Remember our symbol-pile

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

\[ e \ | \ v \]

\[ e_1 \downarrow \lambda x. \ e_3 \]
\[ e_2 \downarrow v_2 \]
\[ e_3(v2x) \downarrow v \]
\[ \lambda x. \ e \downarrow \lambda x. \ e \]
\[ e_1 \ e_2 \downarrow v \]

\[ e_3(v2x) \] is the "capture-avoiding substitution of \( v_2 \) for \( x \) in \( e_3 \)"
- Capture is an insidious error in program rewriters
- Formally avoided via "systematic renaming (alpha conversion)"
  - Ensure free variables in \( v_2 \) are not binders in \( e_3 \)

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

In OCaml

```
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)
```

Small-step CBV

- Left-to-right small-step judgment

\[
\begin{array}{c}
  e_1 \rightarrow e'_1 \\
  e_2 \rightarrow e'_2 \\
  e_1 e_2 \rightarrow e'_1 e_2 \\
  v e_2 \rightarrow v e'_2 \\
  (\lambda x. \ e) v \rightarrow e(v/x)
\end{array}
\]

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

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

\[
\text{ectx} = \text{Hole} \\
| \text{ALeft \ of \ ctx} \ \ast \ \text{exp} \\
| \text{ARight \ of \ exp} \ \ast \ \text{ctx} \ (* \ \text{exp \ a \ value*)}
\]

\[
\begin{align*}
\text{let rec split e = (* \ \text{return \ ctx \ \& \ \text{what’s in it}*)} \\
& \quad \text{match \ e \ with} \\
& \quad \quad A(L(s1,e1),L(s2,e2)) \rightarrow (\text{Hole}, e) \\
& \quad \quad A(L(s1,e1),e2) \rightarrow \text{let \ (ctx2, e3) = split \ e2 in} \\
& \quad \quad \quad \text{(ARight}(L(s1,e1),\text{ctx2}), e3) \\
& \quad \quad A(e1,e2) \rightarrow \text{let \ (ctx1, e3) = split \ e1 in} \\
& \quad \quad \quad \text{(ALeft}(\text{ctx1,e3}), e3) \\
& \quad \quad _ \rightarrow \text{raise \ BadArgument}
\end{align*}
\]

Fill a context

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

\[
\begin{align*}
\text{let rec fill \ ctx \ e = (* \ \text{plug the hole} *)} \\
& \quad \text{match \ ctx \ with} \\
& \quad \quad \text{Hole} \rightarrow \ e \\
& \quad \quad \text{ALeft}(\text{ctx2},e2) \rightarrow \text{A}(\text{fill \ ctx2} \ e, e2) \\
& \quad \quad \text{ARight}(e2,\text{ctx2}) \rightarrow \text{A}(e2, \text{fill \ ctx2} \ e)
\end{align*}
\]

So what?

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

\[
\begin{align*}
\text{let rec interp_small e =} \\
& \quad \text{match \ e \ with} \\
& \quad \quad \text{V v} \rightarrow \text{failwith} \ “\text{interp_small}”(*\text{unbound \ var}*) \\
& \quad \quad L \rightarrow e \\
& \quad \quad A \rightarrow \text{match \ split \ e \ with} \\
& \quad \quad \quad \text{(ctx, A(L(s3,e3),v)) \ \rightarrow} \\
& \quad \quad \quad \quad \text{interp_small(fill \ ctx \ (subst \ e3 \ v \ s3))} \\
& \quad \quad \quad _ \rightarrow \text{failwith} \ “\text{bad \ split}”
\end{align*}
\]

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)

\[
\begin{align*}
\text{let \ ctx = Hole \ (* \ \text{no \ change} \ *)} \\
| \text{ALeft \ of \ ctx} \ \ast \ \text{exp} \\
| \text{ARight \ of \ exp} \ \ast \ \text{ctx} \ (* \ \text{no \ change} \ *)
\end{align*}
\]
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 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(s,e) -> (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”

Toy Examples

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

1. $1 + (\text{letcc } k. 2 + 3)$ → $6$
2. $1 + (\text{letcc } k. 2 + (\text{throw } k 3))$ → $4$
3. $1 + (\text{letcc } k. (\text{throw } k (2+3)))$ → $6$

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

```
letcc k. (throw k 1) + (throw k 2)
```

Example Uses

- Continuations for exceptions is “easy”
  - $\text{Letcc}(x,e)$ for try, $\text{Apply}(\text{Var } x, v)$ 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

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$((\lambda x. x))$ 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 \texttt{letcc} and \texttt{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 \texttt{int} expression becomes an
  \[ (\texttt{int} \to \texttt{answer\_type}) \to \texttt{answer\_type} \]
- Example: CPS$((73)) = (\lambda x. \texttt{73})$
- Convert entire program this way and then "main" is some \((\lambda k. e)\) that you can call with \((\lambda x. x)\)

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

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

\[ \text{CPS}(c) = \lambda k. k \ c \]
\[ \text{CPS}(x) = \lambda k. k \ x \quad (\text{any } k \neq x) \]
\[ \text{CPS}(\lambda x. e) = \lambda x. \text{CPS}(e) \quad \text{(any } k \text{ not in } FV(e)) \]
\[ \text{CPS}(e_1 + e_2) = \lambda k. \text{CPS}(e_1) \quad (\text{any } k, x_1 \text{ not in } FV(e_1 + e_2)) \]
\[ \quad \lambda f. \text{CPS}(e_2) \quad \lambda x. f \ x \ k \quad (\text{why not } k \ (f \ x)?) \]

Without further ado [but slowly 😎]

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

- 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 \texttt{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:
  CPS(c) = λk. k c
  CPS(x) = λk. k x (any k != x)
  CPS(λx. e) = λx. CPS(e) or λk. CPS(e) k
  CPS(e1 e2) = λk. CPS(e1) (λf. CPS(e2) (λx. f x))
• Now:
  CPS(letcc my_k. e) = λmy_k. CPS(e) my_k
  CPS(throw e1 e2) = λk. CPS(e1) CPS(e2) (doesn't use k!!)
  (easier to understand but verbose:

  λk. CPS(e1) (λf. CPS(e2) (λx. f x)) )

Really small examples

The rule:

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

Example #1:

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

Example #2:

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

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;e1 ⊸ [v1, ..., vn] H;e2 ⊸ i i > n
```

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?)

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 (‘t’), Abstract types (3)
- Type inference (maybe)
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: \( \text{plus}, \text{minus}, \ldots, e \)

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

Stuck

• Key issue: can a program \( e \) “get stuck” (small-step):
  
  \( e \rightarrow^* e_1 \)
  
  – \( e_1 \) is not a value
  
  – There is no \( e_2 \) such that \( e_1 \rightarrow e_2 \)

• “What is stuck” depends on the semantics:

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

STLC Stuck

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

• 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

A judgment: \( \left\{ \begin{array}{c} e : \tau \\ c : \text{int} \\ (\lambda x. e) : \text{function} \\ e_1 : \text{function} \\ e_2 : \text{int} \end{array} \right\} \)

(for which we “hope” there’s an efficient algorithm)
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): \(\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma \vdash e : \tau\)
   - A type-checking environment (called a context)

For (2): \(\tau ::= \int \mid \tau \rightarrow \tau\)
   - Arrow is part of the (type) language (not meta)
   - An infinite number of types
   - Just like OCaml

Examples and syntax

- Examples of types
  - \(\int \rightarrow \int\)
  - \((\int \rightarrow \int) \rightarrow \int\)
  - \(\int \rightarrow (\int \rightarrow \int)\)

- 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: \(e ::= x | \lambda x. e | e e | c\)
Values: \(v ::= \lambda x. e | c\)
Types: \(\tau ::= \int \mid \tau \rightarrow \tau\)
Contexts: \(\Gamma ::= \cdot \mid \Gamma, x : \tau\)

\[
\begin{align*}
\Gamma \vdash e : \tau \\
\Gamma \vdash c : \int \quad \Gamma \vdash x : \Gamma(x) \\
\Gamma, x : \tau \vdash e : \tau \\
\Gamma \vdash e1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e2 : \tau_1 \\
\Gamma \vdash (\lambda x. e) : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e1 \ e2 : \tau_2 \\
\end{align*}
\]