Year of the Rabbit Notes

Sunday, October 22, 2023 12:01 PM

Declarative Languages
- Paradigm that describes what program does without specifying control flow
- Includes many functional languages like Haskell and logic-programming languages like Prolog

Imperative Languages
- Paradigm that specifies how program should do something by explicitly specifying each instruction
- Includes OOP languages like C++, C#, Java

Object Oriented Programming
- Part of imperative languages
- Uses main concept of imperative languages
- Contrasting procedural since procedural does not have inheritance & data hiding

Esoteric Languages
- Tests boundaries of computer programming language design
- Mostly troll
- Parody, difficult to read & write

Examples:
- Brainfuck: 8 symbols, minimalist
- JSFuck: Esoteric version of Javascript

Algol
- Family of imperative coding languages developed in 1958
- Introduced code blocks with pairs of begin and end
- First language with nested functions with lexical scope

example of differences:

1) $(\lambda x.\lambda y.\ x\ y)\ (\lambda y.y\ y)\ a$

CBV: $\rightarrow (\lambda x.\lambda y.x\ y)(a\ a)$
$\rightarrow (\lambda y.a\ y)a$
$\rightarrow a\ a$

CBN: $\rightarrow (\lambda y.(\lambda y.y\ y)\ y)a$
$\rightarrow (\lambda y.y\ y)a$
$\rightarrow a\ a$

Same results & same # of steps

2) $(\lambda x.x)\ (\lambda x.xx)(\lambda x.xx)$

CBV: $\rightarrow (\lambda x.x)(\lambda x.xx)(\lambda x.xx)$
$\rightarrow (\lambda x.x)(\lambda x.xx)(\lambda x.xx)$
$\not\rightarrow$

Does NOT terminate

CBN: $\rightarrow (\lambda x.x\ x)(\lambda x.xx)$
$\rightarrow$ OMEGA COMBINATOR

Does NOT terminate

Lambda (λ) Calculus
Methods of Solving

Full Beta Reduction
- Any order non-deterministic

Normal Order
- Left most outermost redex first
- Repeat until no more left

Call By Name
- Evaluate function calls without evaluating arguments
- Stop when outermost term is λ

Call By Value
- Evaluate arguments before function call
- Evaluate left argument, then sub, and repeat if more arguments can be subbed in

Call By Need
- Uses syntax graphs instead of syntax trees
- Similar to call by name, but prevents repeated calls
- Haskell uses this!

$\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$

No matter what, keeps repeating so $\therefore$ divergent

How to Read Rules
- Given the conditions on to, the bottom is the result
- e usually means something can be evaluated and v means something is at its final step

Call by Name	Call by Value
$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$	$\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2}$ $\quad$ $\frac{e_2 \rightarrow e_2'}{v_1 e_2 \rightarrow v_1 e_2'}$
$\frac{}{(\lambda x.e_1)e_2 \rightarrow e_1[e_2/x]}$	$\frac{}{(\lambda x.e_1)e_2 \rightarrow e_1[e_2/x]}$
$\frac{}{fst(e_1,e_2) \rightarrow e_1}$	$\frac{e_1 \rightarrow e_1'}{(e_1,e_2) \rightarrow (e_1',e_2)}$ $\quad$ $\frac{e_2 \rightarrow e_2'}{(v_1,e_2) \rightarrow (v_1,e_2')}$
$\frac{}{snd(e_1,e_2) \rightarrow e_2}$	$\frac{}{fst(v_1,v_2) \rightarrow v_1}$ $\quad$ $\frac{}{snd(v_1,v_2) \rightarrow v_2}$
$\frac{}{let\ x=e_1\ in e_2 \rightarrow e_2[e_1/x]}$	$\frac{e_1 \rightarrow e_1'}{let\ x=e_1\ in e_2 \rightarrow let\ x=e_1'\ in e_2}$ $\quad$ $\frac{}{let\ x=v_1\ in e_2 \rightarrow e_2[v_1/x]}$

and, if_then_else are same

3) $(\lambda x.\lambda y.\ x\ x\ y)((\lambda x.x)\ b)\ ((\lambda x.x)b)$

CBV: $\rightarrow (\lambda x.\lambda y.\ x\ x\ y)\ b\ ((\lambda x.x)\ b)$
$\rightarrow (\lambda x.\lambda y.x\ x\ y)\ b\ b$
$\rightarrow (\lambda y.\ b\ b\ y)b$
$\rightarrow b\ b\ b$
$\not\rightarrow$

$\therefore$ CBV longer

CBN: $\rightarrow (\lambda y.\ ((\lambda x.x)\ b)((\lambda x.x)\ b)\ y)((\lambda x.x)\ b)$
$\rightarrow ((\lambda x.x)\ b)((\lambda x.x)\ b)((\lambda x.x)b)$
$\rightarrow b((\lambda x.x)b)\ ((\lambda x.x)b)$
$\not\rightarrow$

4) $(\lambda x.\ (\lambda y.yy)\ x\ x)\ (\lambda y.y)\ x$

CBV: $\rightarrow (\lambda.\ (\lambda y.yy)\ x\ x)\ x$
$\rightarrow (\lambda y.y\ y)\ x\ x$
$\rightarrow x\ x\ x$
$\not\rightarrow$

$\therefore$ CBN longer

CBN $\rightarrow (\lambda y.yy)\ (\lambda y.y)(\lambda y.y)x$
$\rightarrow (\lambda y.y)(\lambda y.y)(\lambda y.y)x$
$\rightarrow (\lambda y.y)\ (\lambda y.y)x$
$\rightarrow (\lambda y.y)\ x$
$\rightarrow x$

Domain Specific Languages (DSL)
AST: Abstract Syntax Tree

Shallow
- Each AST has only 1 definition
- AST NOT reusable
- Easily extend by defining new functions
- Direct
- Example: Arithmetic evaluator would have functions such as add (x y), Subtract (x y)...

Deep
- ASTs reusable
- Can't extend language w/o excessive recompilation
- Simple
- Example: Arithmetic evaluator would only have 1 function that would have different definitions such as evalExpr(Add x y), evalExpr(Minus x y)...
- Static analysis (type checking)

Tagless
- Extensible Syntax
- Extensible Interpretations
- Lets you have deep-embedding's static

Example: analysis but easier to write

```
class RegExp repr where                    Sh is from shallow
  char :: Char -> repr
  conc :: repr -> repr -> repr
  star :: repr -> repr

instance RegExp Sh.RegExp where...

instance RegExp String where...

to extend, just add a new class and instances
class RegExp repr => PlusRe repr
  plus :: repr -> repr
instance PlusRE Sh.RegExp where...
instance PlusRE String where...
```

Purpose of strategies: Formalizes system for computer to evaluate since something like Full Beta has no strategy and random.

y combinator: $\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x))$

If subbing in f

$g\ f = (\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))\ f$
$= (\lambda x.f(x\ x))(\lambda x.f(x\ x))$
$= \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$

Repeats (goes back to original statement.

Types
- If term t has type T, then well typed

examples:
- true: R => R=Bool
- false: R => R=Bool

- Each term t has at most one type
- Safety = Progress + Preservation

Progress: Well-typed term is not stuck
Preservation: If well-typed term evaluates, then result is also well-typed

Using Progress:
- Either t is a value OR t' exists for $t \rightarrow t'$

Formatting for Rules
- 3 place relation with context, term, and type

context $\vdash$ term : type

Contexts
- Can be represented as Γ, where it represents sets of variable type relations (like tuple pairs where a variable A has type T)

If looking at a variable, must perform following:
1) Check context
2) If in context, then well-typed. Otherwise, false (not well-typed)

Polymorphism
- Types are obtrusive $\rightarrow$ Type Inference
- Inhibits code re-use $\rightarrow$ Polymorphism

Ad Hoc
Overloading:
- Resolved at compile-time
- Overridden methods at run-time
- One name for different functions

int $\rightarrow$ int 1+2
real $\rightarrow$ real 1.0+2.0

Coercion:
- Compile away subtyping by run-time coercions

(real 1) + 2.0 or 1.0 + 2.0

Universal Polymorphism
Inclusion:
- Subtype polymorphism
- One object belongs to many classes

Parametric:
- Uses type variables

$f = \lambda x{:}int \rightarrow int.\ \lambda y{:}int.\ x(x(y))$
bool $\rightarrow$ bool bool
A $\rightarrow$ A A
principle type of $f = \lambda x.\lambda y.\ x(x(y))$

Parametric Polymorphism
How do I find principle type?
e.g. $\lambda x.\lambda y.\ x(x(y))$

Type Check & Accumulate

$X = Y \rightarrow Z$ for $x(y)$

$X = Z \rightarrow W$ for $x(x(y))$

$Z = Y$ and $X = Y \rightarrow Y$ (smallest solution)

Subtyping
- Suppose usual typing system:

$$\frac{\Gamma \vdash x : A \rightarrow B \quad \Gamma \vdash y : A}{\Gamma \vdash xy : B}$$

the term
$(\lambda r{:}\{x{:}Nat\}.r.x)\ \{x=0, y=1\}$
NOT WELL-TYPED

- Some types better in general
- Formalize where subtyping can be used
- Where $S <: T$ means S is a subset of type T

T-Sub: $\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$

- S is better than T
- S is a subset of T
- S is more informative/richer than T

Using the original example,

$(\lambda r{:}\{x{:}Nat\}.\ r.x)\ \{x=0, y=1\}$

where $\{x{:}Nat, y{:}Nat\} <: \{x{:}Nat\}$

(Left side is larger than right)

by subsumption, $\vdash \{x=0, y=1\} : \{x{:}Nat\}$

and as a result, the original statement is well-typed

$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$ S-Arrow

$S <: Top$ S-Top (Top is maximum element)

$S <: S$ S-Refl

$\frac{S <: U \quad U <: T}{S <: T}$ S-Trans

Covariance: Allows assigning an instance to a variable whose type is one of the instance's generic type. (supertype)

Contravariance: Allows assigning an instance to a variable whose type is one of the instance's derived type. (subtype)

S-type is an example

Type Substitution
$\sigma = [X/bool, Y/X \rightarrow X]$

$\sigma X = bool$ and $\sigma Y = X \rightarrow X$

$(\sigma \circ \gamma)S = \sigma(\gamma S)$

$\sigma \circ \gamma := [X/\sigma(T)$ for X/T in γ and
X/T for X/T in σ with $X \notin dom(\gamma)]$

Substitution Preserves Typing!
if $\Gamma \vdash t : T$, $\sigma\Gamma \vdash \sigma t : \sigma T$

example: Where $x{:}X \vdash \lambda y{:}X \rightarrow int.\ y\ x : int$ derivable

if $\sigma = [X/bool]$ (means $\sigma X = bool$)

$x{:}bool \vdash y{:}bool \rightarrow int.\ y\ x : int$ derivable

Γ : environment/context Solution Pair: (σ, T) given $(\Gamma; t)$
t: term

if $F = f{:}X, a{:}Y$ and $t = fa$

Then $([X/Y \rightarrow int], int)$
$([X/int \rightarrow int, Y/int], int)$
$([X/Y \rightarrow Z], Z)$
and more

ALL VALID

Unification
- Syntactic equational unification
- Defines sets of terms where $t := x \mid f(t_1, \ldots, t_n)$ $x \in Var$ $f \in$ func symbols
- Given equation $s \approx t$, look for substition such that $\sigma s \approx \sigma t$
- σ_1 more general iff $\exists \sigma$ such that $\sigma\sigma_1 = \sigma_2$ $\sigma_1 \leq \sigma_2$
- Principle unifier σ_1 where $\forall$ unifiers σ $\sigma_1 \leq \sigma$

e.g. $f(x,y) \approx f(a,y)$

$\sigma_1 = [x/a, y/b]$

$\sigma_2 = [x/a]$ (principle unifier)

Practice Exam questions

$a \rightarrow int$ & $bool \rightarrow b$			$S = (a/bool, b/int)$
$a = bool$	$int = b$		
$a \rightarrow bool$	$b = int$		$bool \rightarrow int$
	$b \rightarrow int$		

$a \rightarrow a$ & $bool \rightarrow b$ $S = (a/b)$
$a = b$ $a \rightarrow b$
$b \rightarrow b$ $b = b$ $b \rightarrow b$

a & $a \rightarrow a$
$a = a \rightarrow a$ Keeps continuing
$a \rightarrow a \rightarrow a$ (infinite)

$a \rightarrow (a \rightarrow b)$ & $(c \rightarrow d) \rightarrow (b \rightarrow b)$

$a = c \rightarrow d$
$(a \rightarrow b) = (b \rightarrow b)$ $S = (a/(c \rightarrow d), b/(c \rightarrow d))$
$(c \rightarrow d) \rightarrow b = b \rightarrow b$ $(c \rightarrow d) \rightarrow ((c \rightarrow a) \rightarrow (c \rightarrow d))$
$(c \rightarrow d) = b \Rightarrow b = (c \rightarrow d)$
$b = b$ trivial

$a \rightarrow a$ & $(b \rightarrow c) \rightarrow (d \rightarrow e)$ & $(d \rightarrow c) \rightarrow a$
$a = b \rightarrow c$
$a = d \rightarrow e$
$b \rightarrow c = d \rightarrow e$ $b = d$ $c = e$
$(d \rightarrow c) \rightarrow a \Rightarrow (d \rightarrow c) \rightarrow (b \rightarrow c)$
$d \rightarrow c) \rightarrow (d \rightarrow c)$ $(d = d, c = c$ trivial

$S = (a/(b \rightarrow c), b/d, c/e)$
$(d \rightarrow e) \rightarrow (d \rightarrow e)$

Prolog & Haskell
- Prolog is relationships and Haskell is functions
- Prolog only specifies what is true whereas Haskell needs to specify both

Prolog	Haskell
leaf(_). branch(_,_). is_tree(leaf(X)) :- X in inf...sup. is_tree(branch(X,Y)) :- is_tree(X), is_tree(Y).	data BT = Leaf Int | Branch BT BT
• Must use inf...sup for integers • Can pull variables out of nowhere with relationships as: p(A,B) :- p(A,A1), where A1 & B1 are pulled from relationship p(B,B1), B := DO SOMETHING USING A1 & B1.	
• Must create custom types (relationships) for Maybe Type of Haskell just(_). nothing(). isJust(just(_)). isNothing(nothing()).	data Maybe = Just a | Nothing isJust(Just _) = True isJust _ = False isNothing(Nothing) = True isNothing _ = False

BNF (Backus-Naur Form) & EBNF (Extended Backus-Naur Form)
- Metasyntax notation for context free languages
- BNF does not use " " and surrounds all values with ' ' and NO {} compared to EBNF
- describes syntax, NOT semantics
  e.g. Writing integers

EBNF
<Integer> ::= [-] <digit> {<digit>}
<digit> ::= 0|1|2|3|4|5|6|7|8|9

BNF
<Integer> ::= -<abs> | <abs>
<abs> ::= <digit>|<digit> <abs>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

BNF Evaluation of -690
<Integer>
=> -<abs>
=> -<digit><abs>
=> -6<abs>
=> -6<digit><abs>
=> -69<abs>
=> -69<digit>
=> -690

<>: Specifies rules. <Integer> is top-level & <digit> is sub-rule
1: Set of options (or)
[]: Optional (ONLY in EBNF)
{}: Set of 0 or more repetitions (ONLY in EBNF)

Barendreght's Variable Convention
1) Always keep bound variable's name different from names of other variables at particular time
2) Keep bound variable names different from name of other bound variables names and all free variables.

De Bruijn Index
- Invented without naming bound variables
- Terms written using these indices are invariant to a-conversion, so checking a-equivalence is the same as syntactic equality

e.g. $\lambda x.\lambda y.\ x$ (K combinator) becomes

$\lambda\lambda 2$, where the lambda is 2 away

Church Encoding
- Uses lambda to represent functions

Example:
bools: tru = $\lambda t.\lambda f.t$
fls = $\lambda t.\lambda f.f$

Where tru & fls takes two inputs, return the first one if tru & second if fls

if, then, else: ife = $\lambda c.\lambda then.\lambda else.$ c then else

c: condition then: then result else: else result

ife tru u v
$\rightarrow (\lambda c.\lambda then.\lambda else.$ c then else$)$ tru u v
$\rightarrow (\lambda then.\lambda else.$ tru then else$)$ u v
$\rightarrow (\lambda else.$ tru u else$)$ v
$\rightarrow$ tru u v
$\rightarrow (\lambda t.\lambda f.t)$ u v
$\rightarrow (\lambda f.u)$ v
$\rightarrow$ u
$\not\rightarrow$

and $= \lambda b.\lambda c.$ b c fls
pair $= \lambda f.\lambda s.\lambda b$ b f s
fst $= \lambda p.$ p tru
snd $= \lambda p.$ p fls

$c_0 = \lambda s.\lambda z.\ z$ (0)
$c_1 = \lambda s.\lambda z.\ s\ z$ (1)
$c_2 = \lambda s.\lambda z.\ s(s\ z)$ (2)

left $= \lambda a.\lambda l.\lambda r.$ la
right $= \lambda b.\lambda l.\lambda r.$ rb

Continues for all integers

succ $= \lambda n.\lambda s.\lambda z.\ s(n\ s\ z)$
succ c_2
$\rightarrow (\lambda n.\lambda s.\lambda z.\ s(n\ s\ z))\ c_2$
$\rightarrow \lambda s.\lambda z\ s(c_2\ s\ z)$
$\rightarrow \lambda s.\lambda z\ s((\lambda s.\lambda z.\ s(s\ z))\ s\ z)$
$\rightarrow \lambda s.\lambda z\ s((\lambda z.\ s(s\ z))\ s\ z)$
$\rightarrow \lambda s.\lambda z\ s(s(s\ z))$
$\rightarrow c_3$

mul $= \lambda m.\lambda n.$ m (plus n) c_0

plus $= \lambda m.\lambda n.\lambda s.\lambda z.$ m s (n s z)
plus $c_3\ c_3$
$\rightarrow (\lambda m.\lambda n.\lambda s.\lambda z.$ m s (n s z)$)\ c_3\ c_3$
$\rightarrow (\lambda n.\lambda s.\lambda z.\ c_3\ s\ (n\ s\ z))\ c_3$
$\rightarrow \lambda s.\lambda z.\ c_3\ s\ (c_3\ s\ z)$
$\rightarrow \lambda s.\lambda z.\ (\lambda s.\lambda z.\ s(s(s\ z)))\ s\ ((\lambda s.\lambda z.\ s(s(s\ z)))\ s\ z)$
$\rightarrow \lambda s.\lambda z.\ (\lambda z.\ s(s(s\ z)))\ ((\lambda s.\lambda z.\ s(s(s\ z)))\ s\ z)$
$\rightarrow \lambda s.\lambda z.\ (s(s(s((\lambda s.\lambda z.\ s(s(s\ z)))\ s\ z))))$
$\rightarrow \lambda s.\lambda z.\ (s(s(s((\lambda z.\ s(s(s\ z)))\ z))))$
$\rightarrow \lambda s.\lambda z.\ (s(s(s(s(s(s\ z))))))$
$\rightarrow c_6$
$\not\rightarrow$

Trees:
Leaf $= \lambda eb.e$
Branch x l r $= \lambda eb.$ bx (le b) (re b)

List:
List a $= \lambda nil.\lambda cons.\lambda a.$ nil a $(\lambda head.\lambda tail.$ cons head (tail a))

Type Inference
- Work forwards then backwards
- Find constraints and types, and then determine types
- Start by introducing fresh types, then check for constraints

If-Statement type
env $\vdash$ if e1 then e2 else e3
: 't $\dashv$ C1, C2, C3, C
if fresh 't
and env $\vdash$ e1 : t1 $\dashv$ C1
and env $\vdash$ e2 : t2 $\dashv$ C2
and env $\vdash$ e3 : t3 $\dashv$ C3
and C = {t1 = bool, 't = t2, 't = t3}

e.g. $\{\} \vdash$ if true then true else false : 'a $\dashv$ {'a = bool}
$\{\} \vdash$ true : bool $\dashv \{\}$
$\{\} \vdash$ true : bool $\dashv \{\}$
$\{\} \vdash$ false : bool $\dashv \{\}$
C = {bool = bool, a' = bool, b' = bool}

Function type (anonymous)
env $\vdash$ fun x $\rightarrow$ e : t1' $\rightarrow$ t2 $\dashv$ C
if fresh t1'
and env, x : t1' $\vdash$ e : t2 $\dashv$ C

e.g. $\{\} \vdash$ fun x $\rightarrow$ and true x : bool $\rightarrow$ bool $\dashv$ {'a = bool}
{x : 'a} $\vdash$ and true x : bool $\dashv$ {'a = bool}
{x : 'a} $\vdash$ true : bool $\dashv \{\}$
{x : 'a} $\vdash$ x : 'a $\dashv \{\}$
C = {bool = bool, 'a = bool}

Other Inferences: