Remember our symbol-pile

Expressions:
\[ e ::= x \mid \lambda x. e \mid e \ e \]

Values:
\[ v ::= \lambda x. e \]

\[ e_1 \downarrow \lambda x. e_3 \| e_2 \downarrow v_2 \| e_3\{v_2/x\} \downarrow v \]

\[ \lambda x. e \downarrow \| \lambda x. e \downarrow \]

\[ e_1 \downarrow \| e_2 \downarrow v \]

\[ \{\text{app}\} \]

\[ \{\text{lam}\} \]
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** \( e \rightarrow e' \)

\[
\begin{align*}
e1 & \rightarrow e1' \\
e2 & \rightarrow e2' \\
\hline
\begin{array}{c}
e1 \ e2 \rightarrow e1' \ e2 \\
v \ e2 \rightarrow v \ e2' \\
(\lambda x \ . \ e) \ v \rightarrow e\{v/x\}
\end{array}
\]

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

\[
\begin{align*}
e1 & \rightarrow e2 \\
e2 & \rightarrow^* e3 \\
\hline
\begin{array}{c}
e \rightarrow^* e \\
e1 \rightarrow^* e3
\end{array}
\]
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: \( e = \text{fill}(\text{split } e) \)
- But we can write \(\text{interp\_small} \) with them
  - Shows a step has three parts: split, subst, fill

```ocaml
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

```ml
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
  \[ A(L(s_1,e_1),L(s_2,e_2)) \]
- New: could also be some \( Letcc(s_1,e_1) \)

```
let rec split e = (* change: one new case *)
    match e with
    | Letcc(s_1,e_1) -> (Hole,e) (* new *)
    | A(L(s_1,e_1),L(s_2,e_2)) -> (Hole,e)
    | A(L(s_1,e_1),e_2) -> let (ctx2,e_3) = split e_2 in
                           (Right(L(s_1,e_1),ctx2), e_3)
    | A(e_1,e_2)       -> let (ctx2,e_3) = split e_1 in
                           (Left(ctx2,e_2), e_3)
    | _ -> 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”

```ocaml
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’!):

\[ H; e_1 \downarrow \{v_1, \ldots, v_n\} \quad H; e_2 \downarrow i \quad i > n \]

\[ \text{H; e}_1[i]=e_2 \downarrow \text{H’;s’} \]

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 ($\forall$), Abstract types ($\exists$), 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):
  - $e \rightarrow^* e_1$
  - $e_1$ is not a value
  - There is no $e_2$ such that $e_1 \rightarrow e_2$
- “What is stuck” depends on the semantics:

\[
\begin{align*}
e_1 & \rightarrow e_1' & e_2 & \rightarrow e_2' \\
\hline
e_1 e_2 & \rightarrow e_1' e_2 & v e_2 & \rightarrow v e_2' \\
(\lambda x. e) v & \rightarrow e[v/x]
\end{align*}
\]
• 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

\[ \tau ::= \text{int} \mid \text{function} \]

A judgment: \[ \vdash e : \tau \]
(for which we hope there’s an efficient algorithm)

\[ \vdash c : \text{int} \quad \vdash (\lambda x. e) : \text{function} \]

\[ \vdash e_1 : \text{function} \quad \vdash e_2 : \text{int} \]

\[ \vdash e_1 e_2 : \text{int} \]
So very wrong

1. Unsound: \((\lambda x. y) \ 3\)
2. Disallows function arguments: \((\lambda x. \ x \ 3) \ (\lambda y. y)\)
3. Types not preserved: \((\lambda x. (\lambda y. y)) \ 3\)
   - 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): \( \tau ::= \text{int} \mid \tau \rightarrow \tau \)
- 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., \( \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \) is \( \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3) \)
  – 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:  $\tau ::= \text{int} \mid \tau \rightarrow \tau$

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

$$
\begin{align*}
& e_1 \rightarrow e_1' \quad e_2 \rightarrow e_2' \\
\frac{}{e_1 \ e_2 \rightarrow e_1' \ e_2} & \quad \frac{}{v \ e_2 \rightarrow v \ e_2'} & \quad (\lambda x. \ e) \ v \rightarrow e\{v/x\}
\end{align*}
$$

$$
\begin{align*}
\frac{}{\Gamma \vdash c : \text{int}} & \quad \frac{}{\Gamma \vdash x : \Gamma(x)} \\
\frac{}{\Gamma, x : \tau_1 \vdash e : \tau_2} & \quad \frac{}{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1} \\
\frac{}{\Gamma \vdash (\lambda x. \ e) : \tau_1 \rightarrow \tau_2} & \quad \frac{}{\Gamma \vdash e_1 \ e_2 : \tau_2}
\end{align*}
$$
Rule-by-rule

- 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

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \]

\[ \Gamma \vdash (\lambda x. e) : \tau_1 \rightarrow \tau_2 \]

- Where did \( \tau_1 \) come from?
  - Our rule “inferred” or “guessed” it
  - To be syntax-directed, change \( \lambda x. e \) to \( \lambda x : \tau. e \) and use that \( \tau \)
- If we think of \( \Gamma \) 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)
    \[(\lambda x. \ (x \ (\lambda y. y)) \ (x \ 3)) \ (\lambda z. z)\]
Sound

- Statement of soundness theorem:
  If \( \vdash e : \tau \) and \( e \rightarrow^* e_2 \), then \( e_2 \) is a value or there exists an \( e_3 \) such that \( e_2 \rightarrow e_3 \)

- Proof is tough
  - Must hold for all \( e \) and any number of steps
  - But easy if these two theorems hold
  1. Progress: If \( \vdash e : \tau \) then \( e \) is a value or there exists an \( e' \) such that \( e \rightarrow e' \)
  2. Preservation: If \( \vdash e : \tau \) and \( e \rightarrow e' \) then \( \vdash e : \tau \)
Let’s prove it

Prove: If \( \vdash e : \tau \) and \( e \rightarrow^* e_2 \), then \( e_2 \) is a value or \( \exists e_3 \) such that \( e_2 \rightarrow e_3 \), assuming:
1. If \( \vdash e : \tau \) then \( e \) is a value or \( \exists e' \) such that \( e \rightarrow e' \)
2. If \( \vdash e : \tau \) and \( e \rightarrow e' \) then \( \vdash e : \tau \)

Prove something stronger: Also show \( \vdash e_2 : \tau \)

Proof: By induction on \( n \) where \( e \rightarrow^* e_2 \) in \( n \) steps
• Case \( n=0 \): immediate from progress (\( e=e_2 \))
• Case \( n>0 \): then \( \exists e_2' \) 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 $\vdash e : \tau$ then $e$ is a value or there exists an $e'$ such that $e \rightarrow e'$.

Proof: Induction on (height of) derivation tree for $\vdash e : \tau$.

Rough idea:

- Trivial unless $e$ is an application.
- For $e = e_1 e_2$,
  - If left or right not a value, induction
  - If both values, $e_1$ must be a lambda...
Forget a couple things?

Preservation: If \( \vdash e : \tau \) and \( e \rightarrow e' \) then \( \vdash e : \tau \)

Also by induction on assumed typing derivation.

The trouble is when \( e \rightarrow e' \) involves substitution – requires another theorem

Substitution: If \( \Gamma, x : \tau_1 \vdash e : \tau \) and \( \Gamma \vdash e_1 : \tau_1 \), then \( \Gamma \vdash e\{e_1/x\} : \tau \)
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 ($\forall$), Abstract types ($\exists$), 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)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]
(no new values or types)

\[ e_1 \rightarrow e_1' \]

\[ \frac{\quad \frac{e_1 \rightarrow e_1'}{\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2} \quad }{\text{let } x = v \text{ in } e_2 \rightarrow e_2\{v/x\}} \]

\[ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \]
Let as sugar?

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

2. let \( x = e_1 \) in \( e_2 \) is sugar (a different concrete way to write the same abstract syntax) for \( (\lambda x. e_2) \) \( e_1 \)

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

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow (\lambda x. e_2) \ e_1
\]

Note: In Caml, let is \textit{not} sugar for application because let is type-checked differently (type variables)
Booleans

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

e1 → e1’

___
e1 ? e2 : e3 → e1’ ? e2 : e3

___
tru ? e2 : e3 → e2

___
fls ? e2 : e3 → e3

Γ |- tru : bool

Γ |- fls : bool

Γ |- e1 : bool Γ |- e2 : τ Γ |- e3 : τ

Γ |- e1 ? e2 : e3 : τ
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
  \[
  \begin{align*}
  e_1 \downarrow \text{tru} \quad & e_2 \downarrow \text{v} \quad & e_1 \downarrow \text{fls} \quad & e_3 \downarrow \text{v} \\
  \hline
  e_1 ? e_2 : e_3 \downarrow \text{v} \quad & e_1 ? e_2 : e_3 \downarrow \text{v} \\
  \hline
  \text{tru} \downarrow \text{tru} \quad & \text{fls} \downarrow \text{fls}
  \end{align*}
  \]
Pairs (CBV, left-to-right)

\[
\begin{align*}
e & ::= \ldots | (e,e) | e.1 | e.2 \\
v & ::= \ldots | (v,v) \\
\tau & ::= \ldots | \tau^* \tau \\
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1,e_2) : \tau_1^* \tau_2 \\
\Gamma \vdash e_1 : \tau_1 & \\
\Gamma \vdash e.1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 & \\
\Gamma \vdash e.2 : \tau_2
\end{align*}
\]
Best guess of where lecture 5 will end
Toward Sums

• Next addition: sums (much like ML datatypes)

• Informal review of ML datatype basics

\[
type \; t = \text{A of } t_1 \mid \text{B of } t_2 \mid \text{C of } t_3
\]

– 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)

\[
e ::= \ldots \mid A \, e \mid B \, e \mid \text{match } e \text{ with } A \, x \rightarrow e \mid B \, y \rightarrow e
\]

\[
v ::= \ldots \mid A \, v \mid B \, v
\]

\[
\tau ::= \ldots \mid \tau + \tau
\]

\[
e \rightarrow e' \quad e \rightarrow e' \quad e1 \rightarrow e1'
\]

\[
\begin{align*}
A \, e & \rightarrow A \, e' \\
B \, e & \rightarrow B \, e'
\end{align*}
\]

\[
\text{match } e1 \text{ with } A \, x \rightarrow e2 \mid B \, y \rightarrow e3 \rightarrow \text{match } e1' \text{ with } A \, x \rightarrow e2 \mid B \, y \rightarrow e3
\]

\[
\text{match } A \, v \text{ with } A \, x \rightarrow e2 \mid B \, y \rightarrow e3 \rightarrow e2\{v/x\}
\]

\[
\text{match } B \, v \text{ with } A \, x \rightarrow e2 \mid B \, y \rightarrow e3 \rightarrow e3\{y/x\}
\]
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):

<table>
<thead>
<tr>
<th>0</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\[ \lambda x. \lambda y. \left[ \text{“y”,6} \right] \]

\[ x+y \]
Typing rules

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash e : \tau_2 \\
\hline
\Gamma \vdash A \ e : \tau_1 + \tau_2 \\
\Gamma \vdash B \ e : \tau_1 + \tau_2 \\
\hline
\Gamma \vdash e_1 : \tau_1 + \tau_2 \\
\Gamma, x : \tau_1 \vdash e_2 : \tau \\
\Gamma, x : \tau_2 \vdash e_3 : \tau \\
\hline
\Gamma \vdash \text{match } e_1 \text{ with } A \ x \rightarrow e_2 \mid B \ y \rightarrow e_3 : \tau
\end{align*}
\]
**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”

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash A e : \tau_1 + \tau_2 & \quad \Gamma \vdash B e : \tau_1 + \tau_2
\end{align*}
\]
Compare to pairs, part 2

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

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \ast \tau_2 \\
\Gamma & \vdash e : \tau_1 \ast \tau_2 \\
\Gamma & \vdash e.1 : \tau_1 \\
\Gamma & \vdash e.2 : \tau_2 \\
\Gamma, x: \tau_1 & \vdash e2 : \tau \\
\Gamma, x: \tau_2 & \vdash e3 : \tau \\
\Gamma & \vdash \text{match } e1 \text{ with } A \ x \rightarrow e2 \mid B \ y \rightarrow e3 : \tau
\end{align*}
\]
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

\[
\text{type } t = \text{A of } t1 \mid \text{B of } t2 \mid \text{C of } t3
\]
\[
\text{match } e \text{ with } \text{A } x \rightarrow \ldots
\]

Meets C:

\[
\text{struct } t 
\{
\text{enum } \{\text{A, B, C}\} \text{ tag;}
\text{union } \{t1 \ a; \ t2 \ b; \ t3 \ c;\} \text{ data;}
\};
\]

\[
\ldots \text{ switch}(e->\text{tag})\{ \text{ case } \text{A}: \ t1 \ x=e->\text{data}.a;\ldots
\]

• No static checking that tag is obeyed
• As fat as the fattest variant (avoidable with casts)
  – Mutation bites again!
• Shameless plug: \textit{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 \( \vdash e : \tau \) then there exists a value \( v \) such that \( e \rightarrow^* v \)
  – 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

\[
e ::= \ldots \mid \text{letrec } f \ x \ e
\]

\[
v ::= \ldots \mid \text{letrec } f \ x \ e
\]

(no new types)

“Substitute argument like lambda & whole function for f”

\[
(\text{letrec } f \ x \ e) \ v \rightarrow (e\{v/x\})\{(\text{letrec } f \ x \ e) / f\}
\]

\[
\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2
\]

\[
\Gamma \vdash \text{letrec } f \ x \ e : \tau_1 \rightarrow \tau_2
\]
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)