Intro | Dowen pg. 1

Fantastic PL
1. syntax, scope (binding)
2. static & dynamic semantics
3. type safety
4. λ calculus

syntactic equality : are two trees identical
→ Life ops "value"

$x, y, z \in Var ::= \ldots$ (eg, sum, min)

$n \in Num ::= 0 \mid 1 \mid 2 \mid \ldots$

$b \in Bool ::= True \mid False$

meta var

$e \in Expr ::= x \mid Num[n] \mid Bool[b]$
$\mid Plus(e_1; e_2) \mid Less(e_1; e_2)$
$\mid If(e_1; e_2; e_3) \mid Let(e_1, x.e_2)$

Scope

"let $x = e_1$ in e_2"

nonterminal Var
metavar Expr
Let(e_1, x, e_2)
gets value of x
"substitution" bind x to e_1

what is static scope?
1. names for bound vars (bound) don't matter
2. Substitution doesn't capture free variables.

E: let x=5 in let y=x in let x=10 in y
Let(Let(Let(y; 10, x); x, y); 5, x)

let x=5 in let x=10 in x
let x=5 in 10
10

in static scope, look at nearest binder to find value

R: BNF (data construct) is like induction, ~~all is easier~~

N: $e'[e/x]$ only if x is free
"I want e' but every time I see x I replace it w e"

BC (conditions)
$x[e/x] = e$
$y[e/x] = y \quad (x \neq y)$
$Num[n][e/x] = Num[n]$
$Bool[b][e/x] = Bool[b]$
$Plus(e_1; e_2)[e/x] = Plus(e_1[e/x]; e_2[e/x])$
$Less(e_1; e_2)[e/x] = Less(e_1[e/x]; e_2[e/x])$
$If(e_1; e_2; e_3)[e/x] = If(e_1[e/x]; e_2[e/x]; e_3[e/x])$
$Let(e_1; x.e_2)[e/x] = Let(e_1[e/x]; x, e_2)$
$Let(e_1; y.e_2)[e/x] = Let(e_1[e/x]; y.e_2[e/x])$ $(x \neq y \ \& \ y \notin FV(e))$

So we need one more case
(this is called "capture avoidance")

C: $(let\ x=1\ in\ x)[5/x]$
$\Rightarrow 1[5/x]$
$\Rightarrow 1$

Intro | Duncan

this definition allows capture avoidance

Bound Variables inductively defined

BC

$BV(x) = \{\}$

$BV(Num(n)) = \{\}$

$BV(Bool(n)) = \{\}$

$BV(Plus(e_1; e_2)) = BV(e_1) \cup BV(e_2)$

Same for less, if

$BV(Let(e_1, x, e_2)) = BV(e_1) \cup BV(e_2) \cup \{x\}$

Free Variables inductively defined

BC

$FV(x) = \{x\}$

$FV(Num[n]) = \{\}$

$FV(Bool[b]) = \{\}$

$FV(Plus(e_1; e_2)) = FV(e_1) \cup FV(e_2)$

Same for less if

$FV(Let(e_1; x; e_2)) = FV(e_1) \cup (FV(e_2) \setminus \{x\})$

α-Equivalence $e_1 \sim_\alpha e_2$ or $e_1 =_\alpha e_2$

$\sim_\alpha$ is renaming

$let\ x = 1\ in\ x \sim_\alpha let\ y = 1\ in\ y$

$let\ x = 1\ in\ x \not\sim_t let\ y = 1\ in\ 2$ (where $\sim_t$ is free equivalence, i.e. $1+1 \not\sim_t 2$)

$let\ x = e_1\ in\ e_2 \sim_\alpha let\ y = e_1\ in\ e_2[y/x]$ (where $y \notin FV(e_2)$)

Properties of substitution

① for all e, e'; if $BV(e') \cap FV(e) = \{\}$ then $e'[e/x]$ is defined

② for all e, e'; if $x \notin FV(e')$ then $e'[e/x] = e'$.

Properties of α-equivalence

① for all e, e' there exists an e'' s.t. $e \sim_\alpha e''$ and $e''[e'/x]$ is defined

② if $e \sim_\alpha e'$ then $e[e''/x] \sim_\alpha e'[e''/x]$

HW

① add more operations like minus & equal

② prove properties of substitution and renaming inductively

S

① for all e and sets of vars x, there is an expression e' s.t. $e \sim_\alpha e'$ and $BV(e') \cap x = \{\}$ for all e and sets of var,

Intro 2 Hoffman p4

programs are mathematical objects

define a PL

① static semantics := what are valid programs?

② dynamic semantics := how to run programs

lang $\mathcal{E}$: (this is abstract syntax, concrete)

	abstract	
expr ::=	x	
	Num[n]	n
	bool[False]	false
	bool[True]	true
	if(e_1, e_2, e_3)	if
	let(e_1, x, e_2)	
	plus(e_1, e_2)	
	leq(e_1, e_2)	

① statics

option: all expressions are programs

this is not ideal. we should exclude things like 5 + true

observe that numbers and booleans are different types

(1+2)+8 is a valid expr

wh? (1+2) is a valid expr

8 is a valid expr

$\vdash (1+2)+8 : Num$

in general, $\Gamma \vdash e : \tau$ is a judgment

Intermission 1

$\mathcal{E}$ Inductive definition

① emp is a tree

② if n : num, t_1 : tree, t_2 : tree then node(n, t_1, t_2) is a tree

judgments are i.e. t tree

Inference rules are for defining judgments inductively

$$\frac{J_1 \quad \dots \quad J_n}{J}$$ ← premises, ← conclusion

① $\dfrac{}{\text{emp tree}}$

② $\dfrac{n \text{ num} \quad t_1 \text{ tree} \quad t_2 \text{ tree}}{\text{node}(n, t_1, t_2) \text{ tree}}$

$\mathcal{D}$ derivations ~~(turnstile)~~

$\mathcal{E}$ TS node(S(Z), emp, emp) tree

put it on bottom

$\Gamma \vdash e : \tau$

context maps vars to types

so it follows the predicate fncl'ism

assumptions, hypotheses in a proof.

"proof" (derivation)

$$\dfrac{\dfrac{\dfrac{}{Z \text{ num}}\,②}{S(Z) \text{ num}}\,③ \quad \dfrac{}{\text{emp tree}}\,① \quad \dfrac{}{\text{emp tree}}\,①}{\text{node}(S(Z), \text{emp}, \text{emp}) \text{ tree}}\,②$$

Intro 2 Hoffman

Type Rules for E

$$\frac{}{\Gamma \vdash num[n] : num} \qquad \frac{}{\Gamma \vdash bool[b] : bool} \qquad \frac{}{\Gamma, x:\tau \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x:\tau_1 \vdash e_2 : \tau}{\Gamma \vdash let(e_1, x.e_2) : \tau}$$

where $x \notin \Gamma$ by α

$$\frac{\Gamma \vdash e_1 : num \quad \Gamma \vdash e_2 : num}{\Gamma \vdash plus(e_1, e_2) : num} \qquad \frac{\Gamma \vdash e_1 : num \quad \Gamma \vdash e_2 : num}{\Gamma \vdash leq(e_1, e_2) : bool}$$

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash if(e, e_1, e_2) : \tau}$$

$\cdot \vdash let\ x=5\ in\ x \leq 6 : bool$

empty context

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 5 : num}(num) \quad \dfrac{}{x:num \vdash x : num} \quad \dfrac{}{x:num \vdash 6 : num}}{x:num \vdash x \leq 6 : bool}(leq)}{\cdot \vdash let(x \leq 6, 5.x) : bool}(let)}{\cdot \vdash let\ x=5\ in\ x \leq 6 : bool}$$

Lemma 1: For every expr e and every context Γ there is at most one type τ s.t. $\Gamma \vdash e : \tau$

(prove by rule induction)

Intermission 2

nat induction show P(n) by ① show P(0) (BC) ② show if P(n) then P(S(n))

because nat ::= Z | S(Z)

or $\frac{}{Z\ nat}$ ① $\frac{n\ nat}{S(n)\ nat}$ ②

Rule induction show P(a), show for every rule $\frac{a_1 \mathcal{J} \cdots a_n \mathcal{J}}{a\ \mathcal{J}}$ that

if P(a_1) and ... and P(a_n) then P(a)

Lemma 1 shows uniqueness from the mapp predicate, so show if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$ then $\tau_1 = \tau_2$

Inversion is another property we need to show...

Case "var rule" then $e = x$ and $\Gamma = \Gamma', x : \tau_1$

Lemma 2 if $\Gamma \vdash e_1 + e_2 : \tau$ then $\tau = num$ and $\Gamma \vdash e_1 : num$ and $\Gamma \vdash e_2 : num$

Lemma 2 Substitution

if $\Gamma, x:\tau \vdash e':\tau'$ and $\Gamma \vdash e:\tau$

then $\Gamma \vdash [e/x]e':\tau'$

Lemma 3 Weakening

If $\Gamma \vdash e:\tau$ and $x \notin \Gamma$ then $\Gamma, x:\tau' \vdash e:\tau$

Dynamic Semantics

Different kinds

operational : How to run a program

axiomatic : What can you prove about a program?

denotational : Describe programs as mathematical functions

Structural Operational semantics (small-step semantics)

Structural dynamics is a transition-system (like an automata has states and transitions)

4 judgments

s state	s init
s final	$s \mapsto s'$ "s can step to s'"

Iterated transition $\dfrac{}{s \mapsto^* s'}$ $\qquad \dfrac{s \mapsto s' \quad s' \mapsto^* s''}{s \mapsto^* s''}$

states are ~~expressions~~, well-typed and closed ⟵ $\triangleq$ closed := expr e is closed and well-typed if $\cdot \vdash e:\tau$ for some τ

all states are initial

values are final $\qquad \dfrac{}{\text{num}[n]\ \text{val}}$ $\qquad \dfrac{}{\text{bool}[b]\ \text{val}}$

Transitions $\qquad \dfrac{n = n_1 + n_2}{\text{plus}(\text{num}[n_1], \text{num}[n_2]) \mapsto \text{num}[n]}$ (plus1)

$$\frac{e_1 \mapsto e_1'}{\text{plus}(e_1, e_2) \mapsto \text{plus}(e_1', e_2)}\ (\text{plus2})$$

$$\frac{e_2 \mapsto e_2'}{\text{plus}(\text{num}[n_1], e_2) \mapsto \text{plus}(\text{num}[n_1], e_2')}\ (\text{plus3})$$

Intro 3 Haskell

Pg 2

Everything is perfectly sensible and perfectly decidable steps, which is why it's called inductive

$$\frac{e_1 \mapsto e_1'}{\text{let}(e_1, x, e_2) \mapsto \text{let}(e_1', x, e_2)} \qquad \frac{}{\text{let}(e_1, x, e_2) \mapsto e_2[e_1/x]}$$ call by name

$$\frac{e_1 \text{ val}}{\text{let}(e_1, x, e_2) \mapsto e_2[e_1/x]} \qquad \frac{e_1 \mapsto e_1'}{\text{let}(e_1, x, e_2) \mapsto \text{let}(e_1', x, e_2)}$$ call by val

$$\frac{e \mapsto e'}{\text{if}(e, e_1, e_2) \mapsto \text{if}(e', e_1, e_2)} \qquad \frac{}{\text{if}(\text{bool}[\text{T}], e_1, e_2) \mapsto e_1} \qquad \frac{}{\text{if}(\text{bool}[\text{F}], e_1, e_2) \mapsto e_2}$$

E

Call by val

let $x = 8+2$ in $(x+x)+2$

$\mapsto$ let $x = 10$ in $(x+x)+2$

$\mapsto (10+10)+2$

$\mapsto 20+2$

$\mapsto 22$

notation

Call by name

let $x = 8+2$ in $(x+x)+2$

$\mapsto ((8+2)+(8+2))+2$

$\mapsto^* 22$

$\mapsto 22$

let (plus(num[5]; num[2]); x, plus(plus(x;x); num[3]))

$\mapsto$ let (num[10]; x, plus(plus(x;x); num[3]))

$\mapsto$ plus(plus(num[10]; num[10]); num[3])

$\mapsto^2 22$

C Lemma

There is no expr e s.t. e val and $e \mapsto e'$ for some e'

L Lemma for e expr if $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 \alpha e_2$

Type Safety you don't get stuck in the dynamics

T, Ta ① (progress) if $\cdot \vdash e : \tau$ then either e val or there exists e' s.t. $e \mapsto e'$

② (preservation) if $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $e' : \tau$

isEven 5 : Bool

Q X step/transition relation doesn't have functions, if preserves types (no) isEven $\mapsto$ fals

Pf (progress)

Rule (plus) : then $e = plus(e_1, e_2)$, $\tau = num$,
$e_1 : num$ and $e_2 : num$

IH either e_1 val or there exists e_1' s.t. $e_1 \mapsto e_1'$
either e_2 val or there exists e_2' s.t. $e_2 \mapsto e_2'$

Case e_1 val and e_2 val then by canonical forms $e_1 = num[n_1]$ and $e_2 = num[n_2]$ for some n_1, n_2

But then $e \mapsto num[n_1 + n_2]$

Case e_1 val and $e_2 \mapsto e_2'$ then $e_1 = num[n_1]$ and $e \mapsto plus(num[n_1], e_2')$ (for some n_1, n_2

24

λ-calculus Untyped

$e ::= x \mid e_1 e_2 \mid \lambda x.e$

$e ::= x \mid app(e_1; e_2) \mid lam(x.e)$

R condition: $x \notin FV(e)$

dynamic scope convention

laws

(α) $\lambda x.e =_\alpha \lambda y.e[y/x]$ (where $y \notin FV(e)$)

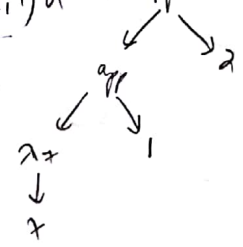
(β) $(\lambda x.e)\, e' =_\beta e[e'/x]$

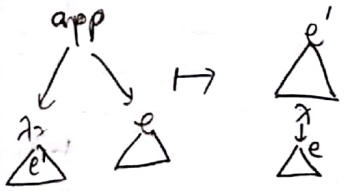
(η) $\lambda x.(e\, x) =_\eta e$ (where $x \notin FV(e)$)

operational semantics

$(\lambda x.e)\, e' \mapsto e[e'/x]$

$$\frac{(\lambda x.\lambda y.x)\,1 \mapsto \lambda y.1}{((\lambda x.\lambda y.x)\,1)\,2 \mapsto (\lambda y.1)\,2 \mapsto 1}$$





call-by-name

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2}$$

$E \in EvalCxt ::= \square \mid E\, e$

$$\frac{e \mapsto e'}{E\,e \mapsto E\,e'}$$

$\square[e] = e$

$(E\, e')[e] \quad (E[e])\, e'$

R call by name just means you can push frees through application or transition steps, instead of

call by Val

$(\lambda x.e)V \mapsto e[V/x]$

not trying to call any Var x or Value if you don't have free variables.

$V \in Value ::= x \mid \lambda x.e$

$E \in EvalCxt ::= \square \mid Ee \mid VE$

$\square[e] = e$

$(Ee')e = (E[e])\,e'$

$(VE)[e] = V(E[e])$

operational call by val as inference rules

$$\frac{}{x \text{ val}} \qquad \frac{}{\lambda x.e \text{ val}} \qquad \frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1 e_2 \mapsto e_1 e_2'}$$

$$\frac{e' \text{ val}}{(\lambda x.e)e' \mapsto e[e'/x]}$$

---

the encoding game

do { if e then e_1 else e_2 / True / False } in λ-calc

$(e\ e_1)\ e_2$ is if e then e_1 else e_2

$\lambda x.\lambda y.\ x$ is True

$\lambda x.\lambda y.\ y$ is False

HW

1. Encode $\mathbb{N}$ in lambda calc
2. Encode Set
   - $e \in e' = e'\ e$ (membership is function application)
   - $e_1 \cup e_2 = \lambda x.(\text{or}\ (e_1 x)\ (e_2 x))$
   - $e_1 \cap e_2 = \lambda x.(\text{and}\ (e_1 x)\ (e_2 x))$
3. Russell's paradox

$R = \{e \mid e \notin e\} = \lambda x.\ \text{not}\ (x\,x)$

$R \in R \mapsto (\text{not}\ (x\,x))[R/x]$

$= \text{not}\ (R\,R)$

$\Rightarrow \text{not}(\text{not}(R\,R))$

$\mathcal{R} = (\lambda x.xx)(\lambda x.xx) \quad (\mathcal{R} \mapsto \mathcal{R})$

$Y\,f = (\lambda x.\ f(xx))(\lambda x.\ f(xx))$

$\mapsto f(Yf)$

$times = \lambda x, \lambda y.$ if $x = 0$ then 0 else $y + (times\ (x-1)\ y)$ → the self reference sabotages you!

$timesish = \lambda next. \lambda x, \lambda y.$ if $x = 0$ then 0 else $y + (next\ (x-1)\ 0)$

the fixpoint of $timesish$: $times^* = Y\ timesish$

$\tau \in type ::= \tau_1 \to \tau_2 \mid \alpha$ (no ghilte types, so we give a placeholder type α, like const)

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \qquad \frac{\Gamma \vdash e:\tau' \to \tau \quad \Gamma \vdash e' \to \tau'}{\Gamma \vdash e\ e':\tau}(\to E) \qquad \frac{\Gamma, x:\tau \vdash e:\tau'}{\Gamma \vdash \lambda x.e:\tau \to \tau'}(\to I)$$

identity $\dfrac{\dfrac{}{x:\alpha \vdash x:\alpha}}{\cdot \vdash \lambda x.x:\alpha \to \alpha}(\to I)$

T = termination: if $\Gamma \vdash e:\tau$ then there is e' s.t. $e \mapsto^* e' \not\mapsto$

stronger types disallow / constrain

Intro 5 Hoffman pg1

Finite Data Structures

Base language:

	abstract syntax	concrete syntax
typ τ ::=	$\text{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$
exp e ::=	$\text{lam}\{\tau\}(x.e)$	$\lambda(x:\tau).e$
	$\text{app}(e_1;e_2)$	$e_1(e_2)$
	X	X

simply typed lambda calculus

Statics $\Gamma \vdash e : \tau$

$$\frac{\Gamma, x:\tau \vdash e:\tau'}{\Gamma \vdash \lambda(x:\tau)\, e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau}$$

(invariant, as evaluating w free vars, dynamics & steps rules omit contexts. No "$\vdash$" here)

Dynamics

Steps

$$\frac{}{\lambda(x)\, e \text{ val}} \qquad \frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)} \qquad \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')}$$

$$\frac{e_2 \text{ val}}{(\lambda(x:\tau).e)(e_2) \mapsto [e_2/x]e}$$

Products

- Conjunctive combination of data
- e.g. tuples, records, structs, unit

Pairs extend our types typ τ ::= ... exp e ::= ...

unit	unit	intro form	triv $\langle\rangle$
$\text{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$		$\text{pair}(e_1;e_2)$ $\langle e_1, e_2\rangle$

Statics for pair

$$\frac{}{\Gamma \vdash \langle\rangle : \text{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

elim form	$\text{pr}[l]\, e$	$e \cdot l$
	$\text{pr}[r]\, e$	$e \cdot r$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot l : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot r : \tau_2}$$

Intro 5 Hoffman p2

"eager pairs" "lazy"
F# vs.

Dynamics for products

$$\frac{}{\langle\rangle \ \text{val}} \qquad \frac{e_1 \ \text{val} \quad e_2 \ \text{val}}{\langle e_1 ; e_2 \rangle \ \text{val}}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \qquad \frac{e_1 \ \text{val} \quad e_2 \mapsto e_2'}{\langle e_1, e_2 \rangle \mapsto \langle e_1, e_2' \rangle} \qquad \frac{e \mapsto e'}{e \cdot l \mapsto e' \cdot l} \qquad \frac{e_1 \ \text{val} \quad e_2 \ \text{val}}{\langle e_1, e_2 \rangle \cdot l \mapsto e_1}$$

$$\frac{e \mapsto e'}{e \cdot r \mapsto e' \cdot r} \qquad \frac{e_1 \ \text{val} \quad e_2 \ \text{val}}{\langle e_1, e_2 \rangle \cdot r \mapsto e_2}$$

P products are very useful

set for free:

① multiple func. args like $(\tau_1 \times \tau_2) \to \tau_3$

② multiple return values like $\tau_1 \to \tau_2 \times \tau_3$ (took 10 years to get to Java)

E in python $(\tau_1 \times \tau_2) \to \tau_3 \neq (\tau_1 \times \tau_2) \to \tau_3$

Sums

- disjunctive combination of data
- e.g. enums, option type, void

E nullary & binary

typ τ ::= ...

void	void
sum(τ_1, τ_2)	$\tau_1 + \tau_2$

Exp e ::=

intro form

- $\text{in}[l]\{\tau_1; \tau_2\}(e)$ — $l \cdot e$ — "e is either τ_1 or τ_2"
- $\text{in}[r]\{\tau_1; \tau_2\}(e)$ — $r \cdot e$

elim form

- $\text{case}(e; x_1.e_1; x_2.e_2)$ — case e {$l \cdot x_1 \hookrightarrow e_1$ | $r \cdot x_2 \hookrightarrow e_2$}
- $\text{abort}\{\tau\}(e)$ — case e {}

Statics

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash in[l]\{\tau_1;\tau_2\}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash in[r]\{\tau_1;\tau_2\}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash case\,(e; x_1.e_1; x_2.e_2) : \tau}$$

$$\frac{\Gamma \vdash e : void}{\Gamma \vdash case\, e\{\} : \tau} \quad \text{or} \quad abort\{\tau\}(e)$$

Dynamics

$$\frac{e\ val}{l \cdot e\ val} \qquad \frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'}$$

$$\frac{e\ val}{r \cdot e\ val}$$

$$\frac{e \mapsto e'}{case\,(e; x_1.e_1; x_2.e_2) \mapsto case\,(e'; x_1.e_1; x_2.e_2)}$$

$$\frac{e\ val}{case\,(\ l \cdot e\ ;\ x_1.e_1;\ x_2.e_2\ ) \mapsto [e/x_1]\,e_1}$$

$$\frac{e\ val}{case\,(\ r \cdot e\ ;\ x_1.e_1;\ x_2.e_2\ ) \mapsto [e/x_2]\,e_2}$$

E

$bool = unit + unit$

$enum = \sum_{k \in I} unit$

option type =

$opt\ \tau := unit + \tau$

$null := l \cdot \langle\rangle$

$just\ e := r \cdot e$

$ifnull\ e\ \{ null \hookrightarrow e_1 \mid just(x_2) \hookrightarrow e_2 \} := case\,(e; x_1.e_1; x_2.e_2)$

varibles, funcets

$e ::= \ldots \mid Z \mid S(e)$ $\quad$ $e ::= \ldots \mid z \mid se$

$\mid rec(e; e_0 ; x y . e_1)$ $\quad$ $\mid re\ e\ as \{ Z \Rightarrow e_0 \mid Sx\ with\ y \Rightarrow e_1 \}$

a recursor, a loopity operation

body of recursor; some result

no in lazy e

$\begin{cases} add\ Z\ n = n \\ add\ (Sm)\ n = S\ (add\ m\ n) \end{cases}$

$add = \lambda m{:}Nat. \lambda n{:}Nat.$ $rec\ m\ as \{ 0 \Rightarrow n \mid Sm'\ w\ r \Rightarrow Sr \}$

not in lazy e

$\begin{cases} mult\ Z\ n = 0 \\ mult\ (Sm)\ n = n + mult\ m\ n \end{cases}$

$mult = \lambda m{:}Nat. \lambda n{:}Nat.\ rec\ m\ as$
$Z \Rightarrow Z$
$\mid Sm' \Rightarrow w\ r \Rightarrow add\ n\ r$

* avoid explicit recursion, have the PL do recursion for you.

$pred\ Z = Z$
$pred\ (Sn) = n$

$pred = \lambda n{:}Nat.\ rec\ n\ as$
$Z \Rightarrow Z$
$Sn'\ with\ r \Rightarrow n'$

$\tau ::= \ldots \mid Nat$

$$\frac{}{\Gamma \vdash Z : Nat} \qquad \frac{e : Nat}{\Gamma \vdash Se : Nat}$$

$$\frac{\Gamma \vdash e : Nat \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x{:}Nat, y{:}\tau \vdash e_1 : \tau}{\Gamma \vdash rec\ e\ as \{ Z \Rightarrow e_0 \mid Sx\ w\ y \Rightarrow e_1 \} : \tau}$$

a lazy semantics

lazy! no premise

$$\frac{}{Z\ val} \qquad \frac{}{Se\ val}$$

$$\frac{}{rec\ Z\ as \{ z \Rightarrow e_0 \mid Sx\ with\ y \Rightarrow e_1 \} \mapsto e_0}$$

$$\frac{}{rec\ Se\ as \{ Z \Rightarrow e_0 \mid Sx\ with\ y \Rightarrow e_1 \} \mapsto e_1 [e/x,\ rec\ e\ as\ Z \Rightarrow e_0\ Sx\ with\ y \Rightarrow e_1 / y]}$$

evaluation context $\rightarrow$ $E ::= \ldots \mid rec\ E\ as \{ \ldots \}$

for CBV $\rightarrow$ $(\mid SE)$

$$\frac{e \mapsto e'}{rec\ e\ as\ Z \Rightarrow e_0\ Sx\ with\ y \Rightarrow e_1 \ \mapsto\ rec\ e'\ as\ Z \Rightarrow e_0\ Sx\ with\ y \Rightarrow e_1}$$

Intro 6

Duneen

p12

| var $\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$ } equi-recursion

$\tau ::= \ldots \mid \mu\alpha.\tau \mid \alpha$ $\quad \mu\alpha.\tau \overset{fold}{\approx} \tau[\mu\alpha.\tau/\alpha]$ } iso recursion

(unfold) difference is that $\approx$ is weaker than $=$

these terms are closed

$e ::= \ldots \mid fold(e) \mid unfold(e)$ $\qquad \alpha \in FV(\tau)$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash fold(e)\{\alpha,\tau\}(e) : \mu\alpha.\tau} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash unfold(e)\{\alpha,\tau\} : \tau[\mu\alpha.\tau/\alpha]}$$

this info is provided by iso-, not in equi

CBN

$$\frac{}{unfold(fold(e)) \mapsto e} \qquad \frac{e \mapsto e'}{unfold(e) \mapsto unfold(e')} \qquad \frac{}{fold(e)\ val}$$

(β) engine

(η) extensionality (paper)

$fold(unfold(e)) \sim_\eta e : \mu\alpha.\tau$

CBV / eager

$$\frac{e\ val}{fold\ e\ val} \qquad \frac{e\ val}{unfold(fold(e)) \mapsto e} \qquad \frac{e \mapsto e'}{fold(e) \mapsto fold(e')}$$

E

$Nat \approx unit + Nat$

an isomorphism on types, you can check if a number is zero

$List\ \tau \approx unit + (\tau \times List\ \tau)$

$Nat = \mu_{\alpha.(unit+\alpha)}$

$zero = fold(l \cdot \langle\rangle)$

$succ = fold(r \cdot e)$

$List\ \tau = \mu_{\alpha.(unit + (\tau \times \alpha))}$

$nil = fold(l \cdot \langle\rangle)$

$cons\ e\ e' = fold(r\langle e \cdot e' \rangle)$

Intro 6
Duncan
pg. 3

$\omega = \lambda x.\ x\ x$
$\Omega = \omega\ \omega$
$\Omega \mapsto \Omega$

• simple types got rid of self application

$\omega : \left(\mu_{\alpha.(\alpha \to \tau)} \to \tau\right) \to (\mu_{\alpha.\alpha \to \tau}) \to \tau$

$\omega = \left(\lambda x : \mu_{\alpha.(\alpha \to \tau)} .\ (\text{unfold}\ x)\right)\ x$

$\Omega : \tau$
$\Omega = \omega\ (\text{fold}\ \omega)$ $\quad \mu_{\alpha.\alpha \to \tau}$

assoc

$Y : (\tau \to \tau) \to \tau$

$Y = \lambda f : \tau \to \tau .\ \left(\lambda x : \mu_{\alpha.(\alpha \to \tau)} .\ f\ (\text{unfold}\ x\ x)\right)$

$\text{fold}\left(\lambda x : \mu_{\alpha.(\alpha \to \tau)} .\ f\ (\text{unfold}\ x\ x)\right)$

$\Omega = \omega\ (\text{fold}\ \omega) \mapsto \left(\text{unfold}\ (\text{fold}\ \omega)\right)\ \text{fold}\ \omega \longmapsto \omega\ (\text{fold}\ \omega)$

Intro Hoffman p1

PCF & cost Semantics

PCF is a small language w general recursion

pcf := "programming computable functions" by Plotkin

Lang PCF

D: let $F: A \to A$ and $F(f) = f$ then we call f a fixed point of F

E: factorial

① operational view:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{if } n > 0 \end{cases}$$

② as an equation

$f = \lambda n.$ if $n=0$ then 1 else $n \cdot f(n-1)$

find a solution f to this equation

succ $= \lambda n.\, n+2$ is not a solution

fac $= \lambda n.\, n!$ is a solution

③ as a fixed point

$F(f) = \lambda n.$ if $n=0$ then 1 else $n \cdot f(n-1)$

we are looking for a fixed point of F. fac is the unique fixed point

Intensional on fixpoints

PCF :

	abstract syntax	concrete syntax
typ τ ::=	nat	nat
	$\text{part}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$

exp e ::=

- x
- z
- $s(e)$
- $\text{ifz}(e; e_0; x.e_1)$ ifz e {z ⇒ e_0 | s(x) ⇒ e_1}
- $\text{lam}\{\tau\}(x.e)$
- $\text{app}(e_1; e_2)$
- $\text{fix}\{\tau\}(x.e)$ fix $x:\tau$ is e

E: fac $\triangleq$ fix f:(nat→nat) is $\lambda(x:\text{nat})$ ifz x {z ⇒ s(z) | s(z) ⇒ x · f(z)}

statics

$$\frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x:\text{nat} \vdash \tau}{\Gamma \vdash \text{ifz}(e; e_0; x.e_1) : \tau} \qquad \frac{\Gamma, x:\tau \vdash e : \tau}{\Gamma \vdash \text{fix}\{\tau\}(x.e) : \tau}$$

dynamics

$$\frac{e \mapsto e'}{\text{ifz}(e; e_0; x.e_1) \mapsto \text{ifz}(e'; e_0; x.e_1)} \qquad \frac{}{\text{fix}\{\tau\}(x.e) \mapsto [\text{fix}\{\tau\}(x.e)/x]\, e}$$

$\lambda x.S(x)\ (\lambda\omega.S(\omega))$ ← problem in lazy setting

$\mapsto S\left(\lambda x.S(x)\ (\lambda\omega.S(\omega))\right)$

$\mapsto S\left(S\left(\lambda x.S(x)\ (\lambda\omega.S(\omega))\right)\right)$ ← but will go on forever in eager setting

$\mapsto^*$ is different from $\Downarrow$ because $\Downarrow$ is more like magic

$\mapsto^*$ has a known path except for this theorem

Structural operational semantics (dynamics) is better for type safety, than Evaluation dynamics

structural dynamics has a natural cost measure by taking length $(\mapsto^*)$

PCF

I

progress: if $e : \tau$ then either e val or $e \mapsto e'$

preservation: if $e : \tau$ and $e \mapsto e'$ then $e' : \tau$

Evaluation dynamics also called big-step operational semantics – another way of defining dynamics

$e \Downarrow v$ is a judgment saying expr e evaluates to value v

$$\frac{}{z \Downarrow^{0} z} \qquad \frac{e \Downarrow^{n} v}{s\,e \Downarrow^{n} s\,v} \qquad \frac{}{\lambda(x:\tau)\,e \Downarrow^{0} \lambda(x:\tau)\,e} \qquad \frac{e \Downarrow^{n_1} z \quad e_0 \Downarrow^{n_2} v_0}{\mathrm{ifz}(e; e_0; x.e_1) \Downarrow^{n_1+n_2} v_0}$$

$$\frac{e[\mathrm{fix}\,x:\tau\,\mathrm{is}\,e/x] \Downarrow^{n} v}{\mathrm{fix}\,x:\tau\,\mathrm{is}\,e \Downarrow^{n+1} v} \qquad \frac{e \Downarrow^{n_1} s(v) \quad e_1[v/x] \Downarrow^{n_2} v_1}{\mathrm{ifz}(e; e_0; x.e_1) \Downarrow^{n_1+n_2} v_1}$$

$$\frac{e[v_2/x] \Downarrow^{n_1} v \quad e_1 \Downarrow^{n_2} \lambda(x:\tau).e \quad e_2 \Downarrow^{n_3} v_2}{e_1\ e_2 \Downarrow^{n} v} \qquad n_1+n_2+n_3+1 = n$$

the judgment "$\square$ is a val"

$$\left\{ \begin{array}{l} \dfrac{}{z\ \mathrm{val}} \\[2ex] \dfrac{v\ \mathrm{val}}{s\,v\ \mathrm{val}} \end{array} \right. \qquad \frac{}{\lambda(x:\tau).e\ \mathrm{val}}$$

I

$$\frac{e \Downarrow v \text{ iff } e \mapsto^{*} v \text{ and } v\ \mathrm{val}}{e \Downarrow^{n} v \text{ iff } e \mapsto^{n} v \text{ and } v\ \mathrm{val}}$$

HW

now, annotate the 7 rules for $\Downarrow$ w counting numbers

polymorphism

$id\ Bool = \lambda(x{:}Bool).x \ : Bool \to Bool$

$id\ Num = \lambda(x{:}Num).x \ : Num \to Num$

$idAlpha = \lambda(x{:}\alpha).x \ : \alpha \to \alpha$ ⟵ not VARS — a name w/o a meaning

$\alpha \to \alpha$ an $\beta \to \beta$ are different types, not two different naming choices under $\sim_\alpha$

$id = \ldots \quad : \forall\alpha, \alpha \to \alpha$

↑ binding w a ∀ & how type-vars come about,

(Girard) (reynolds)

System F / polymorphic λ-calculus

abs $\tau ::= \alpha \mid arr(\tau_1, \tau_2) \mid all(\alpha.\tau)$ ⟵ no FV! closed means all vars are bound

concr $\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$

(∀ lam ↓, type ↓)

abs Expr $::= x \mid lam\{\tau\}(x.e) \mid app(e_1, e_2) \mid Lam(\alpha, e) \mid App(e;\tau)$

(big landa ↓)

concrExpr $::= x \mid \lambda(x{:}\tau).e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e\ \tau$

apparently simply typed lambda has no explicit ... (margin note)

the syntax of Judgments

Γ Context $::= \Gamma \mid x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mid \Theta \qquad \Theta ::= \alpha_1, \ldots, \alpha_n$

$\tau : \star$ "tau is a type"

Θ is a list of type vars

$FV(\tau) \subseteq \Theta$

Judgment $::= J \mid \Gamma \vdash e : \tau \mid \Theta \vdash \tau : \star \mid \Theta, \Gamma \vdash e : \tau$

FV sets for types

$$\frac{}{\Theta, \alpha \vdash \alpha : \star} \qquad \frac{\Theta \vdash \tau_1 : \star \quad \Theta \vdash \tau_2 : \star}{\Theta \vdash \tau_1 \to \tau_2 : \star} \qquad \frac{\Theta, \alpha \vdash \tau : \star}{\Theta \vdash \forall\alpha.\tau : \star}$$

"assuming α is allowed in my scope Θ, α is a type"

(arrow construct)

more or less ↓ $BV(\tau) \cap \Theta \approx \{\}$

$$\frac{}{\Theta;\Gamma, x:\tau \vdash x:\tau} \qquad \frac{\Theta;\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Theta;\Gamma \vdash (e_1\ e_2) : \tau'} \qquad \frac{\Theta;\Gamma, x:\tau \vdash e : \tau'}{\Theta;\Gamma \vdash \lambda(x:\tau).e : \tau \to \tau'}$$

rules for simply typed λC but Θ is also for the rule in the context

$$\frac{\Theta;\Gamma \vdash e : \forall \alpha.\tau \quad \Theta \vdash \tau' : \star}{\Theta;\Gamma \vdash e\ \tau' : \tau[\tau'/\alpha]}$$

$$\frac{\cdot;\Gamma \vdash e : \tau}{\Theta;\Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau}$$

System F is typesafe & terminating

R

$$\frac{}{\cdot;\ x:\alpha \vdash \alpha : \alpha}$$

↑ empty context

is the only counterexample to $FV(\tau) \subseteq \Theta$

T

if $\Theta \vdash \tau : \star$ then $FV(\tau) \subseteq \Theta$

if $\Theta;\Gamma \vdash e : \tau$ then $\Theta \vdash \tau : \star$ and $\Theta \vdash \Gamma$

CBN

$$\frac{}{\lambda(x:\tau).e \ \text{val}} \qquad \frac{}{\Lambda\alpha.e \ \text{val}} \qquad \frac{}{x \ \text{val}}$$

$E ::= \square \mid Ee \mid E\tau$ (① ② ③)

$$\textcircled{2}\ \frac{e \mapsto e'}{e\,e_1 \mapsto e'\,e_1} \qquad \textcircled{3}\ \frac{e \mapsto e'}{e\,\tau \mapsto e'\,\tau}$$

$$\textcircled{0}\ \frac{}{(\lambda x:\tau.e)\ e' \mapsto e[e'/x]} \qquad \textcircled{1}\ \frac{}{(\Lambda\alpha.e)\ \tau \mapsto e[\tau/\alpha]}$$

CBNFix : if $\Theta;\cdot \vdash e:\tau$ and $\tau \neq \forall\alpha.\tau'$

- progress: if $\Theta;\cdot \vdash e:\tau$ then either e val or $e \mapsto e'$ for some e'
- preservation: if $\Theta;\Gamma \vdash e:\tau$ is derivable and $e \mapsto e'$ then $\Theta;\Gamma \vdash e':\tau$ is derivable
- Type safety: if $\Theta;\Gamma \vdash e:\tau$ is derivable and $e \mapsto^* e'$ then e' is not stuck
- Type substitution: if $\Gamma, \alpha;\Gamma \vdash e:\tau$ and $\Theta \vdash \tau' : \Delta$ are derivable then so is $\Theta, \Gamma[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$

CBV

$$\frac{e' \text{ val}}{(\lambda(x:\tau).e)\ e' \mapsto e[e'/x]} \qquad \frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1\,e_2 \mapsto e_1\,e_2'}$$

$$\frac{}{(\Lambda\alpha.e)\ \tau \mapsto e[\tau/\alpha]} \qquad \frac{e_1 \mapsto e_1'}{e_1\,e_2 \mapsto e_1'\,e_2} \qquad \frac{e \mapsto e'}{e\,\tau \mapsto e'\,\tau}$$

Const :=

$\lambda x\ \lambda y\ x$

$\Lambda\alpha\ \lambda x{:}\alpha.\ \Lambda\beta.\ \lambda y{:}\beta.\ x$

$\forall\alpha.\alpha \to \forall\beta.\beta \to \alpha$

$R \mapsto R$ (forever)

$\text{erase}((\lambda x:\alpha.\ldots)(\Lambda\beta.R)) \mapsto (\lambda x.\ldots)R$

Intro to Dawen p.4

D

$$erase(x) = x$$

$$erase(\lambda x{:}\tau.e) = \lambda x.\, erase(e)$$

$$erase(e_1\ e_2) = erase(e_1)\ erase(e_2)$$

$$erase(\Lambda\alpha.e) = erase(e)$$

$$erase(e\ \tau) = erase(e)$$

T

type erasure

if $e \mapsto^* v$ then $erase(e) \mapsto^* erase(v)$

Intro to Hoffmann pg. 1

Encoding of types in System F

① Unit

$unit \triangleq \forall \tau.\ \tau \to \tau$

$\langle\rangle \triangleq \Lambda(\tau)\ \lambda(x:\tau).\ x$

② Pairs

$\tau_1 \times \tau_2 \triangleq \forall \rho.\ (\tau_1 \to \tau_2 \to \rho) \to \rho$

$\langle e_1; e_2 \rangle \triangleq \Lambda(\rho)\ \lambda(f:\tau_1 \to \tau_2 \to \rho).\ f\ e_1\ e_2$

$e \cdot l \triangleq e[\tau_1]\ \left(\lambda(x_1:\tau_1).\ \lambda(x_2:\tau_2).\ x_1\right)$

$e \cdot r \triangleq e[\tau_2]\ \left(\lambda(x_1:\tau_1).\ \lambda(x_2:\tau_2).\ x_2\right)$

③ Bin Sums

$\tau_1 + \tau_2 \triangleq \forall \rho.\ (\tau_1 \to \rho) \to (\tau_2 \to \rho) \to \rho$

$l \cdot e_1 \triangleq \Lambda(\rho).\ \lambda(f_1:\tau_1 \to \rho).\ \lambda(f_2:\tau_2 \to \rho).\ f_1(e_1)$

$r \cdot e_2 \triangleq \Lambda(\rho).\ \lambda(f_1:\tau_1 \to \rho).\ \lambda(f_2:\tau_2 \to \rho).\ f_2(e_2)$

$case(e; x_1.e_1; x_2.e_2) \triangleq e[\tau]\ (\lambda(x_1:\tau_1).\ e_1)\ (\lambda(x_2:\tau_2).\ e_2)$

R $unit \to \rho \cong \rho$

④ Nat

$nat \triangleq \forall \rho.\ (unit \to \rho) \to (\rho \to \rho) \to \rho$

$z \triangleq \Lambda(\rho).\ \lambda(z:\rho).\ \lambda(s:\rho \to \rho).\ z$

$s\,e \triangleq \Lambda(\rho).\ \lambda(z:\rho).\ \lambda(s:\rho \to \rho).\ s\ e$

$iter\{e_1; x.e_2\}\ e \triangleq e[\tau]\ (e_1)\ (\lambda(x:\tau).\ e_2)$