Remember our symbol-pile

Expressions: \( e ::= x \mid \lambda x. \ e \mid e \ e \)
Values: \( v ::= \lambda x. \ e \)

\[ e \downarrow v \]

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

\[ e1 \downarrow \lambda x. \ e3 \] \[ e2 \downarrow v2 \] \[ e3\{v2/x\} \downarrow v \]

\[ e1 \ e2 \downarrow v \]

\( e3\{v2/x\} \) is the “capture-avoiding substitution of \( v2 \) for \( x \) in \( e3 \)”

• Capture is an insidious error in program rewriters
• Formally avoided via “systematic renaming (alpha conversion)”
  – Ensure free variables in \( v2 \) are not binders in \( e3 \)
Untyped Lambda Calculus

- Go back to math metalanguage
  - Notes on concrete syntax (relates to OCaml)
  - Define semantics with inference rules
- Lambda encodings (show our language is mighty)
- Define substitution precisely
  - And revisit function equivalences
- Environments

Now:
- Small-step
- Play with continuations ("very fancy" language feature)

Then: On to types
Small-step CBV

• Left-to-right small-step judgment

\[ e \rightarrow e' \]

\[
\begin{align*}
& e_1 \rightarrow e'_1 \\
\quad & e_2 \rightarrow e'_2 \\
& e_1 e_2 \rightarrow e'_1 e'_2 \\
& v e_2 \rightarrow v e'_2 \\
& (\lambda x. e) v \rightarrow e[v/x]
\end{align*}
\]

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

\[
\begin{align*}
& e \rightarrow^* e' \\

& e \rightarrow^* e \\
& e_1 \rightarrow e_2 \\
& e_2 \rightarrow^* e_3 \\
& e_1 \rightarrow^* e_3
\end{align*}
\]
In OCaml

definition of expression type:

```ocaml
type exp =
  V of string |
  L of string * exp |
  A of exp * exp
```

operations on expressions:

```ocaml
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…

• For all e and v,
  \[ e \downarrow v \text{ if and only if } e \rightarrow^* v \]

• Small-step distinguishes infinite-loops from stuck programs

• It’s closer to a contextual semantics that can define continuations
  – We’ll stick to OCaml for this
  – And we’ll do it much less efficiently than is possible
    • For the curious: read about Landin’s SECD machine [1960!]
Rethinking small-step

- An e is a tree of calls, with variables or lambdas at the leaves

- Find the next function call (or other “primitive step”) to do
- Do it
- Repeat (“new” next primitive step could be various places)

- Let’s move the first step out and produce a data structure describing where the next “primitive step” occurs
  - Called an evaluation context
  - Think call stack
Compute the context

(* represent "where" the next step "is" *)

```ocaml
type ectxt = Hole
  | ALeft of ectxt * exp
  | ARight of exp * ectxt (*exp a value*)

let rec split e = (*return ctxt & what's in it*)
  match e with
  A(L(s1,e1),L(s2,e2)) -> (Hole,e)
  | A(L(s1,e1),e2) -> let (ctx2,e3) = split e2 in
    (ARight(L(s1,e1),ctx2), e3)
  | A(e1,e2)  -> let (ctx1,e3) = split e1 in
    (ALeft(ctx1,e2), e3)
  | _ -> raise BadArgument
```
Fill a context

- We can also take a context and fill its hole with an expression to make a new program (expression)

```ocaml
type ectxt = Hole
  | ALeft of ectxt * exp
  | ARight of exp * ectxt (*exp a value*)

let rec fill ctx e = (* plug the hole *)
  match ctx with
  Hole -> e
  | ALeft(ctx2,e2) -> A(fill ctx2 e, e2)
  | ARight(e2,ctx2) -> A(e2, fill ctx2 e)
```
So what?

• Haven’t done much yet:
  - \( e = (\text{let } \text{ctxt}, e2 = \text{split } e \text{ in } \text{fill } \text{ctxt} e2) \)
• But we can rewrite \( \text{interp\_small} \) with them
  - A step has three parts: split, substitute, fill

```ocaml
let rec interp_small e =
  match e with
  | V _ -> failwith "interp_small" (*unbound var*)
  | L _ -> e
  | A _ ->
    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 Homework 2 with heaps, but this “is” the call stack)
  - It’s easy given this semantics!

- Sufficient for:
  - Exceptions
  - Cooperative threads / coroutines
  - “Time travel” with stacks
  - setjmp/longjmp

- Also (not shown): No need to resplit each time – “keep track”
Language w/ continuations

- New expression: `letcc` gets current context ("grab the stack")
- Now 2 kinds of values, but use application to use both
  - Could instead have 2 kinds of application + errors
- New kind stores a context (that can be restored)

```plaintext
type exp =
  V of string
| L of string*exp
| A of exp * exp
| letcc of string * exp (* new *)
| Cont of ectxt (* new *)

and ectxt = Hole (* no change *)
| ALeft of ectxt * exp
| ARight of exp * ectxt
```
Split with Letcc

- Old: All values were some $L(s,e)$
- New: Values can also be $\text{Cont } c$

- Old: active expression (thing in the hole) always some $A(L(s_1,e_1), L(s_2,e_2))$
- New: active expression (thing in the hole) can be:
  - $A(v_1,v_2)$
  - $\text{Letcc}(s,e)$

- So $\text{split}$ looks quite different to implement these changes
  - Not really that different
- $\text{fill}$ does not change at all
Split with Letcc

```ocaml
let isValue e =  
  match e with  
    L _ -> true | Cont _ -> true | _ -> false

let rec split e =  
  match e with  
    Letcc(s1,e1) -> (Hole,e) (* new *)  
  | A(e1,e2) ->  
      if isValue e1 && isValue e2  
      then (Hole,e)  
      else if isValue e1  
      then let (ctx2,e3) = split e2 in  
          (ARight(e1,ctx2),e3)  
      else let (ctx1,e3) = split e1 in  
          (ALeft(ctx1,e2), e3)  
  | _ -> failwith "bad args to split"
```
All the action

- **Letcc** creates a **Cont** that "grabs the current context"
- **A where** body is a **Cont** "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,e3), v)) ->
              interp_small(fill ctx (subst e3 v s3))
              | (ctx, Letcc(s3,e3)) ->
              interp_small(fill ctx
                              (*woah!!!*) (subst e3 (Cont ctx) s3))
              | (ctx, A(Cont ctx2, v)) ->
              interp_small(fill ctx2 v) (*woah!!!*)
  | _      -> failwith "bad split"
```
Toy Examples

[In language with addition too and explicit “throw”]

\[
1 + (\text{letcc } k. \ 2 + 3) \rightarrow^* 6
\]

\[
1 + (\text{letcc } k. \ 2 + (\text{throw } k \ 3)) \rightarrow^* 4
\]

\[
1 + (\text{letcc } k. \ (\text{throw } k \ (2+3))) \rightarrow^* 6
\]

\[
1 + (\text{letcc } k. \ (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 2)))) \rightarrow^* 3
\]

Also note evaluation-order matters, even without mutation (!)

\[
\text{letcc } k. \ (\text{throw } k \ 1) + (\text{throw } k \ 2)
\]
Example Uses

• Continuations for exceptions is “easy”
  – \texttt{Letcc(x,e)} for try, \texttt{Apply(Var x, v)} for raise \( v \) in \( e \)
• Coroutines can yield to each other
  – 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”
  – Example of “time travel” to “old stack”…
“Time Travel”

OCaml doesn’t have first-class continuations, but if it did:

```ocaml
let valOf x = match x with None -> failwith "" |
  Some y -> y

let x = ref true (*avoids infinite loop*)
let g = ref None
let y = ref (1 + 2 + (letcc k -> (g := Some k); 3))
let z = if !x
  then (x := false;
       throw (valOf (!g)) 7;
       42)
  else !y

(* what is z bound to and why? *)
```
A lower-level view

- If you’re confused, think call-stacks
  - What if YFL had these operations:
    - Store current stack in \( x \) (cf. \texttt{Letcc})
    - Replace current stack with stack in \( x \)
    - Need to “fill the stack’s hole” with something different and/or when state is different or you’ll have an infinite loop

- Implementing (e.g., compiling) \texttt{Letcc}
  - You do not actually split/fill at each step
  - Cannot just do \texttt{setjmp/longjmp} because a continuation can get returned from a function and used later!
  - Can actually copy stacks (expensive)
  - Or can avoid stacks (put stack-frames in heap)
    - Just share and rely on garbage collection
    - Or…
The CPS-Transform

There’s a subset of lambda-calculus called “continuation-passing style” (CPS). It’s amazing:

– Every call is [essentially] a tail-call
– It can do everything full lambda-calculus can
– In fact, one can automatically translate full lambda-calculus into CPS
  • CPS(e) (\(\lambda x.x\)) evaluates to 42 if and only if e does
  • Different translations fix different evaluation orders
– The translation is a powerful compiler technique
– And it motivates/explains a powerful programming idiom
– And it makes letcc and throw \(O(1)\) operations
– And it’s mind-bending…
CPS transformation

A CPS transformation is a metafunction from expressions to expressions

– Intuition: never return; always call the continuation you’re given as an argument

– An int expression becomes an
  \[(\text{int} \to \text{answer\_type}) \to \text{answer\_type}\]

– Example: CPS(73) = (λk. k 73)

– Convert entire program this way and then “main” is some (λk.e) that you can call with (λx.x)
Without further ado [but slowly 😊]

A call-by-value CPS transformation for this source language

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

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

\[ CPS(c) = \lambda k.\ k\ c \]
\[ CPS(x) = \lambda k.\ k\ x \quad (\text{any } k \neq x) \]
\[ CPS(\lambda x.\ e) = \lambda x.\ CPS(\ e) \]
\[ \quad \text{or } \lambda x.\ \lambda k.\ CPS(\ e)\ k \quad (\text{any } k \text{ not in } \text{FV}(e)) \]
\[ CPS(e_1 + e_2) = \lambda k.\ CPS(e_1) \]
\[ \quad (\lambda x_1.\ CPS(e_2)) \]
\[ \quad (\lambda x_2.\ k\ (x_1 + x_2)) \]  
\[ CPS(e_1\ e_2) = \lambda k.\ CPS(e_1) \]
\[ \lambda f.\ CPS(e_2) \]
\[ \lambda x.\ f\ x\ k \quad (\text{why not } k\ (f\ x)?) \]
Everything is a tail-call

• For all $e$, CPS($e$) is in this sublanguage and stays in it during evaluation:

  $e ::= a \mid a\ a \mid a\ a\ a \mid a\ (a\ +\ a)$

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

• An interpreter for the target of CPS doesn’t need a call-stack because every call is a tail-call

• Essentially, the program itself is encoding the conceptual call-stack in nested continuations (lambdas bound to $k$ variables)
Programming this way

• Even if your compiler doesn’t use the CPS transform, you can program directly (“manually”) in CPS (a “style” or “idiom”)
  – So you are manually using only tail-calls by using “clever” (but mechanical) lambdas for continuations
  – Moves “deep recursion” from the stack to the heap

• See examples in cps_examples.ml
Back to first-class continuations

• Next “amazing” thing: If we add (back) letcc and throw:
  – CPS(e) works fine
  – It “compiles away” letcc and throw to constant-time operations (!!)
  – “The continuations” are just lambdas bound to variables

• See next slide…
CPS transformation for continuations

• Old news:

\[
\begin{align*}
\text{CPS}(c) &= \lambda k. \ k \ c \\
\text{CPS}(x) &= \lambda k. \ k \ x \quad \text{(any k \neq x)} \\
\text{CPS}(\lambda x. e) &= \lambda x. \ \text{CPS}(e) \quad \text{or} \quad \lambda x. \ \lambda k. \ \text{CPS}(e) \ k \\
\text{CPS}(e_1 \ e_2) &= \lambda k. \ \text{CPS}(e_1) \ (\lambda f. \ \text{CPS}(e_2) \ (\lambda x. \ f \ x \ k))
\end{align*}
\]

• Now:

\[
\begin{align*}
\text{CPS(\text{letcc my_k}. e)} &= \lambda \text{my_k}. \ \text{CPS}(e) \ \text{my_k}
\end{align*}
\]

\[
\begin{align*}
\text{CPS(\text{throw e1 e2})} &= \lambda k. \ \text{CPS}(e_1) \ \text{CPS}(e_2) \quad \text{(doesn’t use k!!)} \\
\text{(easier to understand but verbose:)} \\
\lambda k. \ \text{CPS}(e_1) \ (\lambda f. \ \text{CPS}(e_2) \ (\lambda x. \ f \ x))
\end{align*}
\]
Really small examples

The rule:

\[ CPS(\text{letcc } my_k.\ e) = \lambda my_k.\ CPS(e)\ my_k \]

Example #1:

\[ CPS(\text{letcc } my_k.\ 42) = \lambda my_k.\ (\lambda k.\ k\ 42)\ my_k \]

Example #2:

\[ CPS(\text{letcc } my_k.\ my_k) = \lambda my_k.\ (\lambda k.\ k\ my_k)\ my_k \]
Back to programming

• You can use this idea in “manual” CPS too

• See OCaml example for “fast-escape from recursion”
  – Same idea for exceptions
    • And a compiler using CPS can implement exceptions this way
  – Time travel works too [not shown]
Another “real-world” use

• A great way to think about some of web programming
  – Each step in a web session is an evaluation context
    send(page1);
    receive(form_input);
    if ... then send(page2); ... send(page3); ...
  – But want to program in “direct style” and have the different steps be automatically “checkpointed”
    • To support the back button and session saving
    • Compile program into something using continuations
    • Then encode continuation in a URL or some other hack
Where are we

Finished major parts of the course
• Functional programming
• IMP, loops, modeling mutation
• Lambda-calculus, modeling functions
• Formal semantics
• Contexts, continuations

A mix of super-careful definitions for things you know and using our great care to describe more novel things (state monad, continuations)

Major new topic: Types!
– Continue using lambda-calculus as our model
– But no need to understand continuations for rest of lecture
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 "hi" ^ "mom"
   - But may be too early (code not used, …)

2. “Safety”: Prevent getting stuck / going haywire
   - *Know* evaluation cannot ever get to the point where the next step “makes no sense”
   - Alternative: language makes everything make sense
     - Example: `ClassCastException`
     - Example: `MethodNotFoundException`
     - Example: 3 + "hi" becomes "3hi" or 0
   - Alternative: 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 H2;s2!):

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

\[
\begin{align*}
H; e_1[i] = e_2 \downarrow & \quad H; s_2
\end{align*}
\]

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
   • Requires safety

4. Allow faster implementations
   • Smaller interfaces enable optimizations
   • Don’t have to check for impossible cases
   • Orthogonal to safety

5. Static overloading (e.g., with +)
   • Not super 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, …
   • Deep similarities among different analyses suggests types are a good way to think about and define what you’re checking

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 ($\exists$)
- Type inference (maybe)
Adding integers to the lambda-calculus:

Expressions: \( e ::= x | \lambda x. e | e e | c \)

Values: \( v ::= \lambda x. e | c \)

Could add + and other primitives or just parameterize “programs” by them: \( \lambda plus. \lambda minus. \ldots e \)

– Like Pervasives in OCaml
– A great idea for keeping language definitions small
Stuck

• Key issue: can a program e “get stuck” (small-step):
  – e →* e1
  – e1 is not a value
  – There is no e2 such that e1 → e2

• “What is stuck” depends on the semantics:

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

- \( S ::= c \ v \mid x \ v \mid (\lambda x. e) \ y \mid S \ e \mid (\lambda x. e) \ S \)

- It’s unusual to define these explicitly, but good for understanding.

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

- With \( e_1 + e_2 \), would also be stuck when:
  - \( e_1 \) or \( e_2 \) is itself stuck
  - \( e_1 \) or \( e_2 \) is a lambda
  - \( e_1 \) or \( e_2 \) is a variable
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: \[ \Gamma \vdash e : \tau \]

(for which we “hope” there’s an efficient algorithm)

\[ \begin{array}{c}
\Gamma \vdash c : \text{int} \\
\Gamma \vdash (\lambda x. e) : \text{function} \\
\Gamma \vdash e_1 : \text{function} \\
\Gamma \vdash e_2 : \text{int} \\
\hline
\end{array} \]

\[ \Gamma \vdash e_1 \ e_2 : \text{int} \]
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 int
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): \( \Gamma ::= \cdot \mid \Gamma, x: \tau \text{ and } \Gamma \vdash e : \tau \)
   
   - 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 OCaml
Examples and syntax

- Examples of types
  \[
  \text{int} \rightarrow \text{int} \\
  (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \\
  \text{int} \rightarrow (\text{int} \rightarrow \text{int})
  \]

- Concretely \(\rightarrow\) is right-associative
  - i.e., \(\tau_1 \rightarrow \tau_2 \rightarrow \tau_3\) is \(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)\)
  - Just like OCaml
STLC in one slide

Expressions: \( e ::= x \mid \lambda x.\ e \mid e\ e \mid c \)

Values: \( v ::= \lambda x.\ e \mid c \)

Types: \( \tau ::= \text{int} \mid \tau \rightarrow \tau \)

Contexts: \( \Gamma ::= . \mid \Gamma, x : \tau \)

\[
\begin{array}{ccc}
\text{e} & \rightarrow & \text{e}' \\
\hline
\text{e}_1 \ e_2 & \rightarrow & \text{e}_1' \ \text{e}_2 \\
\text{v} \ e_2 & \rightarrow & \text{v} \ e_2' \\
(\lambda x.\ e) \ v & \rightarrow & e\{v/x\}
\end{array}
\]

\[
\begin{array}{cc}
\Gamma \vdash e : \tau \\
\hline
\Gamma \vdash c : \text{int} \\
\Gamma \vdash x : \Gamma(x)
\end{array}
\]

\[
\begin{array}{ccc}
\Gamma, x : \tau_1 \vdash e : \tau_2 \\
\hline
\Gamma \vdash (\lambda x.\ e) : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash e_1 \ e_2 : \tau_2
\end{array}
\]