CSEP505: Programming Languages

Lecture 5: continuations, types

Dan Grossman

Spring 2006

Remember our symbol-pile

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e$

Values: $v ::= \lambda x.\ e$

$$\frac{}{\lambda x.\ e \Downarrow \lambda x.\ e}\ [\text{lam}] \qquad \frac{e1 \Downarrow \lambda x.\ e3 \quad e2 \Downarrow v2 \quad e3\{v2/x\} \Downarrow v}{e1\ e2 \Downarrow v}\ [\text{app}]$$

And where we were

- Go back to math metalanguage
  - Notes on concrete syntax (relates to Caml)
  - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- Define substitution precisely
  - And revisit function equivalences
- Environments
- Small-step
- Define and motivate *continuations*
  - (very fancy language feature)

Small-step CBV

- Left-to-right small-step judgment e → e’

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x\,.\,e)\ v \rightarrow e\{v/x\}}$$

- Need an “outer loop” as usual: e →* e’
  - * means “0 or more steps”
  - Don’t usually bother writing rules, but they’re easy:

$$\frac{}{e \rightarrow^{*} e} \qquad \frac{e1 \rightarrow e2 \qquad e2 \rightarrow^{*} e3}{e1 \rightarrow^{*} e3}$$

In Caml

```
type exp =
  V of string | L of string*exp | A of exp * exp
let subst e1_with e2_for s = …

let rec interp_one e =
  match e with
   V _ -> failwith “interp_one”(*unbound var*)
  | L _ -> failwith “interp_one”(*already done*)
  | A(L(s1,e1),L(s2,e2)) -> subst e1 L(s2,e2) s1
  | A(L(s1,e1),e2) -> A(L(s1,e1),interp_one e2)
  | A(e1,e2) -> A(interp_one e1, e2)

let rec interp_small e =
  match e with
   V _ -> failwith “interp_small”(*unbound var*)
  | L _ -> e
  | A(e1,e2) -> interp_small (interp_one e)
```

Unrealistic, but…

- Can distinguish infinite-loops from stuck programs
- It’s closer to a *contextual semantics* that can define continuations
- And can be made efficient by “keeping track of where you are” and using environments
  - Basic idea first in the SECD machine [Landin 1960]!
  - Trivial to implement in assembly plus malloc!
  - Even with continuations

Redivision of labor

```
type ectxt = Hole
           | Left of ectxt * exp
           | Right of exp * ectxt (*exp a value*)
let rec split e =
  match e with
   A(L(s1,e1),L(s2,e2)) -> (Hole,e)
 | A(L(s1,e1),e2) -> let (ctx2,e3) = split e2 in
                     (Right(L(s1,e1),ctx2), e3)
 | A(e1,e2)       -> let (ctx2,e3) = split e1 in
                     (Left(ctx2,e2), e3)
 | _ -> failwith "bad args to split"
let rec fill (ctx,e) = (* plug the hole *)
  match ctx with
   Hole           -> e
 | Left(ctx2,e2)  -> A(fill (ctx2,e), e2)
 | Right(e2,ctx2) -> A(e2, fill (ctx2,e))
```

So what?

- Haven't done much yet: `e = fill(split e)`
- But we can write interp_small with them
  - Shows a step has three parts: split, subst, fill

```
let rec interp_small e =
  match e with
   V _ -> failwith "interp_small"(*unbound var*)
  | L _ -> e
  | _   ->
    match split e with
     (ctx, A(L(s3,e3),v)) ->
        interp_small(fill(ctx, subst e3 v s3))
   | _ -> failwith "bad split"
```

Again, so what?

- Well, now we “have our hands” on a context
  - Could save and restore them
  - (like hw2 with heaps, but this is the control stack)
  - It’s easy given this semantics!

- Sufficient for:
  - Exceptions
  - Cooperative threads
  - Coroutines
  - “Time travel” with stacks

Language w/ continuations

- Now 2 kinds of values, but use L for both
  - Could instead have 2 kinds of application + errors
- New kind stores a context (that can be restored)
- Letcc gets the current context

```
type exp = (* change: 2 kinds of L + Letcc *)
  V of string | L of string*body | A of exp * exp
 | Letcc of string * exp
and body = Exp of exp | Ctxt of ectxt
and ectxt = Hole (* no change *)
          | Left of ectxt * exp
          | Right of exp * ectxt
```

Split with Letcc

- Old: active expression (thing in the hole) always some

  `A(L(s1,e1),L(s2,e2))`
- New: could also be some **`Letcc(s1,e1)`**

```
let rec split e = (* change: one new case *)
  match e with
   Letcc(s1,e1) -> (Hole,e) (* new *)
 | A(L(s1,e1),L(s2,e2)) -> (Hole,e)
 | A(L(s1,e1),e2) -> let (ctx2,e3) = split e2 in
                     (Right(L(s1,e1),ctx2), e3)
 | A(e1,e2)       -> let (ctx2,e3) = split e1 in
                     (Left(ctx2,e2), e3)
 | _ -> failwith “bad args to split”
let rec fill (ctx,e) = … (* no change *)
```

All the action

- `Letcc` becomes an L that “grabs the current context”
- `A` where body is a `Ctxt` “ignores current context”

```
let rec interp_small e =
  match e with
   V _ -> failwith “interp_small”(*unbound var*)
 | L _ -> e
 | _ -> match split e with
    (ctx, A(L(s3,Exp e3),v)) ->
       interp_small(fill(ctx, subst e3 v s3))
   |(ctx, Letcc(s3,e3)) ->
     interp_small(fill(ctx,
        subst e3 (L("",Ctxt ctx)) s3))(*woah!!!*)
   |(ctx, A(L(s3,Ctxt c3),v)) ->
       interp_small(fill(c3, v)) (*woah!!!*)
   | _ -> failwith “bad split”
```

Examples

- Continuations for exceptions is “easy”
  - Letcc for try, Apply for raise
- Coroutines can yield to each other (example: CGI!)
  - Pass around a yield function that takes an argument – “how to restart me”
  - Body of yield applies the “old how to restart me” passing the “new how to restart me”
- Can generalize to cooperative thread-scheduling
- With mutation can really do strange stuff
  - The “goto of functional programming”

A lower-level view

- If you’re confused, think call-stacks
  - What if YFL had these operations:
    - Store current stack in x (cf. Letcc)
    - Replace current stack with stack in x
  - You need to “fill the stack’s hole” with something different or you’ll have an infinite loop

- Compiling Letcc
  - Can actually copy stacks (expensive)
  - Or can avoid stacks (put frames in heap)
    - Just share and rely on garbage collection

Where are we

Finished major parts of the course

- Functional programming (ongoing)
- IMP, loops, modeling mutation
- Lambda-calculus, modeling functions
- Formal semantics
- Contexts, continuations

Moral? Precise definitions of rich languages is difficult but elegant

Major new topic: Types!

  – Continue using lambda-calculus as our model

Types Intro

Naïve thought: More powerful PL is better

- Be Turing Complete
- Have really flexible things (lambda, continuations, …)
- Have conveniences to keep programs short

By this metric, types are a step backward

  - Whole point is to allow fewer programs
  - A “filter” between parse and compile/interp
  - Why a great idea?

Why types

1. Catch “stupid mistakes” early
    - 3 + “hello”
    - print_string (String.append “hi”)
    - But may be too early (code not used, …)
2. Prevent getting stuck / going haywire
    - *Know* evaluation cannot ever get to the point where the next step “makes no sense”
    - Alternate: language makes everything make sense (e.g., ClassCastException)
    - Alternate: language can do whatever ?!

Digression/sermon

Unsafe languages have operations where under some situations the implementation “can do anything”

IMP with unsafe C arrays has this rule (any H’;s’!):

$$\frac{H;e1 \Downarrow \{v1,\ldots,vn\} \qquad H;e2 \Downarrow i \qquad i > n}{H;\ e1[i]=e2 \Downarrow H';s'}$$

Abstraction, modularity, encapsulation are impossible because one bad line can have arbitrary global effect

An engineering disaster (cf. civil engineering)

Why types, continued

3. Enforce a strong interface (via an abstract type)
    - Clients can’t break invariants
    - Clients can’t assume an implementation
    - Assumes safety
4. Allow faster implementations
    - Compiler knows run-time type-checks unneeded
    - Compiler knows program cannot detect specialization/optimization
5. Static overloading (e.g., with +)
    - Not so interesting
    - Late-binding very interesting (come back to this)

Why types, continued

6. Novel uses
   - A powerful way to think about many conservative program analyses/restrictions
   - Examples: race-conditions, manual memory management, security leaks, …
   - I do some of this; “a types person”

We’ll focus on safety and strong interfaces

- And later discuss the “static types or not” debate (it’s really a continuum)

Our plan

- Simply-typed Lambda-Calculus
- Safety = (preservation + progress)
- Extensions (pairs, datatypes, recursion, etc.)
- Digression: static vs. dynamic typing
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics (∀), Abstract types (∃), Recursive types
- Type inference

Adding integers

Adding integers to the lambda-calculus:

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e \mid c$

Values: $v ::= \lambda x.\ e \mid c$

Could add + and other primitives or just parameterize "programs" by them: $\lambda plus.\ \lambda minus.\ \ldots\ e$

- Like Pervasives in Caml
- A great idea for keeping language definitions small!

Stuck

- Key issue: can a program e "get stuck" (small-step):
  - $e \rightarrow^{*} e1$
  - e1 is not a value
  - There is no e2 such that $e1 \rightarrow e2$
- "What is stuck" depends on the semantics:

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x\,.\,e)\ v \rightarrow e\{v/x\}}$$

STLC Stuck

- `S ::= c e | x e | (λx.e) x | S e | (λx.e) S`

- It’s not normal to define these explicitly, but a great way to think about it.

- Most people don’t realize “safety” depends on the semantics:
  - We can add “cheat” rules to “avoid” being stuck.

Sound and complete

- Definition: A type system is sound if it never accepts a program that can get stuck
- Definition: A type system is complete if it always accepts a program that cannot get stuck
- Soundness and completeness are desirable
- But impossible (undecidable) for lambda-calculus
  - *If e has no constants or free variables then e (3 4) gets stuck iff e terminates*
  - As is any non-trivial property for a Turing-complete PL

What to do

- Old conclusion: “strong types for weak minds”
  - Need an unchecked cast (a back-door)

- Modern conclusion:
  - Make false positives rare and false negatives impossible (be sound and expressive)
  - Make workarounds reasonable
  - Justification: false negatives too expensive, have compile-time resources for “fancy” type-checking

- Okay, let’s actually try to do it…

Wrong attempt

$$\tau ::= \texttt{int} \mid \texttt{function}$$

A judgment: $\vdash \texttt{e} : \tau$

(for which we hope there's an efficient algorithm)

$$\frac{}{\vdash \texttt{c} : \texttt{int}} \qquad \frac{}{\vdash (\lambda \texttt{x}.\texttt{e}) : \texttt{function}}$$

$$\frac{\vdash \texttt{e1} : \texttt{function} \qquad \vdash \texttt{e2} : \texttt{int}}{\vdash \texttt{e1 e2} : \texttt{int}}$$

So very wrong

$$\frac{}{\vdash \texttt{c} : \texttt{int}} \qquad \frac{}{\vdash (\lambda \texttt{x}.\texttt{e}) : \texttt{function}}$$

$$\frac{\vdash \texttt{e1} : \texttt{function} \qquad \vdash \texttt{e2} : \texttt{int}}{\vdash \texttt{e1 e2} : \texttt{int}}$$

1. Unsound: `(λx.y) 3`
2. Disallows function arguments: `(λx. x 3) (λy.y)`
3. Types not *preserved*: `(λx.(λy.y)) 3`
    - Result is not an integer

Getting it right

1. Need to type-check function bodies, which have free variables
2. Need to distinguish functions according to argument and result types

For (1): $\Gamma ::= . \mid \Gamma, x : \tau$ and $\Gamma \vdash e : \tau$

- A type-checking environment (called a context)

For (2): $\tau ::= \texttt{int} \mid \tau \rightarrow \tau$

- Arrow is part of the (type) language (not meta)
- An infinite number of types
- Just like Caml

Examples and syntax

- Examples of types

  int → int

  (int → int) → int

  int → (int → int)

- Concretely → is *right-associative*, i.e.,
  - i.e., `τ1→ τ2→ τ3` is `τ1→ (τ2→ τ3)`
  - Just like Caml

STLC in one slide

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e \mid c$

Values: $v ::= \lambda x.\ e \mid e\ e$

Types: $\tau ::= \mathtt{int} \mid \tau \rightarrow \tau$

Contexts: $\Gamma ::= . \mid \Gamma, x : \tau$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x.e)\ v \rightarrow e\{v/x\}}$$

$$\frac{}{\Gamma \vdash c : \mathtt{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x{:}\ \tau1 \vdash e : \tau2}{\Gamma \vdash (\lambda x.e) : \tau1 \rightarrow \tau2} \qquad \frac{\Gamma \vdash e1 : \tau1 \rightarrow \tau2 \quad \Gamma \vdash e2 : \tau1}{\Gamma \vdash e1\ e2 : \tau2}$$

Rule-by-rule

$$\frac{}{\Gamma \vdash \mathtt{c} : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$

$$\frac{\Gamma, x\!: \tau 1 \vdash \mathtt{e} : \tau 2}{\Gamma \vdash (\lambda \mathtt{x}.\mathtt{e}) : \tau 1 \rightarrow \tau 2} \qquad \frac{\Gamma \vdash \mathtt{e1} : \tau 1 \rightarrow \tau 2 \quad \Gamma \vdash \mathtt{e2} : \tau 1}{\Gamma \vdash \mathtt{e1}\ \mathtt{e2} : \tau 2}$$

- Constant rule: context irrelevant
- Variable rule: lookup (no instantiation if x not in Γ)
- Application rule: “yeah, that makes sense”
- Function rule the interesting one…

The function rule

$$\frac{\Gamma, x{:}\ \tau 1 \vdash e : \tau 2}{\Gamma \vdash (\lambda x . e) : \tau 1 \rightarrow \tau 2}$$

- Where did `τ1` come from?
  - Our rule “inferred” or “guessed” it
  - To be syntax-directed, change **λ*x*.*e*** to **λ*x*: τ.*e*** and use that **τ**
- If we think of Γ as a partial function, we need x not already in it (alpha-conversion allows)

Our plan

- Simply-typed Lambda-Calculus
- Safety = (preservation + progress)
- Extensions (pairs, datatypes, recursion, etc.)
- Digression: static vs. dynamic typing
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics ($\forall$), Abstract types ($\exists$), Recursive types
- Type inference

Is it “right”?

- Can define any type system we want
- What we defined is sound and incomplete
- Can prove incomplete with one example
  - Every variable has exactly one simple type
  - Example (doesn’t get stuck, doesn’t typecheck)

    `(λx. (x(λy.y)) (x 3)) (λz.z)`

Sound

- Statement of soundness theorem:
  If . $\vdash$ **e:τ** and e $\rightarrow^{*}$e2, then e2 is a value or there exists an e3 such that e2 $\rightarrow$e3
- Proof is tough
  - Must hold for all e and any number of steps
  - But easy if these two theorems hold

1. Progress: If . $\vdash$ **e:τ** then e is a value or there exists an e’ such that e$\rightarrow$e’
2. Preservation: If . $\vdash$ **e:τ** and e$\rightarrow$e’ then . $\vdash$ **e:τ**

Let's prove it

Prove: If $\cdot \vdash e : \tau$ and $e \rightarrow^{*} e2$, then e2 is a value or
∃e3 such that $e2 \rightarrow e3$, assuming:

1. If $\cdot \vdash e : \tau$ then e is a value or ∃ e' such that $e \rightarrow e'$
2. If $\cdot \vdash e : \tau$ and $e \rightarrow e'$ then $\cdot \vdash e' : \tau$

Prove something stronger: Also show $\cdot \vdash e2 : \tau$

Proof: By induction on n where $e \rightarrow^{*} e2$ in n steps

- Case n=0: immediate from progress (e=e2)
- Case n>0: then ∃e2' such that…

What’s the point

- Progress is what we care about
- But Preservation is the invariant that holds no longer how long we have been running
- (Progress and Preservation) implies Soundness

- This is a very general/powerful recipe for showing you “don’t get to a bad place”
  - If invariant holds, you’re in a good place (progress) and you go to a good place (preservation)

- Details on next 2 slides less important…

Forget a couple things?

Progress: If $\cdot \vdash \mathbf{e} : \tau$ then e is a value or there exists an e’ such that e→e’

Proof: Induction on (height of) derivation tree for $\cdot \vdash \mathbf{e} : \tau$

Rough idea:

- Trivial unless e is an application
- For e = e1 e2,
  - If left or right not a value, induction
  - If both values, e1 must be a lambda…

Forget a couple things?

Preservation: If $\cdot \vdash \mathtt{e}:\tau$ and $\mathtt{e} \to \mathtt{e}'$ then $\cdot \vdash \mathtt{e}:\tau$

Also by induction on assumed typing derivation.

The trouble is when $\mathtt{e} \to \mathtt{e}'$ involves substitution –
requires another theorem

Substitution: If $\Gamma, x{:}\ \tau 1 \vdash \mathtt{e}:\tau$ and $\Gamma \vdash \mathtt{e1}:\tau 1$, then
$\Gamma \vdash \mathtt{e}\{\mathtt{e1}/\mathtt{x}\}:\tau$

Our plan

- Simply-typed Lambda-Calculus
- Safety = (preservation + progress)
- Extensions (pairs, datatypes, recursion, etc.)
- Digression: static vs. dynamic typing
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics (∀), Abstract types (∃), Recursive types
- Type inference

Having laid the groundwork…

- So far:
  - Our language (STLC) is tiny
  - We used heavy-duty tools to define it

- Now:
  - Add lots of things quickly
  - Because our tools are all we need

- And each addition will have the same form…

A method to our madness

- The plan
  - Add syntax
  - Add new semantic rules (including substitution)
  - Add new typing rules

- If our addition extends the syntax of types, then
  - We will have new values (of that type)
  - And ways to make the new values
    - (called introduction forms)
  - And ways to use the new values
    - (called elimination forms)

Let bindings (CBV)

e ::= … | let x = e1 in e2

(no new values or types)

$$\frac{e1 \rightarrow e1'}{\text{let } x = e1 \text{ in } e2 \rightarrow \text{let } x = e1' \text{ in } e2}$$

$$\frac{}{\text{let } x = v \text{ in } e2 \rightarrow e2\{v/x\}}$$

$$\frac{\Gamma \vdash e1 : \tau 1 \qquad \Gamma, x{:}\ \tau 1 \vdash e2 : \tau 2}{\Gamma \vdash \text{let } x = e1 \text{ in } e2 \ :\ \tau 2}$$

Let as sugar?

Let is actually so much like lambda, we could use 2 other different but equivalent semantics

2. let x=e1 in e2 is sugar (a different concrete way to write the same abstract syntax) for $(\lambda x.e2)\ e1$

3. Instead of semantic rules on last slide, use just

$$\frac{}{\text{let } x = e1 \text{ in } e2 \rightarrow (\lambda x.e2)\ e1}$$

Note: In Caml, let is *not* sugar for application because let is type-checked differently (type variables)

Booleans

e ::= … | tru | fls | e ? e : e
v ::= … | tru | fls
τ ::= … | bool

$$\frac{e1 \rightarrow e1'}{e1 \; ? \; e2 : e3 \rightarrow e1' \; ? \; e2 : e3}$$

$$\frac{}{\Gamma \vdash \text{tru} : \texttt{bool}}$$

$$\frac{}{\text{tru} \; ? \; e2 : e3 \rightarrow e2}$$

$$\frac{}{\Gamma \vdash \text{fls} : \texttt{bool}}$$

$$\frac{}{\text{fls} \; ? \; e2 : e3 \rightarrow e3}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{bool} \quad \Gamma \vdash \texttt{e2} : \tau \quad \Gamma \vdash \texttt{e3} : \tau}{\Gamma \vdash e1 \; ? \; e2 : e3 : \tau}$$

Caml? Large-step?

- In homework 3, you'll add conditionals, pairs, etc. to our environment-based large-step interpreter
- Compared to last slide
  - Different meta-language (cases rearranged)
  - Large-step instead of small
  - If tests an integer for 0 (like C)
- Large-step booleans with inference rules

$$\frac{e1 \Downarrow \text{tru} \qquad e2 \Downarrow v}{e1 \;?\; e2 : e3 \Downarrow v} \qquad \frac{e1 \Downarrow \text{fls} \qquad e3 \Downarrow v}{e1 \;?\; e2 : e3 \Downarrow v}$$

$$\frac{}{\text{tru} \Downarrow \text{tru}} \qquad \frac{}{\text{fls} \Downarrow \text{fls}}$$

Pairs (CBV, left-to-right)

$e ::= \ldots \mid (e,e) \mid e.1 \mid e.2$

$v ::= \ldots \mid (v,v)$

$\tau ::= \ldots \mid \tau * \tau$

$$\frac{e1 \rightarrow e1'}{(e1,e2) \rightarrow (e1',e2)} \qquad \frac{e2 \rightarrow e2'}{(v,e2) \rightarrow (v,e2')} \qquad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{}{(v1,v2).1 \rightarrow v1} \qquad \frac{}{(v1,v2).2 \rightarrow v2}$$

$$\frac{\Gamma \vdash e1 : \tau1 \quad \Gamma \vdash e2 : \tau2}{\Gamma \vdash (e1,e2) : \tau1 * \tau2} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.1 : \tau1} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.2 : \tau2}$$

Best guess of where lecture 5 will end

Toward Sums

- Next addition: sums (much like ML datatypes)
- Informal review of ML datatype basics

```
type t = A of t1 | B of t2 | C of t3
```

  - Introduction forms: constructor-applied-to-exp
  - Elimination forms: match e1 with pat `->` exp …
  - Typing: If e has type t1, then A e has type t …

Unlike ML, part 1

- ML datatypes do a lot at once
  - Allow recursive types
  - Introduce a new *name* for a type
  - Allow type parameters
  - Allow fancy pattern matching
- What we do will be *simpler*
  - Add recursive types separately later
  - Avoid names (a bit simpler in theory)
  - Avoid type parameters (for simplicity)
  - Only patterns of form A x (rest is sugar)

Unlike ML, part 2

- What we add will also be *different*
  - Only two constructors A and B
  - All sum types use these constructors
  - So A e can have any sum type allowed by e’s type
  - No need to declare sum types in advance
  - Like functions, will “guess types” in our rules

- This should still help explain what datatypes are

- After formalism, will compare to C unions and OOP

The math (with type rules to come)

e ::= … | A e | B e | match e with A x -> e | B y -> e

v ::= … | A v | B v

τ ::= … | $\tau + \tau$

$$\frac{e \rightarrow e'}{\text{A } e \rightarrow \text{A } e'} \qquad \frac{e \rightarrow e'}{\text{B } e \rightarrow \text{B } e'} \qquad \frac{e1 \rightarrow e1'}{\begin{array}{l}\text{match } e1 \text{ with A } x\text{->}e2 \text{ |B } y \text{ -> } e3 \\ \rightarrow \ \text{match } e1' \text{ with A } x\text{->}e2 \text{ |B } y \text{ -> } e3\end{array}}$$

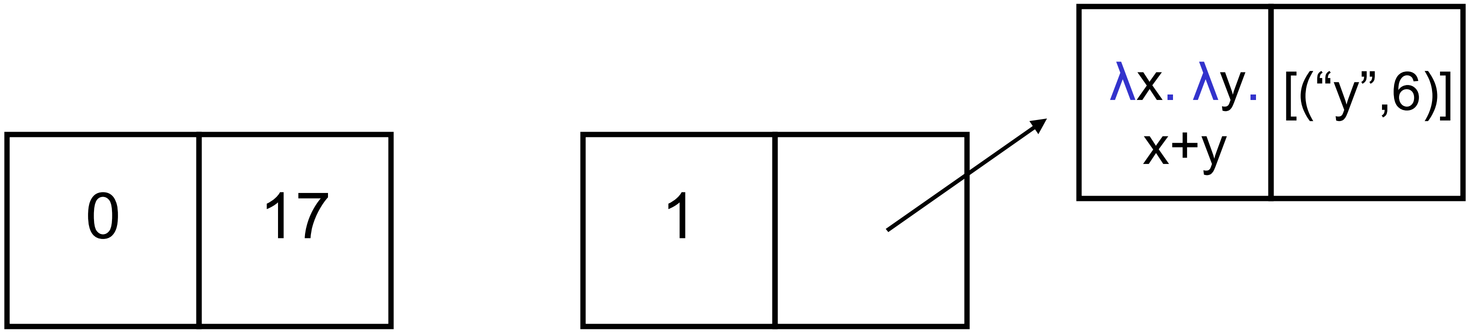
$$\frac{}{\text{match A } v \text{ with A } x\text{->}e2 \text{ | B } y \text{ -> } e3 \ \rightarrow \ e2\{v/x\}}$$

$$\frac{}{\text{match B } v \text{ with A } x\text{->}e2 \text{ | B } y \text{ -> } e3 \ \rightarrow \ e3\{y/x\}}$$

Low-level view

You can think of datatype values as “pairs”

- First component is A or B (or 0 or 1 if you prefer)
- Second component is “the data”
- e2 or e3 evaluated with “the data” in place of the variable
- This is all like Caml as in lecture 1
- Example values of type int + (int -> int):



Typing rules

- Key idea for datatype exp: “other can be anything”
- Key idea for matches: “branches need same type”
  - Just like conditionals

$$\frac{\Gamma \vdash \mathtt{e} : \tau 1}{\Gamma \vdash \mathtt{A}\ \mathtt{e} : \tau 1 + \tau 2} \qquad \frac{\Gamma \vdash \mathtt{e} : \tau 2}{\Gamma \vdash \mathtt{B}\ \mathtt{e} : \tau 1 + \tau 2}$$

$$\frac{\Gamma \vdash \mathtt{e1} : \tau 1 + \tau 2 \quad \Gamma, \mathtt{x} : \tau 1 \vdash \mathtt{e2} : \tau \quad \Gamma, \mathtt{y} : \tau 2 \vdash \mathtt{e3} : \tau}{\Gamma \vdash \mathtt{match}\ \mathtt{e1}\ \mathtt{with}\ \mathtt{A}\ \mathtt{x} \texttt{->} \mathtt{e2} \mid \mathtt{B}\ \mathtt{y} \texttt{->} \mathtt{e3} : \tau}$$

Compare to pairs, part 1

- “pairs and sums” is a big idea
  - Languages should have both (in some form)
  - Somehow pairs come across as simpler, but they’re really “dual” (see Curry-Howard soon)
- Introduction forms:
  - pairs “need both”, sums “need one”

$$\frac{\Gamma \vdash \mathtt{e1} : \tau\mathtt{1} \quad \Gamma \vdash \mathtt{e2} : \tau\mathtt{2}}{\Gamma \vdash (\mathrm{e1},\mathrm{e2}) : \tau\mathtt{1} * \tau\mathtt{2}} \qquad \frac{\Gamma \vdash \mathtt{e} : \tau\mathtt{1}}{\Gamma \vdash \mathrm{A\ e} : \tau\mathtt{1} + \tau\mathtt{2}} \qquad \frac{\Gamma \vdash \mathtt{e} : \tau\mathtt{2}}{\Gamma \vdash \mathrm{B\ e} : \tau\mathtt{1} + \tau\mathtt{2}}$$

Compare to pairs, part 2

- Elimination forms
  - Pairs get either, sums must be prepared for either

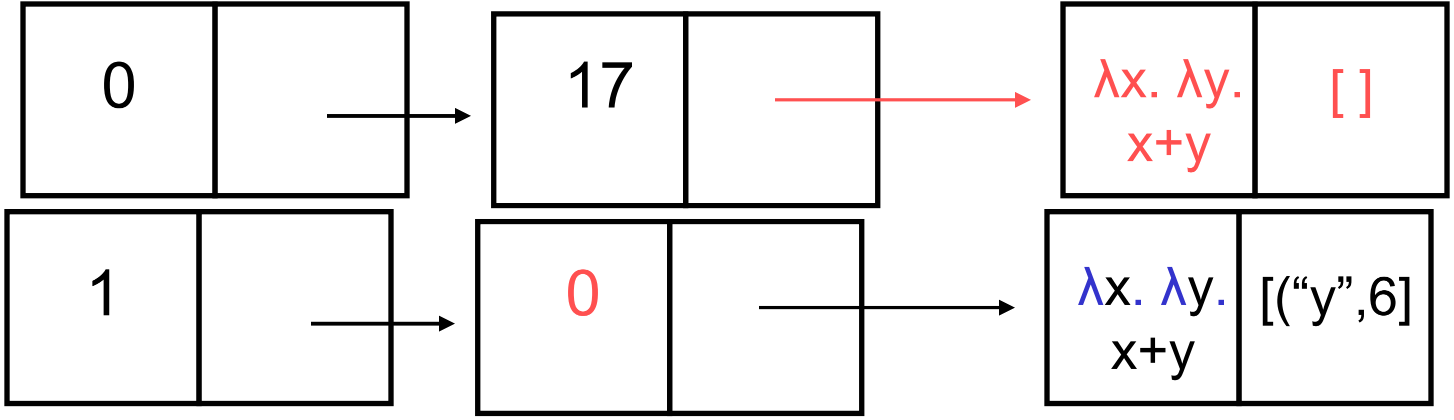
$$\frac{\Gamma \vdash \mathtt{e}:\tau 1 * \tau 2}{\Gamma \vdash \mathtt{e.1}:\tau 1} \qquad \frac{\Gamma \vdash \mathtt{e}:\tau 1 * \tau 2}{\Gamma \vdash \mathtt{e.2}:\tau 2}$$

$$\frac{\Gamma \vdash e1 : \tau 1 + \tau 2 \qquad \Gamma, x:\tau 1 \vdash e2 : \tau \qquad \Gamma, x:\tau 2 \vdash e3 : \tau}{\Gamma \vdash \text{match } e1 \text{ with } A\ x \text{->} e2 \mid B\ y \text{ -> } e3 : \tau}$$

Living with just pairs

- If stubborn you can cram sums into pairs (don't!)
  - Round-peg, square-hole
  - Less efficient (dummy values)
    - Flattened pairs don't change that
  - More error-prone (may use dummy values)
  - Example: `int + (int -> int)` becomes

    `int * (int * (int -> int))`



Sums in other guises

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> …
```

Meets C:

```
 struct t {
  enum  {A, B, C}           tag;
  union {t1 a; t2 b; t3 c;} data;
 };
… switch(e->tag){ case A: t1 x=e->data.a;…
```

- No static checking that tag is obeyed
- As fat as the fattest variant (avoidable with casts)
  - Mutation bites again!
- Shameless plug: *Cyclone* has ML-style datatypes

Sums in other guises

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> …
```

Meets Java:

```
 abstract class t {abstract Object m();}
 class A extends t { t1 x; Object m(){…}}
 class B extends t { t2 x; Object m(){…}}
 class C extends t { t3 x; Object m(){…}}
 … e.m() …
```

- A new method for each match expression
- Supports orthogonal forms of extensibility (will come back to this)

Where are we

- Have added let, bools, pairs, sums
- Could have done string, floats, records, …
- Amazing fact:
  - Even with everything we have added so far, every program terminates!
  - I.e., if $. \vdash \mathtt{e} : \tau$ then there exists a value v such that $\mathtt{e} \rightarrow^{*} v$
  - Corollary: Our encoding of fix won’t type-check
- To regain Turing-completeness, we need explicit support for recursion

Recursion

- We could add “fix e” (ask me if you’re curious), but most people find “letrec f x e” more intuitive

e ::= … | letrec f x e

v ::= … | letrec f x e

(no new types)

“Substitute argument like lambda & whole function for f”

$$\frac{}{(\text{letrec } f\ x\ e)\ v \rightarrow (e\{v/x\})\{(\text{letrec } f\ x\ e) / f\}}$$

$$\frac{\Gamma, f: \tau 1 \rightarrow \tau 2, x:\tau 1 \vdash e : \tau 2}{\Gamma \vdash \text{letrec } f\ x\ e : \tau 1 \rightarrow \tau 2}$$

Our plan

- Simply-typed Lambda-Calculus
- Safety = (preservation + progress)
- Extensions (pairs, datatypes, recursion, etc.)
- Digression: static vs. dynamic typing
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics (∀), Abstract types (∃), Recursive types
- Type inference

A couple slides for context examples

Redivision of labor

```
type ectxt = Hole
           | Left of ectxt * exp
           | Right of exp * ectxt (*exp a value*)
let rec split e =
  match e with
   A(L(s1,e1),L(s2,e2)) -> (Hole,e)
 | A(L(s1,e1),e2) -> let (ctx2,e3) = split e2 in
                     (Right(L(s1,e1),ctx2), e3)
 | A(e1,e2)       -> let (ctx2,e3) = split e1 in
                     (Left(ctx2,e2), e3)
```

Redivision of labor

```
type ectxt = Hole
           | Left of ectxt * exp
           | Right of exp * ectxt  (*exp a value*)
let rec split e =
  match e with
   A(L(s1,e1),L(s2,e2))  -> (Hole,e)
 | A(L(s1,e1),e2)  -> let (ctx2,e3) = split e2 in
                      (Right(L(s1,e1),ctx2), e3)
 | A(e1,e2)        -> let (ctx2,e3) = split e1 in
                      (Left(ctx2,e2), e3)
```