LECTURE 03

Theory and Design of PL (CS 538)

January 31, 2020
MORE ON HW1
**UPDATES**

- Make sure to compile with `--Wall` before submitting
  - If there are warnings in starter code, please fix
- Make sure to run `hlint` before submitting
  - Don’t need to do all changes; use your judgment
SMALL CONTEST

- We will run all solutions on several new puzzles
- Fastest solutions get a small prize (not for grade)
- Details:
  - Run on instructional machines
  - One solution, and first N solutions

*Will grade solutions for given functions*
MORE HIGHER-ORDER
EXAMPLE: APPLICATION

- Apply a function to an argument, get result:

\[
($) :: (a \rightarrow b) \rightarrow a \rightarrow b
\]

`fun $ arg = f arg`

- Why use this? One use: avoiding parentheses

`plusOne :: Int \rightarrow Int`

`val = plusOne $ plusOne 42`

-- SAME AS: `val = plusOne (plusOne 42)`

-- BUT NOT: `val = plusOne plusOne 42`
EXAMPLE: COMPOSITION

• Chain two functions, get another function:

\[(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c\]
\[(.) \text{ sndFun} \text{ fstFun} x = \text{sndFun} (\text{fstFun} x)\]

-- NOTE: order matters!

• Example: repeat functions:

\[\text{doTwice} :: (a \rightarrow a) \rightarrow a \rightarrow a\]
\[\text{doTwice} \text{ fun} = \text{fun} \cdot \text{fun}\]

\[\text{plusTwo} = \text{doTwice} \text{ plusOne}\]
EXAMPLE: FLIP

- Swap arguments of a two-argument function. Type?

```
flip :: (a -> b -> c) -> b -> a -> c
--- SAME AS: (a -> b -> c) -> (b -> a -> c)
```

- How can we implement this function?

```
flip f y x = f x y
```
Example: Until

- Repeat fn from init until condition holds. Type?

\[
\text{until} :: (a \to \text{Bool}) \to (a \to a) \to a \to a
\]

- How can we implement this function?

\[
\text{until \ stop \ f \ cur}  \\
| \text{stop \ cur = cur}  \\
| \text{otherwise = until \ stop \ f \ (f \ cur)}
\]
EXAMPLE: CURRYING
MULTIPLE ARGUMENTS

- Given two integers, produce integer
- First possible type (*uncurried*):

```haskell
myBinaryFn :: (Int, Int) -> Int
foo = myBinaryFn (7, 42)
```
A BETTER TYPE

• Given one integer, produce function from int to int
• Second possible type (curried):

\[ my\text{BinaryFn}' :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

\[-\text{SAME AS: myBinaryFn}' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

\[-\text{BUT NOT: myBinaryFn}' :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \]

\[ \text{foo} = \text{myBinaryFn}' 7 42 \]
PARTIAL APPLICATION

• Don’t need to provide all arguments at once:

```haskell
plus :: Int -> Int -> Int
plus x y = x + y

plusOne :: Int -> Int
plusOne = plus 1          -- SAME AS: plusOne y = 1 + y
```

• Only works for *curried* functions, not uncurried

```haskell
plus' :: (Int, Int) -> Int
plus' (x, y) = x + y

plusOne' = plus' ???
```
CURRY/UNCURRY

• From uncurried to curried:

\[
\text{curry} :: ((a, b) \to c) \to (a \to b \to c)
\]
\[
\text{curry} \ f \ x \ y = f \ (x, y)
\]

• From curried to uncurried:

\[
\text{uncurry} :: (a \to b \to c) \to ((a, b) \to c)
\]
\[
\text{uncurry} \ f \ (x, y) = f \ x \ y
\]
WHAT IS A VALID PROGRAM?
A VALID PROGRAM...

- Doesn’t crash when you run it
- Applies functions to arguments of the right types
- Has properly nested parentheses (…), braces {...}
- ...

BASIC CRITERIA: SYNTAX

- Can check *statically*, without running program
- If syntax is wrong, program is definitely wrong
- If syntax is right, program could still be wrong
WORDS AND PHRASES

- Different kinds of words
  - Constants \((0, \text{true})\), operations \((+, -, \times)\)
  - Variable names \(x\)
  - Keywords \(\text{if, then, else, let, where}\)
- Compound words (phrases)
  - Expressions \(2\times x + 1\)
  - If-statements \(\text{if } b \text{ then 3 else 4}\)
HOW TO SPECIFY SYNTAX?
List of *production rules*: different kinds of phrases
- Terminals written " . . . " or ' . . . '
- Pipe | means or
- Each rule ended by semicolon
- Example:

```plaintext
digit-0-to-4 = "0" | "1" | "2" | "3" | "4" ;
digit-5-to-9 = "5" | "6" | "7" | "8" | "9" ;
digit        = digit-0-to-4 | digit-5-to-9 ;
```
REPEATING, OPTIONAL

- Braces for repetition, zero or more times:
  
  ```
  num = digit { digit }
  ```

- Brackets for option, zero or one times:
  
  ```
  signed-num = [ "-" ] num
  ```

- EBNF grammars, Extended Backus-Naur Form
BASIC EXAMPLES
Begin with Boolean constants:

```plaintext
bool-cons = "true" | "false" ; (* constants *)
```

Then add logical combinations:

```plaintext
bool-expr = bool-cons (* constants *)
| "!" bool-expr (* negation *)
| "(" bool-expr ")" (* paren term *)
| bool-expr "==" bool-expr (* equals *)
| bool-expr "&&" bool-expr (* and *)
| bool-expr "||" bool-expr ; (* or *)
```
NUMBERS

Integers and arithmetic operations

```plaintext
num-expr = signed-num               (* constants  *)
| "-" num-expr                 (* negate    *)
| "(" num-expr ")"              (* paren term *)
| num-expr "+" num-expr          (* add       *)
| num-expr "-" num-expr          (* minus     *)
| num-expr "*" num-expr ;        (* multiply  *)
```
EXAMPLE: LAMBDA
CALCULUS
WHY A CORE LANGUAGE?

- Simple enough to fully model
  - Remove all unnecessary features
  - Easier to study without extra noise
- Clarify key language similarities/differences
BRIEF HISTORY

- Universal model of computation
- Equivalent to Turing machines in power
- Common ancestor of *all* functional languages
STARTING POINT

Begin with variable names and constants:

\[
\begin{align*}
\text{var} & = "x" \mid "y" \mid "z" \mid \ldots ; \\
\text{expr} & = \text{var} \quad \text{(* variables *)} \\
& \quad \mid \text{bool-cons} \mid \text{num-cons} \quad \text{(* base const *)} \\
& \quad \mid "(" \text{expr}"\)" \quad \text{(* paren expr *)}
\end{align*}
\]
DEFINING FUNCTIONS

Functions have input variable, body expression

expr = var          (* variables *)
  | bool-cons | num-cons    (* base const *)
  | "(" expr ")"   (* paren expr *)
  | "λ" var "." expr ; (* functions *)
CALLING FUNCTIONS

expr = var                    (* variables   *)
    | bool-cons | num-cons   (* base const   *)
    | "(" expr ")"           (* paren expr  *)
    | "λ" var "." expr       (* functions   *)
    | expr " " expr        ; (* application *)

Call function with argument by separating with space
ADD PRIMITIVES AS NEEDED

Adding in some Boolean operations...

```plaintext
eexpr = ...
  | eexpr "==" eexpr
  | eexpr "&&" eexpr
  | eexpr "||" eexpr
  | "!" eexpr
;
```

...and some other operations

```plaintext
eexpr = ...
  | eexpr "+" eexpr
  | eexpr "*" eexpr
  | "-" eexpr
  | "if" eexpr "then" eexpr "else" eexpr
;```
CONCRETE VERSUS
ABSTRACT SYNTAX
TWO KINDS OF SYNTAXES

• Both can be described by grammars
• Concrete: string of characters
  ▪ Source code from a file
  ▪ Data sent over a network
• Abstract: tree with labeled nodes
CONCRETE IS GOOD, BUT...

- Keeps a lot of irrelevant details
  - Parentheses, spaces, ...
- Some important features are hard to see
  - Ambiguity: \(1 + 2 \times 3\) is \((1 + 2) \times 3\) or \(1 + (2 \times 3)\)?
  - Where is the scope of variables?
ABSTRACT SYNTAX TREES

- Represent program code as a labeled tree
- Each node has:
  - a label (an operation)
  - some number of child trees (maybe 0)
- Different representation of actual code

*Code is more than a just list of characters*
EXAMPLE
CONCRETE VS. ABSTRACT?

- Concrete: closer to what programmers write
  - Useful when parsing actual programs
- Abstract: closer to what a program means
  - Useful when representing code in compilers
  - Useful when performing optimizations
  - Useful when proving things about programs

We will mostly work with abstract syntax