CSEP505: Programming Languages

Lecture 4: Untyped lambda-calculus, inference rules, environments, …

Dan Grossman
Spring 2006
Interesting papers?

Reading relevant research papers is great! But:

• Much of what we’ve done so far is a modern take on ancient (60s-70s) ideas (necessary foundation)
  – Few recent papers are on-topic & accessible
  – And old papers harder to find and read

• But I found some fun ones…
Interesting papers?

• Role of formal semantics “in practice”
  – The essence of XML [Siméon/Wadler, POPL03]
• Encodings and “too powerful” languages
  – C++ Templates as Partial Evaluation [Veldhuizen, PEPM99]
• Relation of continuations to web-programming (CGI)
  – The influence of browsers on evaluators or, continuations to program web servers [Queinnec, ICFP00]
  – Automatically Restructuring Programs for the Web [Graunke et al., ASE01]
Lambda-calculus

• You cannot properly model local scope with a global heap of integers
  – Functions are not syntactic sugar for assignments
  – You need some stack or environment or substitution or ...

• So let’s build a model with functions & only functions

• Syntax of untyped lambda-calculus (from the 1930s)
  Expressions: $e ::= x \mid \lambda x. \ e \mid e \ e$
  Values: $v ::= \lambda x. \ e$
That’s all of it!

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

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

A program is an \( e \). To call a function:

substitute the argument for the bound variable

Example substitutions:

\[
(\lambda x.\ x) (\lambda y.\ y) \rightarrow \lambda y.\ y
\]

\[
(\lambda x.\ \lambda y.\ y\ x) (\lambda z.\ z) \rightarrow \lambda y.\ y\ (\lambda z.\ z)
\]

\[
(\lambda x.\ x\ x) (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x) (\lambda x.\ x\ x)
\]

Definition is subtle if the 2\textsuperscript{nd} value has “free variables”
Why substitution

• After substitution, the bound variable is *gone*, so clearly its name did not matter
  – That was our problem before

• Using substitution, we can define a tiny PL
  – Turns out to be Turing-complete
Full large-step interpreter

type exp = Var of string
  | Lam of string*exp
  | Apply of exp * exp

extinction BadExp

let subst e1_with e2_for x = ...(*to be discussed*)

let rec interp_large e =
  match e with
  Var _ -> raise BadExp(*unbound variable*)
  | Lam _ -> e (*functions are values*)
  | Apply(e1,e2) ->
    let v1 = interp_large e1 in
    let v2 = interp_large e2 in
    match v1 with
    Lam(x,e3) -> interp_large (subst e3 v2 x)
    _ -> failwith “impossible” (* why? * )
Interpreter summarized

- Evaluation produces a value
- Evaluate application (call) by
  1. Evaluate left
  2. Evaluate right
  3. Substitute result of (2) in body of result of (1)
     - and evaluate result

A different semantics has a different evaluation strategy:
  1. Evaluate left
  2. Substitute right in body of result of (1)
     - and evaluate result
Another interpreter

type exp = Var of string
           | Lam of string*exp
           | Apply of exp * exp
exception BadExp
let subst e1_with e2_for x = ...(to be discussed*)
let rec interp_large2 e =
  match e with
  Var _ -> raise BadExp(*unbound variable*)
  | Lam _ -> e (*functions are values*)
  | Apply(e1,e2) ->
    let v1 = interp_large2 e1 in
    (* we used to evaluate e2 to v2 here *)
    match v1 with
    Lam(x,e3) -> interp_large2 (subst e3 e2 x)
    _   -> failwith “impossible” (* why? * )
What have we done

• Gave syntax and two large-step semantics to the untyped lambda calculus
  – First was “call by value”
  – Second was “call by name”
• Real implementations don’t use substitution; they do something equivalent
• Amazing (?) fact:
  – If call-by-value terminates, then call-by-name terminates
  – (They might both not terminate)
What will we do

• 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
• Play with continuations (very fancy language feature)
Syntax notes

• When in doubt, put in parentheses

• Math (and Caml) resolve ambiguities as follows:

1. $\lambda x. e_1 \ e_2$ is $(\lambda x. e_1 \ e_2)$, not $(\lambda x. e_1) \ e_2$

General rule: Function body “starts at the dot” and “ends at the first unmatched right paren”

Example:

$(\lambda x. \ y \ (\lambda z. \ z) \ w) \ q$
Syntax notes

2. \( e_1 \ e_2 \ e_3 \) \ is \ (e_1 \ e_2) \ e_3, \ \textbf{not} \ e_1 \ (e_2 \ e_3) \)

General rule: Application “associates to the left”

So \( e_1 \ e_2 \ e_3 \ e_4 \) \ is \ (((e_1 \ e_2) \ e_3) \ e_4) \)
It’s just syntax

• As in IMP, we really care about abstract syntax
  – Here, internal tree nodes labeled “\(\lambda\)” or “app”

• The previous two rules just cut down on parens when writing trees as strings

• Rules may seem strange, but they’re the most convenient (given 70 years experience)
What will we do

• 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
• Play with continuations (very fancy language feature)
Inference rules

• A metalanguage for operational semantics
  – Plus: more concise (& readable?) than Caml
  – Plus: useful for reading research papers
  – Plus?: natural support for nondeterminism
  – Minus: Less tool support than Caml (no compiler)
  – Minus: one more thing to learn
  – Minus: Painful in Powerpoint

• Without further ado:
Large-step CBV

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

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

\[
\vdots
\]

\[
e_1 e_2 \downarrow v
\]

- Green is metanotation here (not in general)
- Defines a set of pairs: exp * value
  - Using definition of a set of 4-tuples for substitution (exp * value * variable * exp)
Some terminology

λx. e

General set-up:

1. A judgment, (here e ▼ v, pronounced “e goes to v”)
   • Metasyntax is your choice
   • Prefer interp(e, v)?
   • Prefer « v e »?

2. Inference rules to specify which tuples are in the set
   • Here two (names just for convenience)
Using inference rules

An inference rule is “premises over conclusion”
- “To show the bottom, show the top”
- Can “pronounce” as a proof or an interpreter

To “use” an inference rule, we “instantiate it”
- Replace metavariables consistently
Derivations

- Tuple is “in the set” if there exists a derivation of it
  - An upside-down (or not?!) tree where each node is an instantiation and leaves are axioms (no premises)
- To show $e \Downarrow v$ for some $e$ and $v$, *give a derivation*
  - But we rarely “hand-evaluate” like this
  - We’re just defining a semantics remember
Summary so far

- Judgment via inference rules
- Tuple in the set ("judgment holds") if a derivation (tree of instantiations ending in axioms) exists

As an interpreter, could be “non-deterministic”:
- Multiple derivations, maybe multiple $v$ such that $e \Downarrow v$
  - Our example is deterministic
  - In fact, “syntax directed” ($\leq 1$ rule per syntax form)
- Still need rules for $e\{v/x\}$

- Let’s do more judgments to get the hang of it…
CBN large-step

\[ e_1 \downarrow_N \lambda x. e_3 \quad e_3\{e_2/x\} \downarrow_N v \]

\[ \lambda x. e \downarrow_N \lambda x. e \quad e_1 e_2 \downarrow_N v \]

- Easier to see the difference than in Caml
- Formal statement of amazing fact:
  For all \( e \), if there exists a \( v \) such that \( e \downarrow v \) then there exists a \( v_2 \) such that \( e \downarrow_N v_2 \)

(Proof is non-trivial & must reason about substitution)
Lambda-syntax

- Inference rules more powerful & less concise than BNF
  - Must know both to read papers

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

\[
\begin{array}{c|c|c}
\text{isexp}(x) & \text{isexp}(e1) & \text{isexp}(\lambda x. \ e) \\
\text{isexp}(e) & \text{isexp}(e1 \ e2) & \text{isvalue}(\lambda x. \ e)
\end{array}
\]
IMP

• Two judgments $H;e \downarrow i$ and $H;s \downarrow H2$
• Assume $\text{get}(H,x,i)$ and $\text{set}(H,x,i,H2)$ are defined
What will we do

• 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
• Play with continuations (very fancy language feature)
Encoding motivation

• It’s fairly crazy we omitted integers, conditionals, data structures, etc.
• Can encode whatever we need. Why do so:
  – It’s fun and mind-expanding.
  – It shows we didn’t oversimplify the model (“numbers are syntactic sugar”)
  – It can show languages are too expressive
    Example: C++ template instantiation
Encoding booleans

- There are 2 bools and 1 conditional expression
  - Conditional takes 3 (curried) arguments
  - If 1st argument is one bool, return 2nd argument
  - If 1st argument is other bool, return 3rd argument
- Any 3 expressions meeting this specification (of “the boolean ADT”) is an encoding of booleans
- Here is one (of many):
  - “true” \( \lambda x. \lambda y. x \)
  - “false” \( \lambda x. \lambda y. y \)
  - “if” \( \lambda b. \lambda t. \lambda f. b \ t \ f \)
Example

• Given our encoding:
  – “true” $\lambda x. \lambda y. x$
  – “false” $\lambda x. \lambda y. y$
  – “if” $\lambda b. \lambda t. \lambda f. b \; t \; f$
• We can derive “if” “true” v1 v2 $\downarrow$ v1
• And every “law of booleans” works out
  – And every non-law does not
But…

• Evaluation order matters!
  – With ↓, our “if” is not YFL’s if

  “if” “true” (\(\lambda x. x\)) (\(\lambda x. x x\)) (\(\lambda x. x x\)) diverges
  but
  “if” “true” (\(\lambda x. x\)) (\(\lambda z. (\lambda x. x x) (\lambda x. x x) z\)) terminates

  – Such “thunking” is unnecessary using \(\downarrow_N\)
Encoding pairs

- There is 1 constructor and 2 selectors
  - 1st selector returns 1st argument to constructor
  - 2nd selector returns 2nd argument to constructor
- This does the trick:
  - “make_pair” \( \lambda x. \lambda y. \lambda z. z \ x \ y \)
  - “first” \( \lambda p. \ p \ (\lambda x. \lambda y. \ x) \)
  - “second” \( \lambda p. \ p \ (\lambda x. \lambda y. \ y) \)

- Example:
  “snd” (“fst” (“make_pair” (“make_pair” v1 v2) v3)) \( \downarrow \) v2
Digression

- Different encodings can use the same terms
  - “true” and “false” appeared in our pair encoding
- This is old news
  - All code and data in YFL is encoded as bit-strings
- You want to program “pre-encoding” to avoid misusing data
  - Even if post-encoding it’s “well-defined”
  - Just like you don’t want to write self-modifying assembly code
Encoding lists

• Why start from scratch? Build on bools and pairs:
  – “empty-list”
    “make_pair” “false” “false”
  – “cons”
    \( \lambda h. \lambda t. \text{“make_pair” “true” “make_pair” } h t \)
  – “is-empty”
  – “head”
  – “tail”

• (Not too far from how one implements lists.)
Encoding natural numbers

• Known as “Church numerals”
  – Will skip in the interest of time

• The “natural number” ADT is basically:
  – “zero”
  – “successor” (the add-one function)
  – “plus”
  – “is-equal”

• Encoding is correct if “is-equal” agrees with elementary-school arithmetic
Recursion

- Can we write *useful* loops? Yes!
  
To write a recursive function

- Have a function take \( f \) and call it in place of recursion:
  - Example (in enriched language):
    \[
    \lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } (x * f(x-1))
    \]

- Then apply “fix” to it to get a recursive function
  “fix” \( \lambda f. \lambda x. \text{if } x=0 \text{ then } 1 \text{ else } (x * f(x-1)) \)

- Details, especially in CBV are icky; but it’s possible and need be done only once. *For the curious:*
  “fix” is \( \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) \)
More on “fix”

• “fix” is also known as the Y-combinator

• The informal idea:
  – “fix” (\(\lambda f. e\)) becomes something like
    \[e\{\text{“fix” (\(\lambda f. e\)) / f}\}\]

  – That’s unrolling the recursion once
  – Further unrolling happen as necessary

• Teaser: Most type systems disallow “fix” so later we’ll add it as a primitive
What will we do

• 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
• Play with continuations (very fancy language feature)
Our goal

Need to define the “judgment” \( e_1 \{ e_2 / x \} = e_3 \)

- (“used” in app rule)
- Informally, “replace occurrences of \( x \) in \( e_1 \) with \( e_2 \)”

Wrong attempt #1

\[
\begin{align*}
  y \neq x & \quad e_1 \{ e_2 / x \} = e_3 \\
  x \{ e / x \} = x & \quad y \{ e / x \} = y \\
  (\lambda y . e_1) \{ e_2 / x \} = \lambda y . e_3 \\
  ea \{ e_2 / x \} = ea' & \quad eb \{ e_2 / x \} = eb' \\
  (ea \ eb) \{ e_2 / x \} = ea' \ eb'
\end{align*}
\]
Try #2

<table>
<thead>
<tr>
<th>y != x</th>
<th>e1{e2/x} = e3</th>
<th>y!=x</th>
</tr>
</thead>
<tbody>
<tr>
<td>x{e/x} = x</td>
<td>y{e/x} = y</td>
<td>(λy . e1){e2/x} = λy . e3</td>
</tr>
<tr>
<td>ea{e2/x} = ea’</td>
<td>eb{e2/x} = eb’</td>
<td></td>
</tr>
<tr>
<td>(ea eb) {e2/x} = ea’ eb’</td>
<td></td>
<td>(λx . e1){e2/x} = λy . e1</td>
</tr>
</tbody>
</table>

- But what about (λy . e1){y/x} or (λy . e1){(λz.y/x)}
  - In general, if “y appears free in e2”
- Good news: this can’t happen under CBV or CBN
  - If program starts with no unbound variables
The full answer

• This problem is called capture. Avoidable…
• First define an expression’s “free variables” (braces here are set notation)
  – FV(x) = \{x\}
  – FV(\lambda y . e) = FV(e) – \{y\}
  – FV(e1 e2) = FV(e1) \cup FV(e2)
• Now require “no capture”:
  \[
  e1{e2/x} = e3 \quad y! = x \quad y \text{ not in } FV(e2)
  \]
  \[
  (\lambda y . e1){e2/x} = \lambda y . e3
  \]
Implicit renaming

- But this is a partial definition, until…
- We allow “implicit, systematic renaming” of any term
  - In general, we never distinguish terms that differ only in variable names
  - A key language principle
  - Actual variable choices just as “ignored” as parens
- Called “alpha-equivalence”
Back to equivalences

Last time we considered 3 equivalences.
• Capture-avoiding substitution gives correct answers:
  1. $$(\lambda x . e) = (\lambda y . e\{y/x\})$$ – actually the same term!
  2. $$(\lambda x . e) e2 = e\{e2/x\}$$ – CBV may need e2 terminates
  3. $$(\lambda x . e \ x) = e$$ – need e terminates

These don’t hold in most PLs
• But exceptions worth enumerating
• See purely functional languages like Haskell
  – “Call-by-need” the in-theory “best of both worlds”

Amazing fact: There are no other equivalences!
What will we do

• 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
• Play with continuations (very fancy language feature)
Where we’re going

• Done: large-step for untyped lambda-calculus
  – CBV and CBN
  – Infinite number of other “strategies”
  – Amazing fact: all partially equivalent!

• Now other semantics (all equivalent to CBV):
  – With environments (in Caml to prep for hw)
  – Basic small-step (easy)
  – Contextual semantics (similar to small-step)
  • Leads to precise definition of continuations
type exp = Var of string
    | Lam of string * exp
    | Apply of exp * exp

exception BadExp

let subst e1_with e2_for x = ...(*to be discussed*)

let rec interp_large e =
    match e with
    | Var _ -> raise BadExp(*unbound variable*)
    | Lam _ -> e (*functions are values*)
    | Apply(e1, e2) ->
        let v1 = interp_large e1 in
        let v2 = interp_large e2 in
        match v1 with
        | Lam(x, e3) -> interp_large (subst e3 v2 x)
        | _ -> failwith "impossible" (* why? *)
Environments

• Rather than substitute, let’s keep a map from variables to values
  – Called an environment
  – Like IMP’s heap, but immutable and 1 not enough
• So a program “state” is now exp + environment
• A function body is evaluated under the environment where it was defined (see lecture 1)!
  – Use closures to store the environment
No more substitution

```ml
type exp = Var of string
  | Lam of string * exp
  | Apply of exp * exp
  | Closure of string * exp * env

and env = (string * exp) list

let rec interp env e =
  match e with
  Var s -> List.assoc s env (* do the lookup *)
| Lam(s,e2) -> Closure(s,e2,env) (*store env!*)
| Closure _ -> e (* closures are values *)
| Apply(e1,e2) ->
    let v1 = interp env e1 in
    let v2 = interp env e2 in
    match v1 with
      Closure(s,e3,env2) -> interp((s,v2)::env2) e3
    | _ -> failwith "impossible"
```
Worth repeating

• A closure is a pair of code and environment
  – Implementing higher-order functions is not magic
    or run-time code generation
• An okay way to think about Caml
  – Like thinking about OOP in terms of vtables
• Need not store whole environment of course
  – See homework
What will we do

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

- Left-to-right small-step judgment $e \rightarrow e'$
- Need an “outer-loop” as usual (written $e \rightarrow^* v$)
  - $^*$ means “0 or more steps”

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

- But 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

```ocaml
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 interp_small with them
  - Shows a step has three parts: split, subst, fill

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

```plaintext
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 (*exp a value*)
```
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)

```ml
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”

```ml
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,
        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, thing 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 not use stacks (put frames in heap and share)
For examples

<table>
<thead>
<tr>
<th>lam</th>
<th>app</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x. e \downarrow \lambda x. e )</td>
<td>( e_1 \downarrow \lambda x. e )  ( e_2 \downarrow v_2 )  ( e_3[v_2/x] \downarrow v )  ( e_1 e_2 \downarrow v )</td>
</tr>
</tbody>
</table>
For examples

\[ e_1 \downarrow \lambda x. e_3 \quad e_2 \downarrow v_2 \quad e_3[v_2/x] \downarrow v \]

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

\[ e_1 \downarrow e_2 \downarrow v \]
For examples

\[ e_1 \downarrow \lambda x. e \quad e_2 \downarrow v_2 \quad e_3[v_2/x] \downarrow v \]

\[ \lambda x. e \downarrow \lambda x. e \quad e_1 e_2 \downarrow v \]
For examples

\[
\begin{align*}
\lambda x. \mathbf{e} \downarrow \lambda x. \mathbf{e}, & \quad \lambda x. \mathbf{e} \downarrow \lambda x. \mathbf{e}, \\
\mathbf{e}_1 \downarrow \lambda x. \mathbf{e}_3 & \quad \mathbf{e}_2 \downarrow \mathbf{v}_2 & \quad \mathbf{e}_3[\mathbf{v}_2/x] \downarrow \mathbf{v}, \\
\mathbf{e}_1 \mathbf{e}_2 \downarrow \mathbf{v} & \quad \mathbf{e}_1 \mathbf{e}_2 \downarrow \mathbf{v}
\end{align*}
\]
For examples

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

\[\lambda x. e \downarrow \lambda x. e \quad e_1 e_2 \downarrow v\]