#### CSEP505: Programming Languages Lecture 5: Continuations, Types

# Dan Grossman

Autumn 2016

#### Remember our symbol-pile

Expressions: e ::= Values: v ::=	$\boldsymbol{x} \mid \lambda \boldsymbol{x}. \boldsymbol{e} \mid \boldsymbol{e} \boldsymbol{e}$ $\lambda \boldsymbol{x}. \boldsymbol{e}$
e↓v	
	$e1 \checkmark \lambda x. e3 e2 \checkmark v2 e3\{v2/x\} \checkmark v$
[lam] λx. e ↓λx. e	e1 e2 ↓ v [app]
Capture is an insidio	avoiding substitution of $v2$ for x in $e3$ " us error in program rewriters

Formally avoided via "systematic renaming (alpha conversion)"
 Ensure free variables in v2 are not binders in e3

CSE P505 Autumn 2016 Dan Grossman

# 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:

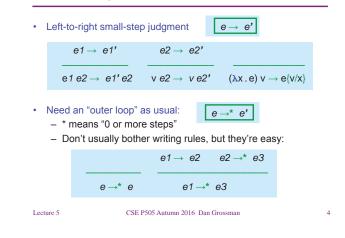
- Small-step
- Play with continuations ("very fancy" language feature)

#### Then: On to types

Lecture 5	CSE P505 Autumn 2016 Dan Grossman

# Small-step CBV

Lecture 5



# 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*)
 L
 | 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
 L
 | A(e1,e2) -> interp small (interp one e)
                  CSE P505 Autumn 2016 Dan Grossman
Lecture 5
                                                         5
```

#### Unrealistic, but...

- For all e and v,
  - $e \checkmark v$  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!]

```
Lecture 5
```

2

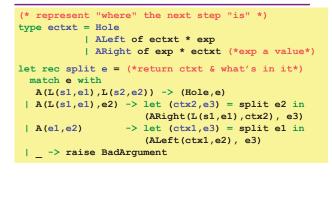
# Rethinking small-step

- 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

#### Lecture 5

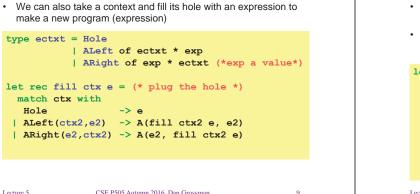
CSE P505 Autumn 2016 Dan Grossman

#### Compute the context



CSE P505 Autumn 2016 Dan Grossman

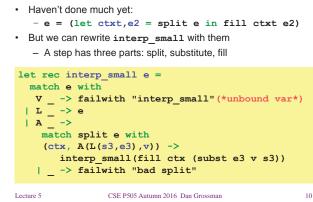
Fill a context



7

### So what?

Lecture 5



#### 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"

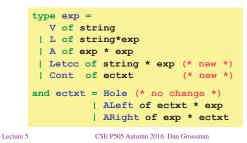
Lecture 5

CSE P505 Autumn 2016 Dan Grossman

```
11
```

# 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)



8

#### Split with Letcc

- Old: All values were some L(s,e)
- New: Values can also be Cont c
- Old: active expression (thing in the hole) always some
   A(L(s1,e1),L(s2,e2))
- New: active expression (thing in the hole) can be:
  - A(v1,v2)
  - Letcc(s,e)
- So split looks quite different to implement these changes
   Not really that different
- fill does not change at all

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

#### Split with Letcc

```
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 el
   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"
                                                    14
Lecture 5
                CSE P505 Autumn 2016 Dan Grossman
```

#### All the action

- Letcc creates a Cont that "grabs the current context"
- A where body is a Cont "ignores current context"

<pre>let rec interp_small e =   match e with   V&gt; failwith "interp_small" (*unbound var*)     L&gt; e    &gt; match split e with    (ctx, A(L(s3,e3), v)) -&gt;       interp_small(fill ctx (subst e3 v s3))    (ctx, Letcc(s3,e3)) -&gt;       interp_small(fill ctx         (*woah!!!*) (subst e3 (Cont ctx) s3))    (ctx, A(Cont ctx2, v)) -&gt;       interp_small(fill ctx2 v) (*woah!!!*)    &gt; failwith "bad split"</pre>	
Lecture 5 CSE P505 Autumn 2016 Dan Grossman 1	15

#### **Toy Examples**

[In language with addition too and explicit "throw"]

1 + (letcc k. 2 + 3)  $\rightarrow^*$  6

1 + (letcc k. 2 + (throw k 3))  $\rightarrow^* 4$ 

1 + (letcc k. (throw k (2+3)))  $\rightarrow^* 6$ 

1 + (letcc k. (throw k (throw k 2)))  $\rightarrow^* 3$ 

Also note evaluation-order matters, even without mutation (!) letcc k. (throw k 1) + (throw k 2)

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

#### 16

#### **Example Uses**

- Continuations for exceptions is "easy"
   Letcc(x,e) for try, 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"...

Lecture 5

17

13

#### "Time Travel"

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

```
Lecture 5
```

CSE P505 Autumn 2016 Dan Grossman

#### 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
  - 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) Letcc
  - You do not actually split/fill at each step
  - Cannot just do 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...

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

# 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...

Lecture 5

19

21

CSE P505 Autumn 2016 Dan Grossman

20

#### **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 (int -> answer\_type) -> answer\_type
- Example:  $CPS(73) = (\lambda k. k 73)$
- Convert entire program this way and then "main" is some  $(\lambda \mathbf{k} \cdot \mathbf{e})$  that you can call with  $(\lambda \mathbf{x} \cdot \mathbf{x})$

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

# 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   <i>c</i>   e + e	
Values: v	$::= \lambda x. e \mid c$		
$CPS(c) = \lambda k \cdot k c$			
$CPS(\mathbf{x}) = \lambda k \cdot k \mathbf{x}$	(any k != x)		
$CPS(\lambda x. e) = \lambda x. C$	PS(e)		
or $\lambda x$	$\lambda k$ . CPS(e) k	(any k not in FV(e))	
$CPS(e1 + e2) = \lambda k$	k. CPS(e1)	(any k,x1 not in FV(e1+e2))	
	(λx1. CPS(e2)		
	$(\lambda x^2)$ . k	(x1 + x2)))	
$CPS(e1 \ e2) = \lambda k$ .	CPS(e1)	(any k,f not in FV(e1 e2))	
	<b>λf.</b> CPS(e2)		
	$\lambda x. f x k$	(why not k (f x)?)	
Lecture 5	CSE P505 Autumn 2016 Dan G	rossman 22	

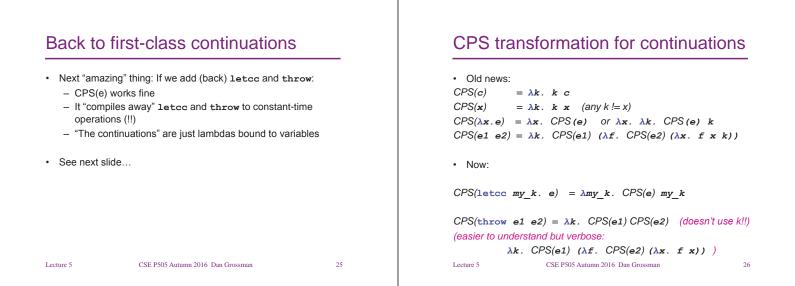
#### Everything is a tail-call

- For all e, CPS(e) is in this sublanguage and stays in it during evaluation:
  - e ::= a | a a | a a a | a (a + a) $a ::= x | \lambda x. e | c$
- 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 callstack 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

23



#### Really small examples

The rule: $CPS(\texttt{letcc } my\_k. e) = \lambda my\_k. CPS(e) my\_k$ Example #1: $CPS(\texttt{letcc } my\_k. 42) = \lambda my\_k. (\lambda k. k 42) my\_k$ Example #2: $CPS(\texttt{letcc } my\_k. my\_k) = \lambda my\_k. (\lambda k. k my\_k) my\_k$		<ul> <li>You can use this idea in "manual" CPS too</li> <li>See OCaml example for "fast-escape from recursion" <ul> <li>Same idea for exceptions</li> <li>And a compiler using CPS can implement exceptions this way</li> <li>Time travel works too [not shown]</li> </ul> </li> </ul>			
Lecture 5 CSE P505 Autumn 2016 Dan Grossman	27	Lecture 5	CSE P505 Autumn 2016 Dan Grossman	28	

Back to programming

30

# Another "real-world" use

Another "real-world" use			Where are we			
- Each s sen rec if - But wa steps t • To • Co	ay to think about some of web programming step in a web session is an evaluation conternation (page1); metive(form_input); then send(page2); send(page3) ant to program in "direct style" and have the be automatically "checkpointed" support the back button and session saving mpile program into something using continu en encode continuation in a URL or some of	xt 9); different 1 ations	<ul> <li>Functional</li> <li>IMP, loops</li> <li>Lambda-cc</li> <li>Formal set</li> <li>Contexts, our great or continuation</li> <li>Major new top – Continuation</li> </ul>	continuations uper-careful definitions for things you know care to describe more novel things (state m ons)	ionad,	
Lecture 5	CSE P505 Autumn 2016 Dan Grossman	29	Lecture 5	CSE P505 Autumn 2016 Dan Grossman	3	

### **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?

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

#### Why types

- 1. Catch "stupid mistakes" early
  - 3 + "hello"

Lecture 5

- 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

CSE P505 Autumn 2016 Dan Grossman

Alternative: language can do whatever ?!

Digression/sermon

Unsafe languages have operations where under some situations 3. Enforce a strong interface (via an abstract type) the implementation "can do anything' Clients can't break invariants Clients can't assume an implementation IMP with unsafe C arrays has this rule (any H2;s2!): . Requires safety 4. Allow faster implementations H;e2 🕹 i H;e1 ↓ {v1,...,vn} i > n Smaller interfaces enable optimizations Don't have to check for impossible cases H; e1[i]=e2 ↓ H2;s2 Orthogonal to safety 5. Static overloading (e.g., with +) Abstraction, modularity, encapsulation are impossible because one Not super interesting bad line can have arbitrary global effect Late-binding very interesting (come back to this?) • An engineering disaster (cf. civil engineering) Lecture 5 CSE P505 Autumn 2016 Dan Grossman 33 Lecture 5 CSE P505 Autumn 2016 Dan Grossman 34

31

#### 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)

#### Lecture 5

35

Lecture 5

Our plan

Subtyping

Type Variables:

Type inference (maybe)

Simply-typed lambda-calculus

Safety = (preservation + progress)

Digression: static vs. dynamic typing

Generics (∀), Abstract types (∃)

Digression: Curry-Howard Isomorphism

Extensions (pairs, datatypes, recursion, etc.)

32

#### Why types, continued

#### Adding integers

Adding integers to the lambda-calculus:

Expressions:  $\mathbf{e} ::= \mathbf{x} \mid \lambda \mathbf{x}. \mathbf{e} \mid \mathbf{e} \mathbf{e} \mid \mathbf{c}$ Values:  $\mathbf{v} ::= \lambda \mathbf{x}. \mathbf{e} \mid \mathbf{c}$ 

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

CSE P505 Autumn 2016 Dan Grossman

- 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  $\rightarrow$  e2
- · "What is stuck" depends on the semantics:

e1 $\rightarrow$ e1'	$e2 \rightarrow \ e2'$	
$e1 e2 \rightarrow e1'e2$	$v e2 \rightarrow v e2$ '	$\overbrace{(\lambda x.e)}^{}v\to e\{v/x\}$

Lecture 5

37

39

CSE P505 Autumn 2016 Dan Grossman

38

#### STLC Stuck

Lecture 5

- $S ::= c v | x v | (\lambda x.e) y | S e | (\lambda x.e) S$
- · 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 e1 + e2, would also be stuck when:
  - e1 or e2 is itself stuck
  - e1 or e2 is a lambda
  - e1 or e2 is a variable

Lecture 5

CSE P505 Autumn 2016 Dan Grossman

#### 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

```
Lecture 5
```

1

(

CSE P505 Autumn 2016 Dan Grossman

#### 40

#### What to do

•	Old conclusion: "strong types for weak minds"
	<ul> <li>Need an unchecked cast (a back-door)</li> </ul>

- Modern conclusion:
  - Make false positives rare and false negatives impossible (be sound and expressive)
  - Make workarounds reasonable
  - Justification: false negatives too expensive, have compiletime resources for "fancy" type-checking
- Okay, let's actually try to do it...

#### Lecture 5

#### 41

# Wrong attempt

		•	t ::=	= int	1	funct	tion	
A judgme	ent:	⊦e :	τ					
for whic	h we	e "hope" t	here	's an	effic	cient al	gorith	m)
	-	-c : i	.nt		- (2	x.e)	:fund	cti

- e1	:	function	- e2	:	int

- e1 e2 : int

Lecture 5	CSE P505 Autumn 2016	Dan Grossman

on

