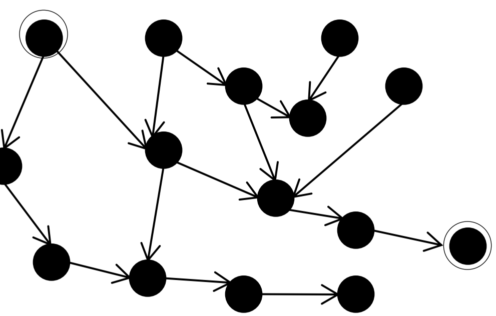


$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'' \qquad \langle c_1, \sigma'' \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \sigma'}$$

$P(x) \wedge !q(y)$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$





```
Node find (Node prev, Node cur, int key) {
    while (cur.key < key) {
        prev = cur;
        cur = cur.next;
    }
    return cur;
}
```

6.820

Armando Solar-Lezama

Course Staff

- **Armando Solar-Lezama**
  - **Instructor**

What this course is about

The top N good ideas in programming languages that you might be embarrassed not to know about. ;)

What this course is about

- How we define the meanings of programs and programming languages unambiguously?
- How can we prove theorems about the behavior of individual programs?
- How can we design programming tools to automate that kind of understanding?

Applications:

- finding bugs
- designing languages to prevent bugs
- synthesizing programs
- manipulating programs automatically (refactoring, optimization)

Course outline

Functional Programming

- learn about lambda calculus, Haskell, and OCaml
- learn to make formal arguments about program behavior

Type Theory

- learn how to design and reason about type systems
- use type-based analysis to find synchronization errors, avoid information leaks and manage your memory efficiently

Axiomatic Semantics/Program Logics

- a different view of program semantics
- learn how to make logical arguments about program correctness

Course Outline

Abstract Interpretation

- use abstraction to reason about the behavior of the program under all possible inputs

Model checking

- learn how to reason exhaustively about program states
- learn how abstraction and symbolic reasoning can help you find bugs in device drivers and protocol designs

Big Ideas (recurring throughout the units)

Operational Semantics

(give programs meanings via stylized *interpreters*)

Program Proofs as Inductive Invariants

(all induction, all the time!)

Abstraction

(model programs with *specifications*)

Modularity

(break programs into *pieces* to analyze separately)

Skills

- Haskell
- Coq
- Ocaml
- Spin

Grading

6 homework assignments

- Each is 15-20% of your grade
- start on them early!

6 Homework Assignments

Pset 1 (out now, due in about 2 weeks!)

- Practice functional programming
- Build some Lambda Calculus interpreters

Pset 2

- Practice more functional programming
- Implement a type inference engine
- Practice writing proofs in Coq

Pset 3

- How to make formal arguments about the properties of a type system
- Coq proof of type safety for a simple language

Pset 4

- Learn about SMT solvers
- Implement your own verifier for simple C-like programs

Homework Assignments Cont.

Pset 5

- Implement an analysis to check for memory errors in C-like programs

Pset 6

- Practice LTL and CTL (two specification languages)
- Learn how to use a model checker

Functional Programming: Functions and Types

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Adapted from Arvind 2010

September 9, 2015

Function Execution by Substitution

```
plus x  y = x + y

1.   plus  2  3  →  2 + 3  →  5

2.   plus  (2*3)  (plus 4 5)
```

```
→ plus 6 (4+5)
→ plus 6 9
→ 6 + 9
→ 15
```

```
→ (2*3) +(plus 4 5)
→ 6 + (4+5)
→ 6 + 9
→ 15
```

The final answer did not depend upon the order in which reductions were performed

Confluence

Informally - The order in which reductions are performed in a Functional program does not affect the final outcome

This is true for all functional programs regardless whether they are right or wrong

A formal definition will be given later

Blocks

```
let
    x =  a * a
    y =  b * b
in
    (x - y)/(x + y)
```

- a variable can have at most one definition in a block
- ordering of bindings does not matter

Layout Convention in Haskell

This convention allows us to omit many delimiters

```
let
    x =  a * a
    y =  b * b
in
    (x - y)/(x + y)
```

is the same as

```
let
    { x =  a * a ;
      y =  b * b ;}
in
    (x - y)/(x + y)
```

Lexical Scoping

```
let
    y = 2 * 2
    x = 3 + 4
    z = let
          x = 5 * 5
          w = x + y * x
        in
          w
in
    x + y + z
```

def

use

Lexically closest definition of a variable prevails.

Renaming Bound Identifiers
(α-renaming)

```
let
  y = 2 * 2
  x = 3 + 4
  z = let
        x = 5 * 5
        w = x + y * x
      in
        w
in
   x + y + z
```

≡

```
let
  y = 2 * 2
  x = 3 + 4
  z = let
        x′ = 5 * 5
        w = x′ + y * x′
      in
        w
in
   x + y + z
```

Lexical Scoping and α-renaming

```
plus  x y = x + y

plus' a b = a + b
```

`plus` and **`plus'`** are the same because **`plus'`** can be obtained by *systematic renaming of bounded identifiers* of **`plus`**

Capture of Free Variables

```
f x = . . .
g x = . . .
foo f x = f (g x)
```

Suppose we rename the bound identifier **`f`** to **`g`** in the definition of **`foo`**

```
foo' g x = g (g x)
```

`foo` ≡ **`foo'`** ? No

While renaming, entirely new names should be introduced!

Curried functions

```
plus x  y =  x + y

let
      f  = plus 1
 in
      f 3
```

→ **(plus 1) 3 → 1 + 3 → 4**

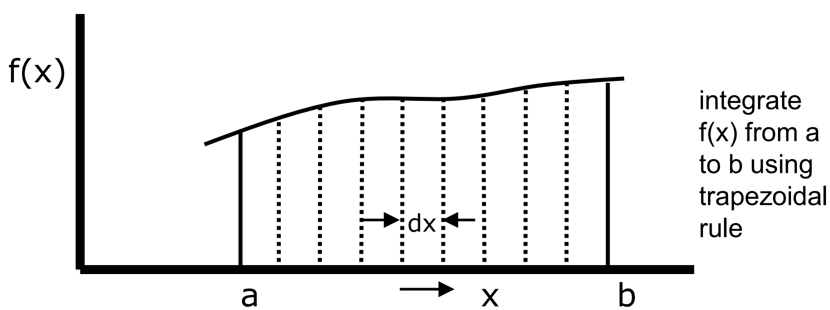
syntactic conventions:

e1 e2 e3   ≡   ((e1 e2) e3)

x + y ≡ (+) x y

Local Function Definitions

```
integrate dx a b f =
     let
        sum x tot =
             if x > b then tot
             else sum (x+dx) (tot+(f x))
     in
        (sum (a+dx/2) 0) * dx
```



Integral(a,b) = (f(a + dx/2) + f(a + 3dx/2) + ...) ☐ dx

Local Function Definitions

Free variables of **sum**?

```
integrate dx a b f =
      let
          sum x tot =
                if x > b then tot
                else sum (x+dx) (tot+(f x))
      in
          (sum (a+dx/2) 0) * dx
```

```
integrate dx a b f =
      (sum dx b f (a+dx/2) 0) * dx

sum dx b f x tot =
      if x > b then tot
      else sum dx b f (x+dx) (tot+(f x))
```

Any function definition can be "closed" and "lifted"

Types

All expressions in Haskell have a type

```
23 :: Int
```

"`23` belongs to the set of integers"
"The type of `23` is `Int`"

```
   true :: Bool
"hello" :: String
```

Type of an expression

```
(sq 529)   :: Int
 sq        :: Int -> Int
```

"**`sq`** is a function, which when applied to an integer produces an integer"

"**`Int -> Int`** is the set of functions, each of which when applied to an integer produces an integer"

"The type of **`sq`** is **`Int -> Int`**"

Type of a Curried Function

```
plus x  y =  x + y

(plus 1) 3:: Int

(plus 1)  :: Int -> Int

 plus     :: Int -> (Int -> Int)  ?
```

λ-Abstraction

Lambda notation makes it explicit that a value can be a function. Thus,

`(plus 1)` can be written as `\y -> (1 + y)`

(In Haskell `\x` is a syntactic approximation of λ**x**)

```
plus x  y =  x + y
```

can be written as

```
plus = \x -> \y -> (x + y)
```

or as

```
plus = \x y -> (x + y)
```

Parentheses Convention

```
f e1 e2     ≡   ((f e1) e2)
f e1 e2 e3 ≡  (((f e1) e2) e3)
```

application is *left associative*

```
Int -> (Int -> Int) ≡ Int -> Int -> Int
```

type constructor "->" is *right associative*

Type of a Block

```
(let
   x1 =  e1
   .
   .
   .
   xn =  en
 in
    e )           ::    t
```

provided

```
         e        ::    t
```

Type of a Conditional

$$(\textit{if } e \textit{ then } e_1 \textit{ else } e_2) :: t$$

provided

$$e :: \text{Bool}$$
$$e_1 :: t$$
$$e_2 :: t$$

The type of expressions in both branches of conditional must be the same.

Polymorphism

```
twice f x = f (f x)
```

1. `twice (plus 3) 4`

   `→ (Plus 3) ((plus 3) 4)`

   `→ ((plus 3) 7)`

   `→  10`

   `twice ::` $\boxed{\texttt{(Int → Int) → Int → Int}}$ **?**

2. `twice (append “Zha") “Gabor"`

   `→“ZhaZhaGabor”`

   `twice ::` $\boxed{\texttt{(Str → Str) → Str → Str}}$ **?**

Deducing Types

`twice f x = f (f x)`

What is the most "general type" for twice?

1. Assign types to every subexpression

   `x :: t0` `f :: t1`

   `f x :: t2` `f (f x) :: t3`

   $\Rightarrow$ `twice :: t1 -> t0 -> t3` ?

2. Set up the constraints

   `t1 = t0 -> t2` because of `(f x)`

   `t1 = t2 -> t3` because of `f (f x)`

3. Resolve the constraints

   `t0 -> t2 = t2 -> t3`

   $\Rightarrow$ `t0 = t2 and t2 = t3` $\Rightarrow$ `t0 = t2 = t3`

   $\Rightarrow$ `twice :: (t0 -> t0) -> t0 -> t0`

Another Example: *Compose*

`compose f g x = f (g x)`

What is the type of **`compose`** ?

1. Assign types to every subexpression

```
   x :: t0      f :: t1      g :: t2
   g x :: t3    f (g x) :: t4
```

   ⇒ `compose :: t1 -> t2 -> t0 -> t4`

2. Set up the constraints

   `t1 = t3 -> t4` because of `f (g x)`

   `t2 = t0 -> t3` because of `(g x)`

3. Resolve the constraints

   ⇒ `compose :: (t3 -> t4) -> (t0 -> t3) -> t0 -> t4`

Now for some fun

```
        twice f x = f (f x)
    a = twice₁ (twice₂ succ) 4
    b = twice₃ twice₄ succ 4
```

1. **Is a=b ?**

   yes `succ (succ (succ (succ 4)`

2. **Are the types of all the twice instances the same?**

   no

```
twice₁ :: (I -> I) -> I -> I
twice₂ :: (I -> I) -> I -> I
twice₃ :: ((I -> I) -> I -> I) ->
                   (I -> I) -> I -> I
twice₄ :: (I -> I) -> I -> I
```

The first person with the right types gets a prize!

Hindley-Milner Type System

Haskell and most modern functional languages follow the Hindley-Milner type system.

The main source of polymorphism in this system is the *Let block*.

The type of a variable can be instantiated differently within its lexical scope.

much more on this later ...