Lecture notes by Edward Loper

Course: 6.821 (Programming Languages)
Professor: David Gifford (gifford@lcs)
Institution: Massachusetts Institute of Technology

`http://www.psrg.lcs.mit.edu/6821`

"the rewrite rules on your brain aren't confluent" :)

Lecture

Thursday, September 9, 1999

Programming language consists of a syntax (concrete/abstract), a semantics, and a pragmatics.

1 PostFix

A B C sel → if A then B else C Example programs:

```
1               → 1
4 2 3 add mul   → 20
31 sub 7 mul    → 14
0 2 1 sel       → 2
1 1 2 sel       → 1
1 2 lt          → 1
1 2 gt 7 8 sel  → 8
1 2 swap pop    → 2
2 (3 add) exec  → 5
2 (3 add) 4 swap exec mul → 14
```

Semantic questions left unanswered: what do these do?

```
4 2 div         → 2
4 0 div         → error?  NAN, infinity?
4 3             → 3
3 add           → error?
10 exec         → error? 10?
```

Use abstract syntax tree to define language and help figure out semantics:

```
                        /operational semantics  ⟶ meaning
concrete syntax ⟶ AST --denotational semantics⟶ meaning
                        \ compiler⟶ code --exec⟶ meaning
```

Parse "(2 add) 3 swap exec"... Start with concrete tokens:

```
P ∈ Program
Q ∈ Commands
C ∈ Command

P ::= (Q)
Q ::= C*
C ::= N [intlit] | pop [pop] | swap [swap] | A [arith-op] |
      R [relational-op] | sel [select] | exec [execute] |
      (Q) [executable-sequence]

A ∈ arithmatic op = {add, mul, div, sub}
R ∈ relational op = {<, >, =}
N ∈ intlit = {..., -2, -1, 0, 1, 2, ...}
```

```
([Program] ([Commands]
              ([Executable Sequence]
              ([Commands] ([Intlit] 2) ([arith-op] add)))
              ([IntLit] 3)
              ([Swap] swap)
              ([Execute] execute)))
```

2 Operational Semantics

Program –I⟶ C0 ⇒ C1 ⇒ ... ⇒ Cn ⇒ Cf –O⟶ answer

Ci ∈ C, the configuration space. translate program into a final state (Cf ∈ set of valid final states, F). A semantics is a 5-tuple:

```
< C, ⇒, F, I, O >
```

(swap exec swap exec) (1 sub) swap (2 mul) swap 3 swap exec (5 :))

For postfix (stack language), we maintain 2 pieces of state:

```
C = < Q, stack > = Q x stack
S ∈ stack = value*
V ∈ value = intlit | executable-sequence
answer = value | error
error = {error}
F = <[]command, stack>
I: program → C
        lambda(Q).<Q, []value>
O: F → answer
        lambda(<[]command, F.S').(Value Answer V)
⇒: C x C (relation, not function, to allow nondeterminism)
```

```
. = cons (sort of.. we can do it to end of lists too)
& = append
```

The rules for postfix are:

```
<N.Q, S> ⇒ <Q, N.S>
<(Qexec).Qrest, S> ⇒ <Qrest, Qexec.S>
<pop.Q, V.S> ⇒ <Q, S>
<swap.Q, V1.V2.S> ⇒ <Q, V2.V2.S>
<sel.Q, VF.VT.0.S> ⇒ <Q, VF.S>
<sel.Q, VF.VT.n.S> ⇒ <Q, VT.S>   (n != 0)
{arithmatic operations}
<exec.Q, Qexec.S> ⇒ <Qexec & Q, S>
```

or use a recursive inference rule:

```
<exec.Q, []command.S> ⇒ <Q,S>

        <Qexec, S> ⇒ <Qexec', S>
----------------------------------------
<exec.Qrest, Qexec.S> ⇒ <exec.Qrest, Qexec'.S'>
```

Not all inference rules are good. For example:

```
<loop.Q, S> ⇒ <Q', S'>
----------------------
<loop.Q, S> ⇒ <Q', S'>
```

In general, the antecedent should be "simpler" than the consequent Can't use =/⇒ or ⇒* in antecedents..

Postfix always terminates. But if we add "dup," it may not: "(dup exec) dup exec" will not terminate. But adding dup makes it turing universal.

(in particular, <[dup exec], [(dup exec)]> ⇒* <[dup exec], [(dup exec)]>).

3 Proofs about operational semantics

3.1 Prooving that a language terminates

Define an energy function of configurations. Then prove that energy is strictly decreasing with each transition, and that initial energy must be finite.

Energy Function

```
Econfig [[ <Q,S> ]] = Eseq[[ Q ]] + Estack[[ S ]]

Eseq[[ [] ]] = 0
Eseq[[ C . Q ]] = 1 + Ecom[[ C ]] + Eseq[[ Q ]]

Ecom[[ C ]] = Eseq[[ C ]]  (for any sequence C)
Ecom[[ C ]] = 1            (for C not a sequence)

Estack[[ N.S ]] = 1+Estack[[ S ]]
Estack[[ Q.S ]] = Eseq[[ Q ]] + Estack[[ S ]]
Estack[[ [] ]] = 0
```

Proof for command N

```
<N.Q, S> ⇒ <Q, N.S>
E(<N.Q, S>) = 1+E(N)+E(Q)+E(S) = 2+E(Q)+E(S) =
1+E(Q)+E(N.S) = 1 + E(<Q, N.S>)
```

Proof for exec

```
<exec.Qrest, Qexec.S> ⇒ <Qexec @ Qrest, S>
E(<exec.Qrest, Qexec.S>) =  1+E(exec)+E(Qrest)+E(Qexec)+E(S)
= 2+E(Qrest)+E(Qexec)+E(S) = 2+E(Qexec@Qrest)+E(S)
= 2+E(<Qexec@Qrest, S>)
```

Try proving for dup:

```
<dup.Q, V.S> ⇒ <Q, V.V.S>
E(<dup.Q, V.S>) = 2+E(Q)+E(V)+E(S)
but:
E(<Q, V.V.S>) = E(Q)+E(V)+E(V)+E(S)
```

But note that the energy still decreases, as long as V is a number. Thus, we can safely duplicate numbers. Also, empty command sequences.

Note also that we can duplicate any command sequence that doesn't contain "dup." To do so, simply make the energy of dup very high. Then getting rid of the dup will release enough energy that duplicating the sequence won't give more energy than dup. (the command sequence lengths that we can dup depend on the energy we assign to dup, but since we could give dup arbitrarily high energy, we can do this with arbitraily long command sequences). To do it formally, we could define a tuple-energy <$E_d$up, $E_o$ther> and define <a,b> > <c,d> if a>c or if a=c and b>d... etc.

4 Domains

4.1 Primitive Domains

The elements of the domain are explictly specified. E.g., {error}, {red, green, blue}, {..., -2, -1, 0, 1, 2, ...}. Subscripts can be used on values or variables to label the domain an element is in.

4.2 Compound Domains (Product Domain)

The cross product of two or more domains, i.e., the set of all pairs whose first element is from the 1st domain and 2nd element is from the 2nd domain. E.g., A = B x C x D, Configuration = Commands x Stack. Can use the tuple$_c$ompounddomain operator to construct elements. Or shorthand with angle brackets, e.g., <red, true> ∈ (Colors x Bools) or <B, C, D> ∈ A.

4.3 Sum Domain

The sum of multiple domains, i.e., the set of all elements of each of the summand domains. E.g., if A=Colors+Bools, then true ∈ A and orange ∈ A.

In order to construct elements of a sum domain from a summand domain, use the Inj1$_s$umdomain or Inj2$_s$umdomain etc. Note that sum domains are "tagged," so we can determine which of the constituants it came from. So for example, if B = bool+bool, then we can tell Inj1(true) apart from Inj2(true).

If the summand domains are all distinct, then we can use abbreviations for the Injection functions. For example, if Value=Intlit+Error, then Intlit → Value is shorthand for Inj1$_v$alue. If we write "error," it is in the Error domain. If we want an error value, we write "(Error → Value error).

Extract from sum domains with matching:

```
matching_bool,D Eb
  > (True → Bool Ignore) | Et
  > (False → Bool Ignore) | Ef
endmatching
```

This is essentially equivalant to "(if Eb Et Ef)". Note that "Ignore" is a bound variable.. Also, Eb must be in the bool domain, and Et and Ef must be in the D domain.

4.4 Sequence Domains

A sequence domain is basically an arbitrary number of products of the same domain. For example, Bool*, A*. []$_B$ool, [true]$_B$ool, [true, false, true]$_B$ool. Access the sequence domains with nullp, cons, and car.

4.5 Arrow Domain

A function from one domain to another domain. Constructor is lambda. For example, lambda x . x is in the domain bool → bool (and other domains too, but each one is a different function). Each function takes exactly 1 argument. Another example: + : Intlit → Intlit → Intlit. Note that arrows right-associate, so this is equivalant to Intlit→(Intlit → Intlit). "+ 1 2" is equivalant to "((+ 1) 2)". (currying, after haskell).

5 Expression Language (EL)

5.1 Basic Components of an SOS

```
      < C, ⇒, F, I, O >
C ∈ Configuration = Num-Exp
F ∈ Final = (map Numeral → Num-Exp M)
O = lambda (Numeral → Num-Exp M).M
I = (substitute M for input in N)
```

5.2 Defining ⇒

First define $\Rightarrow_B$, since ⇒ is just defined on N's.

```
(Rop M1 M2) ⇒B a   (where a=relate Rop M1 M2)
(Lop M1 M2) ⇒B a   (where a=do1og Lop M1 M2)
```

Axioms

```
(Aop M1 M2)     ⇒ a   (where a=(calculate Aop M1 M2))
(if true N1 N2)  ⇒ N1
(if false N1 N2) ⇒ N2
```

Inference rules

$$\frac{N1 \Rightarrow N1'}{(Aop\ N1\ N2) \Rightarrow (Aop\ N1'\ N2)} \qquad \frac{N2 \Rightarrow N2'}{(Aop\ N1\ N2) \Rightarrow (Aop\ N1\ N2')}$$

$$\frac{B \Rightarrow_B B'}{(\text{if } B\ N1\ N2) \Rightarrow (\text{if } B'\ N1\ N2)}$$

$$\frac{N1 \Rightarrow N1'}{(Rop\ N1\ N2) \Rightarrow_B (Rop\ N1'\ N2)} \qquad \frac{N2 \Rightarrow N2'}{(Rop\ N1\ N2) \Rightarrow_B (Rop\ N1\ N2')}$$

$$\frac{B1 \Rightarrow_B B1'}{(Lop\ B1\ B2) \Rightarrow_B (Lop\ B1'\ B2)} \qquad \frac{B2 \Rightarrow B2'}{(Lop\ B1\ B2) \Rightarrow_B (Lop\ B1\ B2')}$$

6 Denotational Semantics

```
Syntactic Domains
      |
      | Meaning function
      V
Semantic Domains
```

Don't operationally give a meaning for a program fragment.. Instead, associate a "semantic value" for each program fragment. Then we can assert that 2 program fragments have the same semantic value.

For each syntactic domain D, we can define a semantic value..

For postfix, we start with 2 semantic domains: answers, and (stack → stack) transforms. Functions to map from syntactic domains to semantic domains:

```
Psem: Program → Answer
Qsem: Commands → (Stack → Stack)
Csem: Command → (Stack → Stack)
Nsem: Intlit → Int
Asem: Arithop → (Int → Int → Answer)
Rsem: Relop → (Int → Int → Bool)
```

We want to be able to operate in the semantic domain. If + joins 2 phrases syntactically, and * joins 2 phrases semantically, and f maps syntax → semantics, then we want: f(x1+x2) = f(x)*f(g) (homomorphism)

```
t ∈ Stack-Transform = Stack → Stack
s ∈ Stack = Value* | Error
v ∈ Value = Int | Stack-Transform
a ∈ Answer = Value | Error
      Error = {error}
i ∈ Int = {set of integers}
b ∈ Bool = {true, false}
```

⟦ ⟧ = "element of syntactic domain" (parse the given phrase).

```
Psem⟦ (Q) ⟧ = (top (Qsem⟦ Q ⟧ empty-stack))

Qsem⟦ [] ⟧ = (lambda (stack) stack)
Qsem⟦ C.Q ⟧ = (o Qsem⟦ Q ⟧ Csem⟦ C ⟧)

Csem⟦ N ⟧ = (push (Value → Answer (Int → Value N)))
Csem⟦ pop ⟧ = pop
Csem⟦ swap ⟧ = (lambda (stack)
                       ((push (top (pop stack)))
                          ((push (top stack)) (pop pop stack)
Csem⟦ (Q) ⟧ = (push (value → answer (stack-transform → value Qsem⟦ Q ⟧)))
Csem⟦ A ⟧ (arithop Asem⟦ A ⟧)
```

```
((dup exec) dup exec) maps to bottom.
```

Observational equivlance

```
 X = (1 <hole> plus)
```

Under what circumstances will 2 sequences always give the same result when plugged in for <hole>?

For any two program fragments: Q1, Q2, they are operationally equivalant iff Qsem⟦ Q1 ⟧=Qsem⟦ Q2 ⟧. For example, [1 2 add] is operatonally equivalant to [3], and [(1 2 add)] is operationally equivalant to [(3)].

Semantic domains + helper functions form a "semantic algebra."

Wednesday, September 22, 1999

7 PS1a Comments

Common problem: on problem 1, didn't mention what happened to domains. First, you have to define a new value domain, which contains a pair domain, etc. Also, syntax errors:

```
<> = tuples
[] = sequences
() = application
```

Also injection functions.

8 Denotational Semantics

8.1 Eta Reduction/Expansion

We can eta-reduce things out (e.g., eta-reduce the stack out, and just deal with stack transforms.

```
(lambda (s) (f s)) = f
```

8.2 "Elegant" Stack-Handling

Capture common patterns of stack manipulation. We want to be able to assume for most of our denotational semantics that things will go ok: sweep error handling under the rug – but make sure it's still there.

> With-stack-values handles the error cases of stack transforms for
> us. Basically, we give it a function that maps values to stacks,
> and it will check for error stacks for us. (c.f. with-stack-value)

```
with-stack-values: (Value* → Stack) → Stack-Transform
```

Thursday, September 23, 1999

9 Fixed Points

Examples:

```
FactGen = (lambda (f) (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))
Even0Gen = (lambda (f) (lambda (n) (if (= n 0) 0 (f (- n 2)))))
```

then FactGen(fact)=fact and Even0Gen(even0)=even0. Note that this definition is self-referential.

Model theory creates a mapping between concrete models and mathematical consistancy...

In Even0, it is defined for 0, 2, 4, 6, ... But for odd numbers? It can be anything: the functional definition doesn't specify.

FactGen has one fixed point (excluding negatives): fact. Even0 has a countable number of fixed points.

We need to be able to select one fixed point.

There is a function, fix, which will give you the fixed point of a function. E.g.:

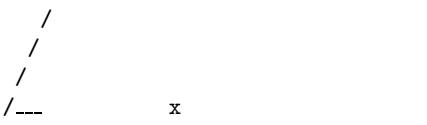
```
fix(FactGen) = fact
fix(Even0) = {<0,0>, <2,0>, <4,0>, ...}
```

But we want fix to give us "bottom" where even0 is undefined.

Whether or not a function has a fixed point (& the # of fixed points) depends on the domain of the functions.

```
function
f(x) = 1-x    1 fixed point: 1/2
f(x) = 4/x    2 fixed points: 2, -2
f(x) = x      infinite fixed points
```

Consider continuous functions on the unit interval. Then it has a fixed point:



Something will have a fixed point if it crosses the line f(x)=x. Because a continuous function on the unit interval must cross this line, it must have a fixed point. But discontinuous funciton doesn't have to have a fixed point.

9.1 Partial order

It is reflexive, antisemetric, and transitive

```
a weaker-than a
if a weaker-than b, b weaker-than a, then a=b
if a weaker-than b, b weaker-than c, then a weaker-than c
```

Use symbol $\sqsubseteq$ for weaker-than.

9.2 Least Upper Bounds

Define a least upper bound on the amount of information in a set. Set S be a subset of D. If D contains an element a s.t. for all x in S, x$\sqsubseteq$ a, then a is the LUB of D...

Let S $\in$ D LUB(S) = x $\in$ D s.t.

1. $\forall$ y $\in$ S, y $\sqsubseteq$ x

2. $\forall$ z $\in$ D, ($\forall$ y $\in$ S, y $\sqsubseteq$ z) $\rightarrow$ (x $\sqsubseteq$ z)

9.3 Information Content

How can we define the information content of a function? For two functions, f and g: X $\rightarrow$ Y

- f $\sqsubseteq$ g if $\forall$ x $\in$ X f(x) $\sqsubseteq$ g(x)

Information content:

```
1   2
 \ /
 bot          (bot is written ⊥)
```

Functions:

```
g1    g2      g1(1)=1, g1(2)=1; g2(1)=2, g2(2)=2
  \  /
   f1         f1(1)=⊥, f1(2)=⊥
```

etc.

Usually, F($\perp$)=$\perp$.

```
f0 = {} (all elided values map to bottom)
f1 = FactGen f0 = {<0,1>}
f2 = Factgen f1 = {<0,1>, <1,1>}
```

Thus, $f_n \sqsubseteq f_n+1$. Fixed point is the lim(n $\rightarrow$ inf) f_n. FactGen: (Nat $\rightarrow$ Nat$_b$ot) $\rightarrow$ (Nat $\rightarrow$ Nat$_b$ot)

Definitions:

- Chain = a totally ordered subset of a partial order.
- Complete partial order = every chain C in D has a least uppper bound in D (so LUB(C) exists) (aka CPO)
- Pointed Complete Partial Order is a CPO with a least element.
- Monotonic: if d1, d2 $\in$ D, then if d1 $\sqsubseteq$ d2 then f(d1) $\sqsubseteq$ f(d2) I.e., if you give a function less information, the function's output must be weakened.
- Continuity: f:D $\rightarrow$ E, D and E are CPOs. Then for all chains S $\in$ D F(LUB_D S) = LUB_E {f(x)|x $\in$ S} Continuity implies monotonicity.

If there are many fixed points, pick the one with the least information content.

Continuity implies monotonicity:

```
C = {d1, d2} f:D → E, d1 ⊑ d2
(f (LUB C)) = LUB{f(c)|c ∈ C}.  Let C={d1, d2}
(f d2) = LUB({f(d1), f(d2)}), so f(d1) ⊑ f(d2)
```

Rules should guarantee monotonic and continuity. Consider halt = (lambda (x) (= x $\perp$)).. It's neither monotonic nor continuous.

Consider always$_f$alse, which returns false for anything (even $\perp$). This is at least monotonic. Continuous?

Monotonicity and continuity guarantee that the fixed point operator will converge...

Fixed Point Theorem: If D is a pointed CPO, then a continuous function f:D $\rightarrow$ D has a fixed point that is least:

```
fix(f) = LUB{f^n(⊥) | n>=0}
```

$\perp \sqsubseteq f(\perp)$ $f^n \perp \sqsubseteq f^(n+1) \perp$

1. Show that fix f is a fixed point of f:

```
f(fix f)
      = f(LUB {f^n(⊥)|n>=0})
      = LUB {f^n(⊥)|n>=1}
      = LUB {f^n(⊥)|n>=6}
      = fix(f)
```

1. Prove that it is least fixed point in D (by contradiction).

```
Assume d is the least
d = f(d)
⊥ ⊑ d
f(⊥) ⊑ f(d)
f^n(⊥) ⊑ d
LUB {f^n(⊥)|n>=0} ⊑ d
fix(f) ⊑ d
```

Assume A, B are CPOs. We should show that A+B, AxB, A*, and A → B are CPOs. Especially difficult part of proof is that D=D → D is a CPO...

Define fix:

```
fix = (lambda f (lambda (x) (f (x x))) (lambda (x) (f (x x))))
```

Tuesday, September 28, 1999

10 Functional Language (FL)

- Every statement or expression has a value
- Two pieces: FLK (kernel), and FL.
- We can desugar FL → FLK.

10.1 Examples

- Dynamically Typed: (if 0 #t #f) = error
- Normal order (lazy): (f $E_b$ot)
- Implicitly curried: (f 1 2) = ((f 1) 2)

```
(proc x (primop * x x)) = square
```

```
(pair (primop not? #f) (primop / 1 0)) → pair with #t in left
```

branch.. If we eval right branch, we'll get an error

```
(call (proc x (call x x)) (proc x (call x x))) → ⊥
```

factorial:

```
(rec fact (proc n (if (primop = n 0) 1
                      (primop * n (call fact (primop - n 1))))))

(rec l (pair 1 (pair 2 l))) → (1 2 1 2 1 2...)

(rec x x) → bottom
```

10.2 Syntax of FLK

E::= ; expressions

- (primop P E*) ; primitives (26 P's)
- (proc I E)
- (call E1 E2)
- I ; identifiers
- (if E1 E2 E3)
- L ; literals
- (pair E1 E2) ; non-strict (lazy)
- (rec I E) ; bind E to I in E, returns E...

L::= ; literals

- #u
- #f
- #t
- N
- (symbol I)

(rec I E) is analagous to (fix (proc I E))

10.3 Syntax of FL

E::=

- FLK expressions
- (lambda (I*) E)
- (E1 E*)
- (list E*)
- (quote S*)
- (cond (Epredicate Econsequent)* (else Ed)?)
- (and E*)
- (or E*)
- (let ((I E)*) E)
- (letrec ((I E)*) E)

S::=

- L
- (S*)

In FL, there are also lots of pre-bound functions.

10.4 Desugaring from FL ⇒ FLK

```
Dexp ⟦  (lambda () E)  ⟧ = (proc Ifresh Dexp⟦ E ⟧)
Dexp ⟦  (E)  ⟧ = (call Dexp⟦ E ⟧ #u)
Dexp ⟦ (lambda (I1 I+) E) ⟧ = (proc I1 (proc Dexp⟦ (lambda (I+) E) ⟧))
Dexp ⟦  (E1 E2 E3+)  ⟧ = D⟦ ((call D⟦ E1 ⟧ D⟦ E2 ⟧) E3+) ⟧
```

```
Dexp ⟦  (letrec ((I1 E1) ... (In En)) Ed)  ⟧ =
 (let ((Iouter (rec Iinner
                (let ((I1 (nth Iinner 1)) ... (In (nth Iinner n)))
                   (list E1 ... En)))))
    (let ((I1 (nth Iouter 1)) ... (In (nth Iouter n)))
       Ed))
```

10.5 Variables and Identifiers

Variable - value bound by proc or rec

Identifier - symbol that stands for a variable

Free Identifier - An identifier that is not bound by an enclosing proc or rec.

Bound Identifier - An identier that is named in an enclosing proc or rec

We can define corresponding functions Free-Ids⟦ E ⟧ and Bound-Ids⟦ E ⟧

Variable Capture

Occurs when we alpha-rename, and a variable's binding changes:

```
(proc a (proc b (call a c)))
Renaming a to c causes problem (external capture)
Renaming a to b causes problem (internal capture)
```

10.6 Substitution

Operator:

```
[E1 / I] E2
```

This operator says to replace all free I's in E_2 with E_1. (read "substitute E_1 for free I in E_2")

```
[E1/I] (if E2 E3 E4) ⇒ (if [E1/I]E2 [E1/I]E3 [E1/I]E4)
[E1/I] (proc I E2) ⇒ (proc I E2)
[E1/I'] (proc I E2) ⇒ (proc Ifresh [E1/I']([Ifresh/I]E2))
```

10.7 SOS for FLK

```
< Exp, ⇒, Val-Exp, I, O >

I = identity
O = (lambda (V) (alpha-class V))
Val-Exp = L | (pair a b) | {(proc I E)} | I

(call (proc I1 E1) E2) ⇒ [E2/I1] E1

        E1 → E1'
    -------------------
(if E1 E2 E3) ⇒ (if E1' E2 E3)

(if #t E2 E3) ⇒ E2
(if #f E2 E3) ⇒ E3

(rec I E) ⇒ [(rec I E)/I] E
     (e.g., (rec l (pair 1 (pair 2 l))) ⇒ (pair 1 (pair 2 (rec l ..)))
```

Wednesday, September 29, 1999

11 Fixed Points

11.1 Least Fixed Point Theorem

Says that fix(D) exists for D which obey certain conditions.

We need to make sure that the least upper bound exists. E.g., if we were trying to find f(x) by successive approximations with less information than f(x), we need to make sure that f(x) exists in our domain..

Pointedness gives us a "starting point" (bottom). We can make any CPO pointed by just adding "bottom" to it.

D $\rightarrow$ E: continuous (i.e., preserves least upper bounds)

- Take a chain C $\subset$ D
- f(LUB_D C)=LUB_E f(C)

Then $\text{fix}_\text{D}(\text{f}) = \text{LUB}_\text{D}\ \text{f}^\text{n}(\bot)$

11.2 Operations on CPOs

+, x, bot, $\rightarrow$, *.

Assume D, E are CPOs

- D x E is a CPO with "product ordering"
- To make D+E a PCPO, we need to create a new bottom.

11.3 Desugaring letrec (again)

```
Church list:
 (lambda (Ireciever)
     (Ireciever I1...In))
```

12 Dynamic Environments

```
(let ((a 1))
  (let ((f (lambda (x) (+ a x))))
     (let ((a 20))
         (f 300))))
```

For static scoping, answer = 301. For dynamic scoping, answer = 320.

Text desugars to:

```
(call (proc a
  (call (proc f
    (call (proc a (call f 300))
          20))                          ; let a=20
    (proc x (primop + a x))))           ; let f=(lambda ..)
 1)
```

In dynamic scoping, free identifiers are defined by caller, not callee..

Problematic, since calle can't preserve invariants. Callee basically can't preserve invariants.

Useful, since you can redefine variables that callee will use, e.g., redefine sqrt, or redefine input stream. But hard to tell what will or won't break the callee.

Advantage over using a single global variable: it will automatically undo your changes once you leave your environment.

Not used very often.

From static scoping:

```
p ∈ Procedure = Denotable → Computation
E⟦ (proc I E) ⟧ = (lambda (e)
                    (val-to-comp (Procedure → Value
                                    (lambda (d) E⟦ E ⟧ [I:d]e))))
E⟦ (call E1 E2) = (lambda (e) (with-procedure (E⟦ E1 ⟧ e))
                                  (lambda (p) (p (E⟦ E2 ⟧ e))))
```

For dynamic scoping:

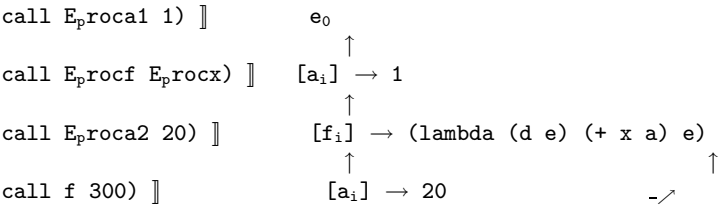
```
p ∈ Procedure = Denotable → Environment → Computation
E⟦ (proc I E) ⟧ = (lambda (e_def)    ; e_def gets ignored
                    (val-to-comp
                        (Procedure → Value
                           (lambda (d e_call) E⟦ E ⟧ e_call))))
E⟦ (call E1 E2) = (lambda (e) (with-procedure (E⟦ E1 ⟧ e))
                                  (lambda (p) (p (E⟦ E2 ⟧ e) e)))
```

Variable lookups in dynamically scoped languages very expensive: since you don't know anything about the environment you're getting, you have to search the environment to find each variable. :(

Static:

```
E⟦ (call E_proca1 1) ⟧        e_0
                               ↑
E⟦ (call E_procf E_procx) ⟧   [a_i] → 1                    ←——
                               ↑                                \
E⟦ (call E_proca2 20) ⟧       [f_i] → (lambda (d) (+ x a) e[a/*])
                               ↑
E⟦ (call f 300) ⟧              [a_i] → 20
```

Dynamic:



13 Packaged Environments

4 primitives: record, selectt, override, conceal. Everything else is sugar.

13.1 records

Create a data structure with name/value pairs:

```
(define joe (record (age 23) (male #t)))
(select age joe)
```

records as environments

We can use joe as an environment (or a partial environment):

```
(with (age male) joe
  (+ 10 age))
```

Desugars to:

```
(with (I1 I2 ... In) E1 E2)
  ⇒
(let ((I E1))
   (let ((I1 (select I1 I))
         (I2 (select I2 I))
         ...
         (In (select In I)))
     E2))
```

If we don't specify I1...In, it's very hard to tell which variables the with expression is binding.

To package x,y,z into an environment:

```
(record (x x) (y y) (z z))
```

13.2 modules

What if we want an environment with mutually recursive bindings?

```
(module (I1 E1) ... (In En))
  ⇒
(letrec ((I1 E1) ... (In En)) (record (I1 I1) ... (In In)))
```

13.3 Misc primitives

override

Appned 2 records, giving precedence to E2. Gives subclassing.

```
(override E1 E2)
(override joe (record (age 32)))
```

conceal

Makess certain values disappear – not accessable from the outside anymore. Allows us to encapsulate variables, protect abstraction barriers.

```
(conceal (I1 ... In) E)
```

13.4 Example

```
(define make-point
  (lambda (x y)
    (module (x x) (y y)
            (rho (sqrt (+ (* x x) (* y y)))) (theta (atan y x)))))
(define p (make-point 1 2))
(with (rho theta) p (* rho rho))
```

Recitation

Wednesday, October 6, 1999

14 Naming

14.1 Parameter Passing

What do names mean?

Call-by-name (call-by-need?)

You can name arbitrary computations (e.g., bottom and errors)

Call-by-value

You can only name values

Call-by-denotation

Denotable = Environment → Computation.

Strictness

14.2 (Hierarchical) Scoping

How do you organize your names? What names are visible, and how do you decide which name shadows which?

Static

Dynamic

14.3 Non-Hierarchical Scoping

Packaging up bunches of names and using them, denoting them.

Records

Modules

OOP

14.4 Bizzare Naming

Multiple namespaces

Mixed scoping

15 Object Oriented Programming

15.1 HooPLA

E ::=

- L
- I
- (method M (Iself Iformal*) E)
- (object-compose E1 E2)
- (null-object)
- (send M Eobj Earg*)
- (object E*)
- (class (Iinit*) Einstance*)

Desugaring of class:

```
D⟦ (class (Iinit*) E*) ⟧ =
      (method make (Ignore Init*) (object E*))
```

Example:

```
(define point (class (init-x init-y)
      (method x (self) init-x)
      (method y (self) init-y)
      (method move (self dx dy)
              (object (send make point (send + (send x self) dx)
                                       (send + (send y self) dy))
                      self))))
```

Example:

```
(define color (class clr)
      (method color (self) clr)
      (method new-color (self new) (object (send make color new)
                                           self)))
```

When we make a new point, point doesn't know about all the methods you support. For example, consider:

```
(define colored-point (class (x y col)
                             (send make point x y)
                             (send make color col)))
```

Then if we call the move method on a colored point, we'll get back a colored point! This happens because the (object (send ..) self) clause combines the new methods (send ...) with the old ones (self).

16 State

```
Store = Location → Assignment
Location = Nat
Assignment = (Storable | Unbound)⊥
Storable = Value            {depends on language}
```

Keep E as:

```
E = Exp → Envvironment → Computation
```

But we have to redefine Computation:

```
Computation = Store → Store x Expressible
Expressible = (Value | Error)⊥
Value = Int + Bool + Procedure + Location
```

New types of E:

- (cell E)
- (primop cell-set! E1 E2)
- (primop cell-get E1)

```
E⟦ (cell E) = (lambda (e) (with-value E⟦ E ⟧ e)
                (lambda (v)
                  (allocating v
                    (lambda (l)
                      (val-to-comp (Location → Value l))))))
```

Define with-value:

```
with-value: Computation → (Value → Computation) → Computation
  = Computation → (Value → Computation) → Store → Store x Expressible

(lambda (c) (lambda (f) (lambda (s)
   (matching (c s)
       (<s1, (value → expressible v)> ((f v) s1))
       (<s1, (error → expressible E)> <s1, (error → expressible E)>)
   ))))
```

Define allocating:

```
allocating: Storable → (Location → Computation) → Computation

(lambda (storable) (lambda (f) (lambda (s)
  ((f (fresh-loc s)) (assign (fresh-loc s) storable s)))))
```

Define sequence, which evaluates 1st argument, ignores its value, and then evaluates second argument.

```
E⟦ (sequence E1 E2) ⟧ =
    (lambda (e) (with-value (E⟦ E1 ⟧ e)
                        (lambda (v) (E⟦ E2 ⟧ e))))
```

17 FLAVAR!

```
(set! I E)
```

E.g.:

```
(let ((a 0)) (f (lambda (x) (+ x x))))
      (f (begin (set! a (+ a 1)) a))
```

- Call by value: 2
- Call by name: 3
- Call by need: 2

(Call by need: like call by name, but keep track of whether we've evaluated it yet.. if we have, then just use old value (memoizing))

```
(let ((a 0)
      (double (lambda (in out) (set! out (+ in in)))))
  (begin
    (double 17 a)
    (+ a 1)))
```

- Call by reference: 35

```
Denotable = Location
Storable = Value            {CBV or CBRef}
Storable = Computation      {CBN}
```

```
E⟦ I ⟧ = (lambda (e) (with-denotable (lookup e I)
                        (lambda (l) (fetching l val-to-comp))))
```

Tuesday, October 12, 1999

18 Standard Semantics

The standard way to describe the semantics of a programming language. Use continuations to model control transfers.

SOS is good with parallelism, multi-processing Denotational Semantics good with non-local control transfer

So far, we model state with:

```
E: Exp → Environment → Store → <Expressible x Store>
```

Where we called "Store→<Expressible x Store" a computation.

Add continuations (k). A continuation is a function that recieves a value and a new store.

```
     ************ <-k- [         ]
e → *E⟦ (...) ⟧* -v→ [ expcont ] → result
s → ************ -s→ [         ]
```

So:

```
E: Exp → Environment → Expcont → Store → Expressible
Cmdcont = Store → Expressible
Exprcont = Value → Store → Expressible
Computation = Expcont → Sore → Expressible
Procedure = Denotable → Computation
```

("Direct Semantics")

Allows us to specify any continuation (i.e., where the value will go) that we want.

```
L: Lit → Value
E⟦ L ⟧ = (lambda (e k s) ((k L⟦ L ⟧) s))
```

```
TL⟦ E ⟧: Exp → Expressible
TL⟦ E ⟧ = ((E⟦ E ⟧ empty-env
                   (lambda (v s') (value → expressible v))
                   empty-store)
```

Store is last argument so we can eta-convert stores away in any form that doesn't modify the store:

```
E⟦ L ⟧ = (lambda (e k s) ((k L⟦ L ⟧) s))
       = (lambda (e k) (k L⟦ L ⟧))
```

```
E⟦ I ⟧ = (lambda (e k) (ensure-bound (lookup e I) k))
```

```
ensure-bound: Binding → Expcont → Store → Expressible
ensure-bound: Binding → Computation

(define (ensure-bound b k s)
  (matching (b)
    ((unbound → binding unbound) (error → expressible
                                  (sym → error unbound-identifier)))
    ((denotable → binding v) (k v s))))

(define (ensure-bound b k)
  (matching (b)
    ((unbound → binding unbound) (lambda (s) (error → expressible
                                  (sym → error unbound-identifier))))
    ((denotable → binding v) (k v))))
```

```
E⟦ (proc I E) ⟧ =
  (lambda (e k1)
     (k1 (Procedure → Value (lambda (d k2) (E⟦ E ⟧ [I:d]e k2)))))
```

```
E⟦ (call E1 E2) ⟧ =
  (lambda (e k1)
    (with-procedure (E⟦ E1 ⟧ e)
           (lambda (p) (E⟦ E2 ⟧ e (lambda (v) (p k v))))))
```

Note that this implements tail recursion directly: it gives the continuation to the proc, so the proc will return directly to whatever place actually needs the value.

```
E⟦ (begin E1 E2) ⟧ =
  (lambda (e k)
    (with-value (E⟦ E1 ⟧ e)
                (lambda (v) ((E⟦ E2 ⟧ e) k))))

E⟦ (begin E1 E2) ⟧ =
  (lambda (e k)
    ((E⟦ E1 ⟧ e) (lambda (v) ((E⟦ E2 ⟧ e) k))))
```

In this case, we're creating a continuation to pass to (E⟦ E1 ⟧ e). That continuation function just figures out (E⟦ E2 ⟧ e2) and applies it to the original continuation...

```
with-procedure : (Expcont → Store → Expressible) →
                   (Procedure → Store → Expressible) →
                     Store → Expressible
with-procedure :
    Computation → (Procedure → Cmdcont) → CmdCont
```

```
E⟦ (call (proc i (primop + i 1)) 2) ⟧

(lambda (e k)
(with-procedure (E⟦ (proc i (primop + i 1)) ⟧ e)
                (lambda (p) ((E⟦ 2 ⟧ e) (lambda (v) p v k)))))

(lambda (e k)
((lambda (p) (p (Int → Value 2) k))
      (E⟦ (proc i (primop + i 1)) ⟧ e)))

(lambda (e k)
((lambda (p) (p (Int → Value 2) k))
      (lambda (d k2) (E⟦ (primop + i 1) ⟧ [i:d]e k2))))

(lambda (e k)
  (E⟦ (primop + i 1) ⟧ [i:2]e k)

(lambda (e k) (k 3))
```

Wednesday, October 13, 1999

19 Direct Semantics (FL & Mutation)

- Mutible Cells, Store
- FLAVAR! (and set!)
- Parameter Passing Mechanism

20 Standard Semantics

- Continuations
- Valuation Clauses
- Helper Functions
- Control Problem
- Expcont & Cmdcont

We can now give values to errors:

```
E: Exp → Environment → Expcont → Store → Expressible
Exprcont = Value → Store → Expressible

E⟦ (error Y) ⟧ =
  (lambda (e k) (error-cont Y⟦ Y ⟧))
```

Valuation for if:

```
Expcont = Value → Cmdcont
Cmdcont = Store → Answer
Computation = Expcont → Cmdcont

E⟦ (if E1 E2 E3) ⟧ =
  (lambda (e k) ((E⟦ E1 ⟧ e)
                 (lambda (v)
                   (matching (v)
                      ((bool → value true) (E⟦ E2 ⟧ e k))
                      ((bool → value false) (E⟦ E3 ⟧ e k))
                      ( (error-cont 'non-boolean))))))
```

20.1 Example with Continuations

Define a new language: FLK! | (loop E) | (jump) | (exit E)

Sample program:

```
(let ((c (cell 0)))
  (loop
    (begin (cell-set! c (+ (cell-ref c) 1))
           (if (> (cell-ref c) 10)
               (exit (cell-ref c))
               (jump)))))
; ⇒ 11
```

The problem: define valuation functions for loop and exit..

First, change the signature of E:

```
E: Exp → Environment → Exitcont →
     Jumpcont → Expcont → Store → Expressible
j ∈ Jumpcont = Cmdcont
w ∈ Exitcont = Expcont
CmdCont = Store → Expressible

E⟦ (jump) ⟧ = (lambda (e w j k) j)

E⟦ (exit E) ⟧ =
  (lambda (e w j k) (E⟦ E ⟧ e w j w))

E⟦ (loop E) ⟧ =
  (lambda (e w j k)
    (fix_c mdcont (lambda j (E⟦ E ⟧ e k j k))))
                  ;                                    ; This k is only used if we don't exit via exit.
```

Next problem: Show that E⟦ (loop (jump)) ⟧ = (lambda (e w j k) bot$_c$mdcont).

Plug E⟦ (jump) ⟧ into our value for E⟦ (loop (jump)) ⟧] and we get:

```
(fix_c mdcont (lambda j ((lambda (e w j k) j) e k j k))) =
(fix_c mdcont (lambda j j))
```

Midterm review Sun 10/17 7-9pm Room: <TBA>

21 Control

Modelling non-local control transfers.

21.1 Label and Jump

Define a new FLK-based language:

```
E ::= ... | (label I E) | (jump E1 E2)
```

Examples:

```
(label n
  (+ (jump n 2) 2)) ⇒ 2
```

```
(let ((p (lambda (c v)
           (jump c (+ 1 v)))))
  (label done (p done 2))
```

Note in this example that the jump is called where it's not statically surrounded by a label – jumps are dynamic.

```
(label x (+ 1 (label x (jump x 0)))) ⇒ 1
```

Add a label (ControlPoint) to the value domain:

```
Value = ... + ControlPoint
ControlPoint = Value → Store → Expressible = Expcont
```

```
E⟦ (label I E) ⟧ =
  (lambda (e k) (E⟦ E ⟧ [I:(ControlPoint → Value k)]e k))
```

```
E⟦ (jump E1 E2) ⟧ =
  (lambda (e k) ((E⟦ E1 ⟧ e)
                 (test-control-point
                      (lambda (k2) (E⟦ E2 ⟧ e k2)))))
```

Now consider the example:

```
(jump (label x x) (label y y))
E⟦ (label x x) ⟧ = (lambda (e k) (k (ControlPoint → Value k)))

(jump (label x x) E) = (jump E E)
(jump (label x x) (label y y)) = (jump (label y y) (label y y))
      =⊥
```

21.2 Exceptions

Define a new FLK-based language:

```
E := ... | (raise I E) | (trap I E1 E2)
```

- (raise I E) raises an exception named I..
- (trap I E1 E2) Evaluates E2, trapping an exception named I. Value of the exception is passed to the procedure E1, and the return value of E1 determines the value of raising the exception.

```
(let ((p (lambda (x) (raise out x))))
   (trap out (lambda (y) (+ 1 y))
       (p 0)))
```

```
(trap a (lambda (x) 3)
   (trap b (lambda (x) (raise a 5)))
      (trap a (lambda (x) 7)
         (raise b 11)))
```

Trap handlers are dynamically scoped, so this evaulates to 7, not 3.

We need to redefine computation:

```
E⟦ ⟧ : Exp → Environment → Computation
Computation = Handler-Env → ExpCont → Cmdcont
w ∈ HandlerEnv = Identifier → Procedure
Procedure = Denotable → Computation
Empty_Handler = (lambda (I d) (error-to-comp I))
```

```
E⟦ (trap I E1 E2) ⟧ =
  (lambda (e w k)
     (E⟦ E1 ⟧ e w (test-procedure
                    (lambda (p)
                       (E⟦ E2 ⟧ e (extend-handlers w I p) k)))))

E⟦ (raise I E) ⟧ =
  (lambda (e w k) (E⟦ E ⟧ e w (lambda (v) ((w I) v w k))))

E⟦ (call E1 E2) ⟧ =
  (lambda (e w k) (E⟦ E1 ⟧ e w
     (test-procedure (lambda (p)
                       (E⟦ E2 ⟧ e w (lambda (v) (p v w k)))))

E⟦ (proc I E) ⟧ =
  (lambda (e w k)
    (k (Proc → Value (lambda (v w' k') (E⟦ E ⟧ [I:v]_e w' k'))))))
```

Termination semantics: Change it so trap always returns to its continuation, even if E1 tries to return to some other continuation. Call this new version handle.

```
E⟦ (handle I E1 E2) ⟧ =
  (lambda (e w k)
     (E⟦ E1 ⟧ e w (test-procedure
                    (lambda (p)
                       (E⟦ E2 ⟧ e (extend-handlers w I
                           (lambda (v2 w2 k2) (p v2 w2 k)) k))))))
```

We can do this with sugar instead:

```
D⟦ (handle I E1 E2) ⟧ =
  (label I1 (trap I (proc I2 (jump I1 (D⟦ E1 ⟧ I2)))
                  (jump I1 (D⟦ E2 ⟧))))
```

22 Continuation Passing Style

A source-to-source translation that ensures that no procedure ever returns. Thus, we don't need control stacks..

```
C: FLK! → FLK!CPS
Top: FLK! → FLK!CPS
```

```
Top⟦ E ⟧ = (call C⟦ E ⟧ (proc x x))
C⟦ L ⟧ = (proc k (call k L))
C⟦ I ⟧ = (proc k (call k I))
C⟦ (primop P E) ⟧ =
 (proc k (call C⟦ E ⟧ (proc v (call k (primop P v)))))
C⟦ (proc I E) ⟧ =
 (proc k1 (call k1 (proc I (proc k2 (call C⟦ E ⟧ k2)))))
C⟦ (call E1 E2) ⟧ =
 (proc k
   (call C⟦ E1 ⟧
         (proc v1 (call C⟦ E2 ⟧ (proc v2 (call (call v1 v2) k))))))
```

Tuesday, October 19, 1999

23 Explicit Types

Define type as a set of values? A description of a value? An approximation of a value (i.e., say that the type has less information content than any of the elements of the type...??)

23.1 Terminology

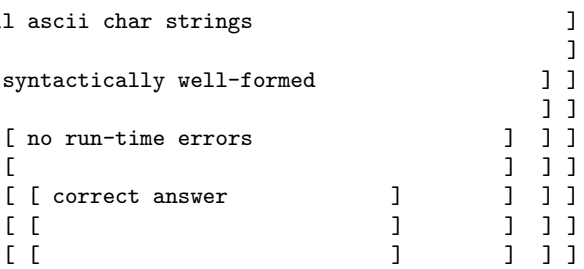
We can prove that expression E has type T. $\vdash$ E : T We can also say that with respect to a type environment A, expression E has type T. A $\vdash$ E : T

A[x:int] $\vdash$ x : A A[x:int] $\vdash$ x $\rightarrow$ x: A $\rightarrow$ A

So "$\vdash$" = provable; ":" = has type; "A" = type environment.

23.2 Overall claims of typing

Different factions about types:



Type people make the following assertion: well-typed programs is contained in programs with no run-type errors. Also, it intersects the correct-answer space. Also, well-typed programs are more likely to give correct answers..?

23.3 Scheme/X

Scheme with explicit types.

We want to catch run-time typing errors:

- applying a non-proc
- if on a non-boolean
- applying a primop to wrong types.
- correct # of arguments to procedures
- all parameters must be of correct type (?)

Sound type system: prove with operational semantics and typing rules that certain classes of errors cannot occur on well-typed expressions.

Our typings sytem: T := int | bool | sym | ($\rightarrow$ (T1 .. Tn) Tr) | unit

Change the syntax of lambdas:

```
(lambda ((x xtype) (y ytype) (z ztype)) body)
```

Define axioms of the form:

```
⊢ 1:int
⊢ #t:bool
⊢ (lambda ((x int) (+1 x))) : (→ (int) int)
⊢ (lambda ((x int)) (lambda ((y int)) (+ x y))) :
              (→ (int) (→ (int) int))
```

23.4 Typing rules

```
A[I:T] ⊢ I:T                                        [var]

            A[I1:T1, ..., In:Tn] ⊢ Eb : Tb
      ---------------------------------------- [lambda]
A ⊢ (lambda (I1 T1) ... (In Tn) Eb) : (→ (T1 .. Tn) Tb)

  A ⊢ Ei:Ti 1≤ i≤ n  ;  A ⊢ E0: (→ (T1 .. Tn) Tr)
      ---------------------------------------- [app]
              A ⊢ (E0 E1 ... En): Tr

       A | E1: bool ;  A | E2: T ;  A | E3: T
      ---------------------------------------- [if]
               A ⊢ (if E1 E2 E3) : T
```

We can simply look at a form and figure out its type. For example, for any A, it's true that:

```
A[x:int] ⊢ 1: int                                   [intlit]
A[x:int] ⊢ x: int                                   [var]
A[x:int] ⊢ +: (→ (int int) int)                     [var]
A[x:int] ⊢ (+ 1 x): int                             [app]
A ⊢ 1:int                                           [intlit]
A ⊢ (lambda ((x int)) (+ 1 x)): (→ (int) int)       [lambda]
A ⊢ ((lambda ((x int)) (+ 1 x)) 1): int             [app]
```

So to do typechecking, just apply whatever rule applies to the expression.. Recursively check all the subexpressions.. Will find the type of the program, if it has one.

Is it possible to do type checking with dynamic scoping?? E.g., typing on exceptions?

24 Subtyping

24.1 one-of (sum type)

Examples:

```
(define-type nlights (one-of (car int) (bike bool)))
  ⊢ (one nlights car 4) : nlights
  ⊢ (one nlights bike #t) : nlights
⊢ (lambda ((x nlights))
      (tagcase x
        (car lights lights)
        (bike b (if b 1 0)))): (→ (nlights) int)
```

```
A ⊢ E:T  ; ∃ Ii = I and T=Ti
-------------------------------------------
A ⊢ (one (one-of ((I1 Tn) .. (In Tn))) I E):
              (one-of (I1 T1) .. (In Tn))
```

Note that "define-type" is basically just a macro facility that replaces nlights with (one-of ...); thus we can't tell apart 2 types with the same def & different names...

24.2 pair-of (product type)

24.3 record-of (named product type)

24.4 rec-of (recursive types)

Examples:

```
(pair-of int (rec-of t (pair-of bool (pair-of int t))))
(rec-of t (pair-of int (pair-of bool t)))
```

How do we know if two rec-of types are equivalant?? (rec-of I T) = [(rec-of I T) I] T

```
(define-type tree
  (rec-of t (oneof (leaf int))
                   (inner (record-of
                          (left t)
                          (right t)))))

(one tree inner (record (left (one tree leaf 1))
                        (right (one tree leaf 2))))
```

(Is this really a tree, or just a generic directed graph? Can't we define "trees" that aren't trees? :))

24.5 Subtyping

"T1 $\sqsubseteq$ T2" means T1 is a sybtype of T2. I.e., we can use a T1 where a T2 is expected.

Depending on our implementation, either or both of these may be true:

```
(record-of (age int) (member bool)) ⊑ (record-of (age int))
(record-of (age int) (member bool)) ⊑ (record-of (member bool))
```

```
(one-of (car int)) ⊑ (one-of (car int) (bike bool))
```

Mutable things can't be sybtyped. E.g., (cell-of E) isn't a subtype of anything (except itself).

```
           Tr ⊑ Tr'; Ti' ⊑ Ti
----------------------------------------
(→ (T1 .. Tn) Tr) ⊑ (→ (T1' .. Tn') Tr')
```

Tuesday, October 26, 1999

25 Type Reconstruction

Questions:

- How do you know the types are correct?
- What classes of errors are guaranteed not to occur?
- Is it guaranteed to discover the types of any well-typed program?

Examples:

```
⊢ (lambda (x) (+ x 1)) : (→ (int) int)
⊢ (lambda (x) ((x 1) 2)) :
              (→ (→ int (→ int t)) t)
⊢ (lambda (x) (+ 1 ((x 1) 2))) :
              (→ (→ int (→ int int)) int)
```

Define Scheme/R:

```
P :: (program Ebody (define I E)*)
E :: (E E*) | I | (lambda (I*) E) |
     (if E1 E2 E3) | L | (let ((I E)*) E) |
     (letrec ((I E)*) E)
```

Type saftey is required for automatic storage management (garbage collection); otherwise, we can't tell what's a pointer and what's a not.

25.1 Typing Rules

Typing rules tell us whether a particular program is well-typed. The typing rules for Scheme/R are basically just the rules given in [Explicit Types/Typing Rules]

```
            A[I1:T1, ..., In:Tn] ⊢ Eb : Tb
      ---------------------------------------- [lambda]
   A ⊢ (lambda (I1 In) Eb) : (→ (T1 .. Tn) Tb)
```

How do we come up with the types of the arguments?

Any well-typed rule won't give you the class of errors discussed above.

25.2 Unification

Define substitution functions:

```
TypeVar = ?v1, ?v2, ...
TypeLit = IntType, BoolType, ...
t ∈ Type = TypeVar + TypeLit
S ∈ Substitution = Type → Type
```

Define a unify operator U. The unification operator takes two types, and tries to coerse them to be the same type. If this is possible, then return S extended with enough bindings to make the two types equivalant.

```
S' = U(t1, t2, S)
```

Therefore, if S'=U(t1, t2, S), then we should have:

```
S' t1 ≡ S' t2
S ⊑ S'
```

For example:

```
U((?x ?y), (?y int), ø)
      ⇒ ?x=int ?y=int
U[(→ (?x) (→ (?x) ?y)), (→ (int) ?z), ø]
      ⇒ ?x=int, ?z=(→ (int) ?y)
```

25.3 Reconstruction

Define a reconstruction function, R:

```
R⟦ E ⟧ = Bindings → Substitution → <Type, Substitution>
```

The type reconstruction function takes an expression, a set of bindings (from identifiers to types), and a set of substitutions. It then determines the type of that expression, assuming the given bindings and substitutions.

```
R⟦ E ⟧ A S0 = <T,S>
```

A type reconstruction function R is sound iff:

```
R⟦ E ⟧ A S0 = <T,S> ⇒ S A ⊢ E : S T
```

A type reconstruction function R is complete iff:

```
R⟦ E ⟧ A S0 = <T,S> ⇐ S A ⊢ E : S T
```

Reconstruction Algorithm:

```
R⟦ #u ⟧ A S = <unit, S>

R⟦ I ⟧ A[I:T] S = <T, S>

R⟦ (if E1 E2 E3) ⟧ A S =
  (let* ((<T1,S1> (R⟦ E1 ⟧ A S))   ; Get type of E1.
         (S1' U(T1,bool,S1))       ; Force T1 to be bool
         (<T2,S2> (R⟦ E2 ⟧ A S1')) ; Get type of E2.
         (<T3,S3> (R⟦ E3 ⟧ A S2 )) ; Get type of E3.
         (S3' U(T2, T3, S3)))
      <T2, S3'>)

R⟦ (lambda (I1 ... In) E) ⟧ A S =
   R⟦ E ⟧ A[I1=?v1, ..., In=?vn] S

R⟦ (E0 E1 ... En) ⟧ A S =
  (let* ((<T0, S0> (R⟦ E0 ⟧ A S))
         (<T1, S1> (R⟦ E1 ⟧ A S0))
         ...
         (<Tn, Sn> (R⟦ En ⟧ A Sn-1))
         (T U(T0, (→ (T1 ... Tn) ?vout))))
      <?vout, Sn>)
```

Wednesday, October 27, 1999

26 Type Reconstruction

26.1 Unification

Unification is a partial function (model errors as times when we try to apply the parial function to something that it doesn't have a value for).

```
U : Type → Type → Subst → Subst
```

```
U(T1, T2, S) = U(T2,T1,S)
U(t, t, S) = S
U(?t, t, ∅) = {?t1=t}
U(?t, t, S) = [U((S ?t), t, ∅)+S]
U(?t1, ?t2, S) = S + {?t1=?t2}
U( (→ (T1) T3), (→ (T2) T3), S)
   = U(T1,T2,S) + U(T3,T4,S)
```

Reconstruct:

```
R⟦ (lambda (x y) (if x y 0)) ⟧ ∅ ∅
let <Tbody, Sbody> = R⟦(if x y 0)⟧ [x:?v1, y:?v2] ∅
R⟦(if x y 0)⟧ [x:?v1, y:?v2] ∅ ⇒
   <Ttest, Tbody> = R⟦ x⟧ [x:?v1, y:?v2] ∅
   Stest' = U(Ttest, bool, ∅)
   R⟦ y⟧ A {?v1=bool} = <?v2, {?v1=bool}>
   R⟦0⟧ A {?v1=bool} = <int, {?v1=bool}>
   U(?v2, int, {?v1=bool}) = {?v1=bool; ?v2=int}
  <(→ (?v1 ?v2) int), {?v1=bool, ?v2=int}>
```

Compose:

```
R⟦ (lambda (g x) (lambda (x) (g (f x)))) ⟧ ∅ ∅
= (((→ (a) b) (→ (c) a)) → (→ (c) b))
```

26.2 Substitution

Define a substitution as a function from type variables to (generic) types. Then if we want to apply the subst to a complex type, use the function complex-subst:

```
complex-subst⟦ S, primt⟧ = (S primt)
complex-subst⟦ S, t1→ t2⟧ = complex-subst⟦ S, t1⟧ → CS⟦ S, t2⟧
```

Consider

```
(lambda (id) (if (id #t) (id 1) (id 0)))
```

Thursday, October 28, 1999

27 Polymorphic Types

Consider the examples:

```
(λ (x) x)
(λ (f) (λ (g) (λ (x) (f (g x)))))
```

What do we do when we have underconstrained variables? For example, our type system will give (→ (?t) ?t) as the type for the identity function.

How can we prove that a polymorphic type system is sound and complete?

We defined let as:

```
A[I1=T1 ... In=Tn] ⊢ Eb:T  ;  A ⊢ Ei:Ti
   ---------------------------------    [mono-let]
 A ⊢ (let ((I1 E1) ... (In En)) Eb) : T
```

but consider the expression:

```
(let ((id (lambda (x) x)))
   (if (id #t) (id 1) (id 2)))
```

The identity function (id) must be given a single type. Whenever it is used, its type gets defined.. So we need a new typing rule.

```
            A ⊢ [Ei/Ii] Eb:T
   ---------------------------------     [poly-let]
 A ⊢ (let ((I1 E1) ... (In En)) Eb) : T
```

Here, we're replacing each identifier by its binding for type-checking, so each time an identifier is used, it can get its own binding..

We would prefer to not actually insert the text of Ei for every identifier.. Precompute its type scheme, and then use that whenever we want to check the type of a particular expression. So define type schemes (not a member of type):

```
TS := (generic (I*) T)
(generic (?t) (→ (?t) ?t))
```

Consider the following example:

```
(lambda (x)
  (let ((y x))
     (if y 1 x)))
```

We can't generalize pattern variables that are defined in the surrounding environment: they might get frozen later.

rewrite polylet as (semantically equivalant):

```
A[I1:Gen(T1,A) ... In:Gen(Tn,A) ⊢ Eb:T ; E⊢ Ei:Ti
   ------------------------------------------    [poly-let]
     A ⊢ (let ((I1 E1) ... (In En)) Eb) : T


Gen(T,A) = (generic (J1 ... Jn) T)
           {Ji} = FreeTypeVar(T) - FreeTypeEnv(A)


A[I:(generic (Ii In) T) ⊢ [Ti/Ii] T
```

```
RGen(T, A, S) = Gen((S T) (S A))
```

Define reconstruction clauses:

```
R⟦ I ⟧ A[I:(generic I1... In T)] S =
    <[?v1/I1] ... [?vn/In] T,S>

R⟦ (let ((I1 E1) ... (In En)) Eb) ⟧ A S =
   (let* ((<T1,S1> (R⟦ E1 ⟧ A S))
          (<Tn,Sn> (R⟦ En ⟧ A S_n-1)))
      R⟦ Eb ⟧ A[I1:Rgen(T1,A,Sn) ... In:Rgen(Tn,A,Sn)] Sn)
```

```
           A[I1:Ti... In:Tn] ⊢ Ei:Ti
    A[I1:gen(Ti A)... In:gen(Tn A)] ⊢ Eb:T
   ----------------------------------------  [poly-letrec]
   A ⊢ (letrec ((I1 E1) ... (In En)) Eb) : T
```

Thus, the new bindings in a letrec can be polymorphic in Eb.. But they can't be polymorphic when they call each other.

How do we do the reconstruction algorithm?

```
R⟦ (letrec ((I1 E1) ... (In En)) Eb) ⟧ A S =
   (let* ((A0 A[I1:?v1 I2:?v2 ... In:?vn])
          (<T1,S1> (R⟦ E1 ⟧ A0 S))
          (<Tn,Sn> (R⟦ En ⟧ A0 S_n-1))
          (Sfinal unify U[(?v1 ... ?vn), (T1 ... Tn), Sn]))
      R⟦ Eb ⟧ A[I1:Rgen(T1,A,Sfinal) ...
                In:Rgen(Tn,A,Sfinal)]
               Sfinal)
```

Consider the program:

```
(letrec ((x x))
  (if x 1 x))
```

This type-checks! The type of bottom is:

```
(generic (?v1) v1)
```

Consider:

```
(letrec ((id (lambda (x) x))
         (f (lambda (y) + y (id y))))
     ...)
```

What is the type of id? (→ (int) int)

Lists:

```
Null: (generic (t) (→ () (list-of t)))
Cons: (generic (t) (→ (t (list-of t)) → (list-of t)))
Car: (generic (t) (→ ((list-of t)) → t))
Cdr: (generic (t) (→ ((list-of t)) → (list-of t)))
```

Side effects:

```
Cell : (generic (t) (→ (t) (cell-of t)))
    : (generic (t) (→ ((cell-of t)) t))
:=   : (generic (t) (→ ((cell-of t) t) unit))
```

But there's a problem! Consider the code:

```
(let ((x (cell (null))))
  (begin (update x (cons 1 (null)))
         (if (car ( x)) 2 3)))
```

Just don't generalize expressions that have (immediate) side effects? Note that this doesn't include procedures with side effects inside them, because these will come later..

Tuesday, November 2, 1999

28 Polymorphism II

28.1 Scheme/XSP (explicity types, subtyping, polymorphism)

Consider map. It's polymorphic type is:

```
(generic (t t2) (→ ((→ (t)  t2) (list-of t)) (list-of t2)))
```

This won't type check:

```
(lambda (map)
   (map not? (map odd? ('1 2 3))))
```

But consider introducing explicity polymorphism with something like:

```
(lambda (map (poly (t1 t2)
                   (→ ((→ (t1) t2) (listof t1)) (listof t2))))
   ((proj map bool bool) not?
                         ((proj map int bool) odd?
                                              ('1 2 3))))
```

We explicitly project with (proj Igeneric I1... In), and explicitly make things polymorphic with (plambda (I1... In) E).

But these explicit plambdas and proj's make things look quite ugly:

```
(letrec
   ((map (plambda (t1 t2)
           (lambda ((f (→ (t1) t2)) (l (listof t1)))
              (if ((proj null? t1) l)
                  ((proj null t2))
                  ((proj cons t2) (f ((proj car t1) l))
                      (proj map t1 t2) f ((proj cdr t1) l)))))))
    ...)
```

Extensions:

```
E ::= ... | (plambda (I1... In) E) | (proj E T1... Tn)
T ::= ... | (poly (I1... In) T)
```

New Typing rules:

$$\frac{\texttt{A} \vdash \texttt{E:T} \qquad \forall\ \texttt{i}\ \ \texttt{Ii} \notin \texttt{FTV(FV(E))}}{\texttt{A} \vdash \texttt{(plambda (I1 ... In) E) : (poly (I1 ... In) T)}}$$

$$\frac{\texttt{A} \vdash \texttt{E : (poly (I1 ... In) T')} \qquad \texttt{T = [Ti/Ii]T'}}{\texttt{A} \vdash \texttt{(proj E T1 Tn) : T}}$$

$$\frac{\texttt{[Ii/Ji]T} \sqsubseteq \texttt{T'} \qquad \texttt{I'} \in \texttt{FrreeIds(T)}}{\texttt{(poly (I1 ... In) T)} \sqsubseteq \texttt{(poly (I1 ... In) T')}}$$

We need the "∀..." clause in plambda for a subtle reason. Consider:

```
((proj
  (plambda (int)
    (lambda ((int x))
      (+ x 1)))
  bool)
 #t)
```

```
FTV(x:int) = x
FTV(x:(poly (int) int)) = ∅
```

```
((proj
   (plambda (con)
     (lambda ((x (con int))
              (y (→ ((con int)) int))
          (y x))))
  listof)
 '(1 2 3) (proj car int))
```

Consider:

```
⊢ (lambda ((x listof)) x) : (→ (listof) listof)
```

Well-typed, but not useful.. It's impossible to call this procedure.

28.2 Abstract Types

We want to protect types that live together in a single address space from:

- other people looking inside a type's values
- other people modifying a type's values
- other people forging a type's values

We can use explicit polymorphism to create abstract types. Consider a polymorphic tree module:

```
M = (poly (t)
       (module (new (t) (tree t))
               (get ((tree t)) t)))
```

we want the client to know the type of m, but we don't want them to know the type of tree. i.e., we want the client to know the interface of m without knowing anything about the implementation.

we make the implementaiton accept part of the client as a parameter. then the implementation can play with it in its own space.. consider the type of the implementation:

```
Timpl = (poly (r) (→ ((poly (tree) (→ (M) r)) r)))
(define impl
   (plambda (r)
      (lambda ((c (poly (tree) (→ (m r)))))
          ((proj c listof) (module ...)))))
```

```
r is the result.
(poly (tree) (→ (M) r)) is a part of the client which
      maps from a module to a result.  Note that the impl
      of M is hidden by the poly(tree)..
```

```
(define client
  (lambda ((impl Timpl))
     ((proj impl r)
        (plambda (tree)
           (lambda ((mod M))
              (proj mod int) new 1)))))
```

Thursday, November 4, 1999

29 Pattern Matching (by desugaring)

```
E ::= ... | (match E C*)
C ::= (P E)
P ::= L |  | I | (I P*)
```

For every constructor, assume a deconstructor, whose name is the constructor name with a "˜" added to the end:

```
cons  = constructor
cons˜ = destructor
```

```
(define (cons˜ value success failure)
  (if (not (null? value))
      (success (car value) (cdr value))
      (failure value)))
cons˜ : (generic (t t2)
          (→ ( (listof t)
               (→ (t (listof t)) t2)
               (→ ((listof t)) t2))
             t2))
```

Note that we just test for null?, instead of calling pair?, because this type-checks.

Consider:

```
(match E
  ((cons 1 (cons x )) (+ 1 x))
  ( 2))
```

A first attempt at desugaring:

```
(let ((I1 E))      ;; Make sure we only evaluate E once.
  (cons˜ I1 (λ (I2 I3)
              (if (= I2 1)
                  (cons˜ I3 (λ (I4 I5) (let ((x I4)) (+ 1 x)))
                         (lambda (x) 2))
                  2))
          (lambda (x) 2)))
```

Try this:

```
(let ((I1 E) (I6 (lambda (x) 2)))
  (cons˜ I1 (λ (I2 I3)
              (if (= I2 1)
                  (cons˜ I3 (λ (I4 I5) (let ((x I4)) (+ 1 x)))
                         I6)
                  (I6 I2)))
          I6))                ;; (???)
```

29.1 Desugaring Function

```
D⟦ (match E ((P1 E1) ... (Pn En))) ⟧ =
   (let ((id E))
      expandclause(P1, ... Pn, E1, ... En, id, basefailure)))

expandclause(P1, ... Pn, E1, ... En, v, fail) =
  (if (= n 0)
      (fail v)     ;; No patterns left -- fail.
      (let ((id1 (λ (x)  ;; id1 is the failure cont.
                   expandclause(P2, ... Pn, E2, ... En, v, fail))))
         (expandexp (P1, v, E1, id1))))

expandexp(pat, v, E, fail)
           E
        lit  (if (lit-eq? L v) E (fail v))
          I  (let ((I v)) E)
(I p1 ... pn) (I~ v (λ (id1 ... idn) e') f)
        where e' = expandpat (p1, ... pn, id1, ..., idn, e, f)

expandpat(P1, ... Pn, id1, ... idn, s, fail) =
  (if (= n 0)
      s
      (expandexp (P1, id1, e', fail)
         where e'=expandpat(p2, ... pn, id2, ... idn, s, fail) =

basefailure = (lambda (x) (error "oh no!!"))
```

30 Abstract Types

```
E ::= ... | (module D* B*)
B ::= (I E)
D ::= (define-datatype Iabs V*) |
      (define-datatype (Iabs I1 ... In) V*)
V ::= (I T*)
```

```
(define-datatype sexp
  (unit→ sexp unit)
  (bool→ sexp bool)
  (int→ sexp int)
  (list→ sexp (list-of sexp)))
```

Evaluate in scheme/R. Alpha-rename all type names. Since types are reconstructed, we don't need to remember names of types?

```
A[D*, I1:Tn, ..., In:Tn] ⊢ Ei:Ti 1 ≤ i ≤ n
-----------------------------------------
A ⊢ (module D* (I1 E1) ... (In En)): (moduleof (I1 T1) ... (In Tn))

A[(define-datatype Iabs ... (In T1... Tn)...)]
   ⊢ In:(→ (T1... Tn) Iabs)

A[(define-datatype Iabs ... (In T1... Tn)...)]
   ⊢ In~:(→ (Iabs (→ (T1... Tn) T) (→ (Iabs) T)) T)
              ; T arbitrary
```

```
A[(define-datatype (Iabs I1 ... In) ... (Iv T1... Tn)...)]
   ⊢ Iv:[T'i/Ii](→ (T1... Tn) (Iabs I1... In)) ; T'i arbitrary
```

Tuesday, November 9, 1999

31 Concurrency

2 reasons to emply concurrency:

- performance
- simplicity

31.1 Fork/Join

```
E ::= ... | (fork E) | (join E) | (thread? E)
```

The following is more-or-less equivalant to E1:

```
(let ((t (fork E1)))
   (join t))
```

We can join a thread more than once: just get the value from the thread more than once. Thus, the value produced by the thread has to be kept around as long as any thread handlers that point to it are still accessible.

```
(define (parallel-map f l)
   (map (lambda (x) (join x))
        (map (lambda (x) (fork (f x))) l)))
```

What if f has side effects? Have to be careful...

```
T ∈ Thread-Handle = IntLit
A ∈ Agenda = Thread-Handle → Exp
Configuration = Agenda × Store
```

SOS:

```
          E ⇒ E'
-------------------------
<A[T=E],S> ⇒ <A[T=E'],S'>

<A[T=(fork E)],S> ⇒ <A[T=(*thread T'*),T'=E],S>   (T' new)

<A[T=(join (*thread* T')),T'=V],S> ⇒ <A[T=V,T'=V],S>

       <A[T=E],S> ⇒ <A[T=E'],S'>
---------------------------------------
<A[T=(join E)],S> ⇒ <A[T=(join E')],S'>
```

This won't work very well if we fork more than one increment! on the same cell:

```
(define (increment! (lambda (c)
  (begin (cell-set! c (+ 1 (cell-ref c)))
         (cell-ref c)))))
```

Consider:

```
(let ((c (cell 0)))
  (let ((t1 (fork (increment! c)))
        (t2 (fork (increment! c))))
    (+ (join t1) (join t2))
```

This might produce 2, 3, or 4.

31.2 Obtain! / Release!

Add a locking mechanism:

```
E ::= ... | (lock) | (obtain! E) | (release! E)
```

In general, if we have shared mutable data, we may need a lock.

Make some sugar to help avoid programming errors..

```
D⟦(monitor I E)⟧ = (let ((I (lock))) E)
D⟦(exclusive El Eb)⟧ =
   (let ((Il El))
      (obtain! Il)
      (let ((Iret Eb))
         (release! Il)
         Iret))
```

Rewrite the above example as:

```
(monitor c-lock
  (let ((c (cell 0)))
    (let ((t1 (fork (exclusive c-lock (increment! c))))
          (t2 (fork (exclusive c-lock (increment! c)))))
      (+ (join t1) (join t2))
```

SOS:

```
<A[T=(lock)],S> ⇒ <A[T=(*lock* L)], S[L=#f]>   L fresh
<A[T=(obtain! (*lock* L))], S[L=#f]> ⇒ <A[T=#u], S[L=#t]>
<A[T=(release! (*lock* L))], S[L=#t]> ⇒ <A[T=#u], S[L=#f]>
```

If we wanted to make sure that only the thread that locked something can release it, replace "S[L=#t]" with "S[L=T]"...

Deadlock fun!

Construct a wait-for diagram:

- If A holds L, draw an arrow from L to A.
- If A is waiting for L, draw an arrow from A to L.
- If this diagram has a cycle, then we're in deadlock.

Very hard to construct wait-for diagrams in distributed systems.

31.3 Condition Variables

```
E ::= ... | (condition) | (wait E) | (notify E)
```

Allows us to do something like:

```
while not P do
  wait C
```

Example:

```
(monitor count-up
  (let ((count (cell 0))
        (change (condition)))
    (module
       (increment (lambda ()
                    (exclusive count-up
                               (begin (:= count (+ 1 ( count)))
                                      (notify change)
                                      ( count)))))
       (wait-til-n (lambda ()
                     (if (>= (exclusive count-up ( count)) n)
                         #u
                         (begin (wait change)
                                (wait-til-n n)))))
```

But what if someone changes the value after we do the if condition. Then the if condition sees a value of 99, but then the notify happens before the wait occurs.. :(There are a couple ways to fix that.. E.g., move the exclusive outside the if, and redefine wait to atomically unlock count-up and go into wait state.

SOS:

```
CV = CV-State × CV-Queue
CV-Sate = {wakeup-waiting | no-wakeup}
CV-Queue = Thread-Handle*
```

```
<A[T=(condition),S> ⇒
      <A[T=(*condition* L)],S[L=<no-wakeup,[]>]>

<A[T=(wait (*condition* L)),S[L=<wakeup-waiting,[]>]> ⇒
      <A[T=#u],S[L=<no-wakeup,[]>]>

<A[T=(wait (*condition* L)),S[L=<no-wakeup,Q>]> ⇒
      <A[T=(wait (*condition* L))],S[L=<no-wakeup,T.Q>]>

<A[T=(notify (*condition* L)), S[L=<no-wakeup, [T1,...,Tn]>] ⇒
      <A[T=#u, T1=#u, ..., Tn=#u], S[L=<no-wakeup, []>]>

<A[T=(notify (*condition* L)), S[L=<no-wakeup, []>]>] ⇒
      <A[T=#u], S[L=<wakeup-waiting, []>]>
```

Wednesday, November 10, 1999

32 Abstract Datatypes

```
QI = (recordof (new (poly (t) (→ () (queueof t))))
               (add (poly (t) (→ (t (queueof t)) (queueof t))))
               (examine (poly (t) (→ (int (queueof t)) t))))
```

```
(define queue-client T
  (plambda (queueof)
     (lambda ((queue QI))
        ((proj queue.examine string) 1
             ((proj queue.add string) "skating"
                 ((proj queue.add string) "study"
                    ((proj queue.new string))))))))

T = (poly (queueof (→ (QI) string)))

((proj queue-impl string) queue-client)

queue-impl: T2 = (poly (t) (→ T t))
```

Now let's implement queue:

```
(define queue-impl T2
  (plambda (t)
    (lambda ((qclient (poly (queueof) (→ (QI) t))))
      ((proj qclient listof)
         (record (new null)
                 (add cons)
                 (examine my-list-ref))))))

Note that: null ≡ (poly (t) (proj null t))
```

Try an implementation more like:

```
QI' = (poly (t) (recordof (new (→ () (queueof t)))
                          (add (→ (t (queueof t)) (queueof t)))
                          (examine (→ (int (queueof t)) t))))

(define queue-client T
  (plambda (queueof)
     (lambda ((queue QI'))
        (let ((strqueue (proj queue string)))
          (queue.examine 1
               (queue.add "skating"
                   (queue.add "study"
                      (queue.new))))))))
```

Now let's implement queue':

```
(define queue-impl T2
  (plambda (t)
    (lambda ((qclient (poly (queueof) (→ (QI) t))))
      ((proj qclient listof)
         (plambda (t)
           (record (new (proj null t))
                   (add (proj cons t))
                   (examine (proj my-list-ref t))))))))

Note that: null ≡ (poly (t) (proj null t))
```

Tuesday, November 16, 1999

33 Effects & Types

Consider the program:

```
(let ((c (cell 0)))
   (:= c 2)
   ( c))
```

The type of c is:

```
c: (cellof int)
```

If we want to, we can add more type information. For example, we could give cells "colors" (regions):

```
c: (cellof int blue)
```

The color might, e.g., correspond to the region of the store that contains the cell. Then we can run 2 threads in parallel safely if they use different regions of the store. Also, we could try to figure out how long a cell lives by seeing when colors become unavailable.

```
T ::= ... | (cellof T R)
R ::= I
```

So each cell has a region. Can we put every cell in a different region? No, because types have to match.. E.g.:

```
(if p (cell 0) (cell 1)) : (cellof int I)
```

But we want to maximize diversity of colors.

Define three effects: (init R), (read R), (write R). There is an ACUI algebra "maxeff".. (define an identity element "pure").

Redefine procedures:

```
T ::= (→ (T1... Tn) F Tr)
```

Where "F" is the latent effect – it describes what the procedure will do when it's called. Define a type/effect operator:

```
A ⊢ E : T ! F
```

For example:

```
          A[I:Tin] ⊢ E : Tout ! F
-----------------------------------------------
A ⊢ (lambda (I) E) : (→ (Tin) F Tout) ! pure


    A ⊢ Ep : (→ (T1) Flatent Tout) ! Fp
               A ⊢ E1 : T1 ! F1
-----------------------------------------------
A ⊢ (Ep E1) : Tout ! (maxeff F1 Fp Flatent)
```

Consider the type of:

```
E1 = (lambda (c)
       (:= c (+ 1 ( c)))
       ( c))


E1 : (→ ((cellof int r)) (maxeff (read r) (write r)) int)
```

If we generalize this, then we can use it on any region. If we don't generalize it (e.g., if we pass it as an argument), then we just force every thing that the procedure is called on to be in the same region.

```
(lambda (f x) (f x)) : (→ ((→ (t1) f t2) t1) f t2)
```

Thus we can get polymorphism in effects!

Add the rule:

```
A ⊢ E : T ! F'   ;   F' ≤ F
---------------------------
      A ⊢ E : T ! F
```

But now we can't reconstruct very easily. We have to use constraint propagation to do reconstruction. Define reconstruction algorithm Z.

```
Z⟦ E ⟧ A S = <T, F, S', C>
```

```
C ::= (I F)*  ; map each effect I to a minimum effect F.
TS ::= (generic ((I D)*) T C)         ; type scheme
D ::= type | effect | region
```

```
increment : (generic ((r region) (e effect))
                     (→ (cellof int r) e int)
                     ((e ≥ (read r)) (e ≥ (write r))))
```

Since all effects are variables, we can unify two effects simply by setting them both to their union.

Consider reconstructing lambdas:

```
Z⟦ (lambda (I1 ... In) Ebody) ⟧ A S =
 let <Tbody, Fbody, Sbody, Cbody> be
  Z⟦ Ebody ⟧ A[I1:?v1, ..., In:?vn] S in
   <(→ (?v1 ... ?vn) ?e Tbody), pure, Sbody, (≥ ?e Fbody)+Cbody>
```

Reconstructing lets:

```
Z⟦ (let ((I1 E1) ... (In En)) Eb) ⟧ A S =
 let <T1,F1,S1,C1> = Z⟦ E1⟧ A S in
  ...
   let <Tn, Fn, Sn, Cn> = Z⟦ En⟧ A Sn-1 in
    Z⟦ Eb ⟧ A[I1:ZGen(T1,A,Sn,C1+...+Cn) ...
              In:ZGen(Tn,A,Sn,C1+...+Cn)] Sn
```

ZGen:

```
ZGen(T, A, S, C) = (generic (I1... In) T C)
  where {I1... In} = FV(S T) + FV(S C) - FTV (S A)
```

Consider cwcc:

```
cwcc : (→ ((→ ((→ (t1) t2)) t1)) t1)
```

Now try adding effects! Whee! Add two new effects: goto and comefrom.

```
cwcc : (→ ((→ ((→ (t1) (goto r) t2))
                f t1)) (maxeff f (comefrom r)) t1)
```

If we have an expression E, with (comefrom r) and/or (goto r) in it, and r isn't in the types of the FV's of T or in the output type of E, then we can ignore r.. This is true because the region can't exit/enter the region...

When can we free cells? Once their region goes away..

Wednesday, November 17, 1999

34 Effects

Some examples:

```
(lambda (c) ( c)) : (→ ((cellof t r)) (read r) t) ! pure

E1= (let ((c (cell 1)))                 ; This cell is r3
         (if ( (cell #t))              ; This cell is r2
            (cell 0)
            (begin (:= c 5) (cell 1)))) ; These cells are r1

E1: (cell-of int r1) ! (init r1), (init r2), (read r2),
                       (init r3), (write r3)
```

Clearly, r2 can be thrown away once we leave the if statement, since there's no longer any way to get to it.

```
Z : Exp → Type-Env → Subst → <T, F, S, C>
```

Each expression tends to inherit most of the constraints of its sub-expressions. In the end, find the minimum solution of the constraints.

Thursday, November 18, 1999

35 RPC

Remote procedure call: let a procedure on one computer call a procedure on another. Server needs an interface. Use a client stub and a server stub to handle the actual communications. Given an interface, we should be able to automatically implement stubs, such that p() in client stub sends a message to server stub that tells it to actually call p(). Send the args, too.. Then server p() returns answer, and server stub uses it to return a value etc. whee.

Define an interface description language (IDL). Write ain interface in IDL, and it will give us the server & cleint stubs.

Bind RPC modules as such:

```
(let ((m (open-remote "rpc://lcs.mit.edu/foo")))
     (m.p 1 2))
```

Differences between RPC and normal calls:

- limitations on acceptable data values (e.g., hard to use cells, procedures, thread ids, exceptions)... But most of these can be made to work with an advanced RPC scheme..
- communication failures – impossible to tell the difference between network errors and server errors. Give three different semantics to procedure calls:
  - at least once (idempotent)
  - at most once: server throws away packets with the same identifier. It needs to remember packets that it saw before it crashed. It's ok to complete something partially.
  - exactly once: everything is atomic. Action is done exactly once or not at all.

36 Applet Saftey

Checks we can do on applets:

- Run-time checks
- Types
- Control Environment Access (control applet's namespace)
- Hardware Protection
- Software fault isolation: rewrite object code such that all the stores are checked when the program is run to make sure they don't do anything bad. (5-30% slowdown)

Give each applet a principle (signiture) which gets fed into a policy engine to select what types of thing the applet can do... E.g., if the applet's principle is a big company, we might give it more access than if its principle is some random guy.. Protect file systems and network.

36.1 Simple model

Label each proc as local or remote. If we access disk or network, look up call stack.. if any remote procs, deny.. (allow network connections back to the applet's source sometimes).

36.2 Capability Model

Give applets capabilities: unforgable tokens that permit access. Problems: need to rewrite applets; also, confinement: how do we keep capabilities from being coppied?

36.3 Extended Stack Introspection

- enablePriviledge(target)
- disablePriviledge(target)
- checkPriviledge(target)

Target can be a string like "network access.." Whenever you're about to do something, check that we have that priviledge. An applet is allowed to enable a priviledge if its principle is enabled for that priviledge.

```
checkPriveledge(Target target) {
   foreach stackFrame {
       if stackFrame is enabled for target return ok
       if policyEngine disallows target for principle
          then return Failed
   }
   return(default)
}
```

36.4 ELFS whee

Create a log of operations and undo operations.. So you can get rid of it if you want. Also, you can audit your operations, e.g., to see who has seen certain files..

Tuesday, November 23, 1999

37 Pragmatics

We will compile a subset of scheme:

```
P ::= (program (define I E)* E)
E ::= L | I | (lambda (I*) E) | (call E E*) |
      (primop P E*) | (let ((I E)*) E) | (if E E E) |
      (set! I E) | (letrec ((I E)*) E)

Sugar: (primop P E*) = (%P E*)
```

37.1 Steps of compilation:

- Desugar
- Globalize (eliminate global variables)
- Assignment Convert (get rid of set!)
- CPS Convert
- Closeure Convert
- Lambda Lifting
- Data Conversion
- Code Gen (turn into register machine code)

38 Desugaring

How should we deal with letrec? Fixed-point operator is inefficient. Use the following instead:

```
D⟦ (letrec ((I1 E1) ... (In En)) Eb) ⟧ =
  D⟦ (let ((I1 #f) ... (In #f))
       (set! I1 E1)
       ...
       (set! In En)
       Eb) ⟧
```

But this only works for procedures: if evaluating any E1... En references some other E1... En, then it will get the wrong value. So run a syntactic checker to make sure letrec gets procedures.

Program desugars into a letrec..

Let desugars into procedure call..

39 Globalizing

(c.f. linking with a standard library)

39.1 Inlining

One way: textually substitute primops into the program.

But set! lets us change the meanings of things like +

39.2 wrapping

Alternative: wrap the whole program in a (let ...) that defines each identifier.

39.3 combined approach

Inline whenever you can, and wrap only when necessary. We can't inline something if it's set!'ed. Also, we can't pass primops as procedures, so enclose them in lambdas if they're passed to funcs.

40 Assignment Convertion

```
A⟦ I ⟧ = (%cell-ref I)

A⟦ (lambda (I1 ... In) E) ⟧ =
   (lambda (I1 ... In)
       (let ((I1 (%cell I1)) ... (In (%cell In)))
          A⟦ E⟧))

A⟦ (set! I E) ⟧ = (cell-set!% I A⟦ E ⟧)
```

But we don't need to convert things that don't get set!'ed. So do a syntactic check to make sure..

41 CPS Convertion (Continuation Passing Style)

Transform a program so that procedures never return (don't need a call stack). Also, transform it so there are no nested expressions. This allows us to perform the same optimizations on both the user's data/code and the system's data/code.

41.1 A Simple CPS Converter

Use continuations to model control flow, and make it explicit.

Every CPS-converted expression will give a procedure that takes a continuation.

```
CPS⟦ I ⟧ = (lambda(k) (call k I))

CPS⟦ L ⟧ = (lambda(k) (call k L))

CPS⟦ (call Ep E1 ... En) ⟧ =
  (lambda(k) (call CPS⟦ Ep ⟧ (lambda (vp)
                (call CPS⟦ E2 ⟧ (lambda (v1)
                   ...
                    (call CPS⟦ En ⟧ (lambda (vn)
                   (call vp v1 ... vn k))))))

CPS⟦ (lambda (I1 ... In) E) ⟧ =
  (lambda (k)
     (call k (lambda (I1 ... In k')
                (call CPS⟦ E ⟧ k'))))

CPS⟦ (if E1 E2 E3) ⟧ =
  (lambda (k)
     (call CPS⟦ E1 ⟧
        (lambda (v1)
          (if v1 (call CPS⟦ E2 ⟧ k)
                 (call CPS⟦ E3 ⟧ k)))))

CPS⟦ (primop P E) ⟧ =
   (lambda (k) (call CPS⟦ E ⟧ (lambda (v1)
                  (let ((v2 (primop P v1))) (call k v2)))))
```

CPS outputs expressions of the form:

```
Ecps ::= (call V V*) |
         (let ((I W)*) Ecps) |
         (if V Ecps1 Ecps2)
W ::= V | (primop P V*)
V ::= L | I | (lambda (I*) Ecps)
```

But CPS produces a -lot- of code... Expecially a lot of lambdas. For example:

```
CPS⟦ (%odd? 1) ⟧ =
(lambda (k) (call (lambda (k') (call k' 1))
                  (lambda (x) (call k (%odd? x)))))
```

We would prefer something like:

```
(lambda (k) (call k (%odd? x)))
```

We can do that with lambda-reduction.

Consider CPS converting a let:

```
CPS⟦ (let ((I Ed)) Eb) ⟧ =
  (lambda (k) (call CPS⟦ Ed ⟧ (lambda (vd)
                 (let ((I vd)) (call CPS⟦ Eb ⟧ k)))))
```

41.2 Meta-CPS (A smarter CPS converter)

```
MCPS: Exp → Meta-Continuation → Exp
Meta-Continuation: Exp → Exp
```

Top level meta-continuation:

```
[λ V . (call *top* V)]
```

Where the square brackets indicate application at the meta-level...

```
[MCPS⟦ 1 ⟧ [λ V.(call *top* V)]] = (call *top* 1)
```

We can convert a meta-continuation into a real continuation by using:

```
[exp→ meta-cont k] = [λ V.(call k V)]
```

We can convert a real continuation into a meta-continuation by using:

```
[meta-cont→ exp m] = (lambda (t) [m t])
```

But this gives us the following, which is bad for tail calls:

```
[meta-cont→ exp [exp→ meta-cont k]] = (lambda (t) (call k t))
```

We want instead to have:

```
[meta-cont→ exp [exp→ meta-cont k]] = k
```

so just define it as a special case for meta-cont→ exp..

```
MCPS⟦ L ⟧ = [λ m . [m L]]

MCPS⟦ I ⟧ = [λ m . [m I]]

MCPS⟦ (primop P E1 E2) ⟧ =
   [λ m.[MCPS⟦ E1⟧ [λ V1.
          [MCPS⟦ E2⟧ [λ V2.
             (let ((temp (primop P V1 V2))) [m temp])]]]]]

MCPS⟦ (lambda (I1... In) Eb) ⟧ =
   [λ m. [m (lambda (I1... In k)
             [MCPS⟦ Eb⟧ [exp→ meta-cont k]])]]

MCPS⟦ (call E1 E2) ⟧ =
   [λ m. [MCPS⟦ E1⟧ [λ V1.
           [MCPS⟦ E2⟧ [λ V2.
              (call V1 V2 [meta-cont→ exp m])]]]]]

MCPS⟦ (if E1 E2 E3) ⟧ =
   [λ m. [MCPS⟦ E1⟧ [λ V1.
           (let ((k [meta-cont→ exp m]))
              (if V1
                 [MCPS⟦ E2⟧ [exp→ meta-cont k]]
                 [MCPS⟦ E3⟧ [exp→ meta-cont k]]))]]]
```

An example:

```
MCPS⟦(%+ (%* (%- 3 1) (%/ 9 3)) (%- 10 6))⟧ [λ V.(call *top* V)]
 = (let* ((t1 (%- 3 1))
          (t2 (%/ 9 3))
          (t3 (%* t1 t2))
          (t4 (%- 10 6))
          (t5 (%+ t3 t4)))
     (call *top* t5))
```

Another example:

```
MCPS⟦(%+ z (if b (%+ x 1) (%- x 1)))⟧ [λ V.(call *top* V)] =
  (let ((k (lambda (t1)
             (let ((t2 (%+ z t1))) (call *top* t2)))))
     (if b (let ((t3 (%+ x 1))) (call k t3))
           (let ((t4 (%- x 1))) (call k t4))))
```

Yet another example:

```
(begin (define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))
       (fact 3))
```

Desugar/globalize/assignment conversion gives us:

```
(let ((fact (%cell #u)))
   (%cell-set! fact (lambda ...))
   (call (%cell-ref fact) 3))
```

Consider how to MCPS the lambda that defines fact:

```
(lambda (n k2)
  (let ((t3 (%= n 0))
     (if t3
         (call k2 1)
         (let ((t4 (%cell-ref fact)))
            (let ((t5 (%- n 1)))
                (call t4 t5 (lambda (t6)
                              (let ((t7 (%* t6 n)))
                                 (call k2 t7)))))))))))
```

Wednesday, December 1, 1999

41.3 MCPS conversion, continued

A meta-continuation wants a specific type of expression as its input. In particular, a metacontinuation is like an expression with a hole.. If you put the wrong thing in that hole, it won't work. An expression of a metacontinuation is:

```
my_mcont = (lambda (exp) `(primop + 5 (primop + ,exp)))
```

In particular, the argument to a metacont must be:

- a literal (including a lambda)
- an identifier

```
(define (mcps exp mcont)
   (cond ((id? exp)   (mcps-id exp mcont))
         ((lit? exp)  (mcps-lit exp mcont))
         ((let? exp)  (mcps-let exp mcont))
         ...))
```

```
(define (mcps-id id mcont) (mcont id))
```

```
(define (mcps-lit lit mcont) (mcont lit))
```

```
(define (mcps-if exp mcont)
   (let ((test (cadr exp))
         (con (caddr exp))
         (alt (cadddr exp)))
      (mcps test
            (λ (simp) `(let ((k (λ (t) ,(mcont 't))))
                         (if ,simp
                             ,(mcps con
                                    (lambda (simp)
                                       '(call k ,simp)))
                             ,(mcps alt
                                    (lambda (simp)
                                       '(call k ,simp)))))))))
```

Try mcps'ing a let with one binding:

```
(define (mcps-let exp mcont)
   (let ((var (let-var exp))
         (binding (let-binding exp))
         (body (let-body exp)))
      (mcps binding
            (λ (s)
               `(let ((,var ,s))
                   ,(mcps body mcont))))))
```

Try mcps'ing cwcc! :)

```
(define (mcps-cwcc exp mcont)
   (let ((mcps-proc (cadr exp)))
     `(let ((exit (lambda (t trash) ,(mcont 't))))
         ,(mcps `(call ,mcps-proc exit)
                (lambda (s) `(call exit ,s 'trash))))))
```

Thursday, December 2, 1999

42 Closure Conversion

We want to get rid of all the free variables in sub-expressions. For example, consider:

```
(let ((p1 (lambda (n)
            (lambda (k) (+ n k)))))
   (let ((p2 (p1 7)))
      (p2 p3)))
```

Assume we have closure primops, which basically construct and reference tuples:

```
(%closure Eλ E1 ... En)   ; make a new closure
(%closure-ref Ec n)       ; nth elt of Ec
```

Let each procedure take its own closure as a variable.

```
(let ((p1 (%closure (lambda (.c1. n))
             (%closure (lambda (.c2. k)
                         (+ (%closure-ref .c2. 1) k))
                       n))))
    (let ((p2 (call-closure p1 7)))
       (call-closure p2 p3)))

Where (call-closure Ec E1 ... En) =
         (let ((Itemp Ec))
            (call (%closure-ref Itemp 0)
                  Itemp E1 ... En))
```

42.1 Design Choices in Closure Conversion

Nested vs. flat closures

```
(lambda (a b)
  (lambda (c d)
    (lambda (e f)
       (a c e))))
```

Closure might be:

```
<λ3, a, c>       ; ← flat
<λ3, ptr, c>
  where ptr points to the closure <λ2, a>
```

Doing set! on a flat closure would be a problem, except that we've already done assignment conversion!

Most real compilers use nested frames, and put them on the stack.

43 Lambda Lifting

```
(let ((p1 (%closure (lambda (.c1. n))
             (%closure (lambda (.c2. k)
                         (+ (%closure-ref .c2. 1) k))
                       n))))
    (let ((p2 (call-closure p1 7)))
       (call-closure p2 p3)))
→
(program
  (define .l1. (lambda (.cl1. n) (%closure .l2. n)))
  (define .l2. (lambda (.cl2. k) (+ (%closure-ref .c2. 1) k)))
  (let (p1 (%closure .l1.)))
```

So now we have a program of the form:

```
(program (define .l1. ...)
         ...
         (define .ln. ...)
  Ecps)
```

Convert each .li. into code blocks. Then the lambdas are all basically labels, and we can branch between them..

After closure conversion, lambda lifting, and cps conversion... we have something pretty simple.

44 Data Conversion

In order to allow GC to work, we need to tag data items etc. Assume a 32 bit words. Allocate low 2 bits for a tag:

- 00 = int
- 01 = pointer
- 10 = immediate
- 11 = block header

Register words should never end in 11.

Intermediate values:

- ...<b>0010 = boolean (b=0 is false, b=1 is true)
- ...0110 = nil
- ...1010 = unspecified
- ...<c>1110 = character c.

Block headers: <size, type, 11> where type is:

- 0000 = cell
- 0001 = pair
- 0010 = vector
- 0011 = string
- 0100 = closure
- 0101 = code

Tuesday, December 7, 1999

45 Garbage Collection

(Lecturer: Olin Shivers)

GC is dynamic, types are static. Try to do static garbage collection: figure out when we will know that memory will never be re-used. Then we can just de-allocate it at that point, & avoid gc costs.. Static GC can't be complete, because sometimes you can't construct the appropriate proofs..

45.1 3 types of dynamic GC:

- stop & copy
- mark & sweep
- ref count

stop & copy

Basically equivalant to breadth first search of data structures.. You could try alternates like DFS, which will tend to give you better locality. But it has somewhat higher constants, so it's not used very much.

Comparisons

- allocation is cheaper in S&C than mark&sweep
- locality of newly allocated data is better in S&C than M&S
- S&C does better with large memories, since it only has to deal with the good stuff.
- In S&C, if we don't know if something is an int or a pointer, then we can't tell what to do: if we treat it as a ptr, we'll change the int; if we treat it as an int, we may get dangling pointer problems. But in M&S we can just guess things are pointers, and since we don't move them, we'll be safe. So M&S can be good in C.
- Ref counting has no long pauses
- Ref counting is timely: you get memory back as soon as it gets freed.
- Ref counting has trouple with circular data structures.
- Ref counting is slow

45.2 Variants of GC

Generational GC

Make some observations to help us figure out how to optimize gc:

- young things die frequently. So we can divide our allocation space into two generations: young generation is small, old is large. Do a "minor collection" where you gc the small space and move the live stuff into the old generation. But what do we do when old generation points into new generation?
- young things usually point to old things. So we can make the old/young generation thing work by keeping track of new pointers from old generation into young generation, and add that to the root set.

Real-Time Concurrent GC

- GC introduces bounded pauses.

- GC runs independantly of other processes.

Break from-space and to-space into pages. Maintain the invariant that the registers always point into to-space. Ever page in to-space has a bit that indicates whether everything has been copied to to-space yet. Protect the pages where not everything points into to-space (for reads). If we trap on a read of one of those pages, then run gc for that page. (But traps are really slow so people don't do this)

45.3 Atmoic Allocs

Allocations need to be atomic. But standard methods add a lot of overhead. Different methods to give light-weight atomicity to allocaiton:

- forwards: We could declare that we'll never get interrupts in the middle of basic blocks (blocks of code where you go streight through). Handle hardware interrupts at the end of basic blocks, and check at the beginning of each basic block that we have enough free memory. Alternative: to do allocs, jump to alloc code. If we're in that code, wait to do interrupts until we're done with the alloc code.
- backwards: Forwards allocation doesn't deal well with page faults. Page faults can make basic blocks take a long long time to finish. Another idea: if we wait to increment the fp until the very end of the alloc, then the "fp+=8" or whatever is an atomic point. So if we get interrupted in the middle of the alloc, then check when we return from the interrupt.. If we see that we're in the middle of an alloc, then restart the alloc. But painful to port and high overhead etc.
- backwards II: Use the low bits of the fp for lock bits. The last instruction (fp+=7) simulatneously does commit of the alloc and free-ing of the lock.
- sideways: give each thread its own allocation pool.. Fragmentation and wasted memory.

Recitation

Wednesday, December 8, 1999

46 Review

46.1 Content of the class

- Dynamic Semantics: how can you specify what a program means?
  - Operational Semantics
  - Denotational Semantics
    - Fixed points
    - "Standard" semantics and control
    - Translators
    - Static Semantics
    - Type Systems
    - Monomorphic
    - Subtypes
    - Polymorphic
    - Type reconstruction
    - Abstract types & Pattern matching (playing with types)
    - Effect Systems & Effect Reconstruction
    - Pragmatics
    - Compiler
    - Runtime
    - Data Representations
    - Garbage Collection

The end.