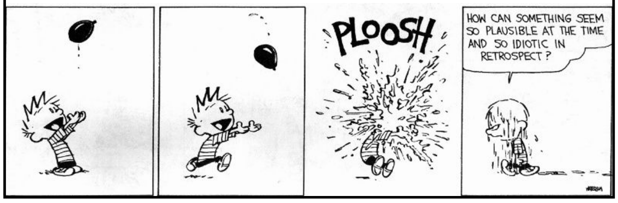
Monomorphic Type Systems

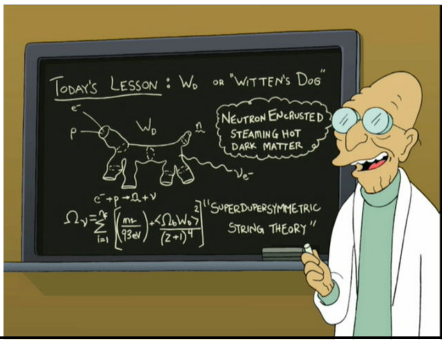


Type Soundness for F_1

- Theorem: If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$
  - Also called, subject reduction theorem, type preservation theorem
- This is one of the most important sorts of theorems in PL
- Whenever you make up a new safe language you are expected to prove this
  - Examples: Vault, TAL, CCured, …

How Can We Prove It?

- Theorem: If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$



Proof Approaches To Type Safety

- Theorem: If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$
- Try to prove by induction on e
  - Won't work because $[v_2/x]e'_1$ in the evaluation of $e_1\ e_2$
  - Same problem with induction on $\cdot \vdash e : \tau$
- Try to prove by induction on τ
  - Won't work because e_1 has a "bigger" type than $e_1\ e_2$
- Try to prove by induction on $e \Downarrow v$
  - To address the issue of $[v_2/x]e'_1$
  - This is it!

Type Soundness Proof

- Consider the case

$$\mathcal{E} :: \frac{e_1 \Downarrow \lambda x : \tau_2 . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

and by inversion on the derivation of $e_1\ e_2 : \tau$

$$\mathcal{D} :: \frac{\cdot \vdash e_1 : \tau_2 \longrightarrow \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\ e_2 : \tau}$$

- From IH on $e_1 \Downarrow$ … we have $\cdot, x : \tau_2 \vdash e_1' : \tau$
- From IH on $e_2 \Downarrow$ … we have $\cdot \vdash v_2 : \tau_2$
- Need to infer that $\cdot \vdash [v_2/x]e_1' : \tau$ and use the IH
  - We need a substitution lemma (by induction on e_1')

Significance of Type Soundness

- The theorem says that the result of an evaluation has the same type as the initial expression
- The theorem does not say that
  - The evaluation *never gets stuck* (e.g., trying to apply a non-function, to add non-integers, etc.), nor that
  - The evaluation *terminates*
- Even though both of the above facts are true of F_1
- We need a small-step semantics to prove that the execution never gets stuck
- I Assert: the execution always terminates in F_1
  - When does the base lambda calculus ever not terminate?

Small-Step Contextual Semantics for F_1

- We define redexes

  r ::= $n_1 + n_2$ | if b then e_1 else e_2 | $(\lambda x{:}\tau.e_1)\ v_2$

- and contexts

  H ::= $H_1 + e_2$ | $n_1 + H_2$ | if H then e_1 else e_2 | $H_1\ e_2$ | $(\lambda x{:}\tau.\ e_1)\ H_2$

- and local reduction rules

$n_1 + n_2$	$\to n_1$ plus n_2
if true then e_1 else e_2	$\to e_1$
if false then e_1 else e_2	$\to e_2$
$(\lambda x{:}\tau.\ e_1)\ v_2$	$\to [v_2/x]e_1$

- and one global reduction rule

  $H[r] \to H[e]$ iff $r \to e$

#7

Decomposition Lemmas for F_1

1. If $\cdot \vdash e : \tau$ and e is not a (final) value then there exist (unique) H and r such that e = H[r]
   - any well typed expression can be decomposed
   - any well-typed non-value can make progress
2. Furthermore, there exists τ' such that $\cdot \vdash r : \tau'$
   - the redex is closed and well typed
3. Furthermore, there exists e' such that $r \to e'$ and $\cdot \vdash e' : \tau'$
   - local reduction is type preserving
4. Furthermore, for any e', $\cdot \vdash e' : \tau'$ implies $\cdot \vdash H[e'] : \tau$
   - the expression preserves its type if we replace the redex with an expression of same type

#8

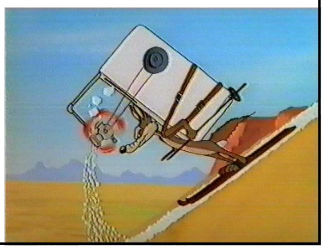
Type Safety of F_1

- Type preservation theorem
  - If $\cdot \vdash e : \tau$ and $e \to e'$ then $\cdot \vdash e' : \tau$
  - Follows from the decomposition lemma
- Progress theorem
  - If $\cdot \vdash e : \tau$ and e is not a value then there exists e' such that e can make progress: $e \to e'$
- Progress theorem says that execution can make progress *on a well typed expression*
- From type preservation we know the execution of well *typed expressions never gets stuck*
  - This is a (very!) common way to *state and prove type safety* of a language

#9

What's Next?

- We've got the basic simply-typed monomorphic lambda calculus
- Now let's make it more complicated …
- By adding features!



Product Types: Static Semantics

- Extend the syntax with (binary) tuples

  e ::= … | (e_1, e_2) | fst e | snd e

  τ ::= … | $\tau_1 \times \tau_2$

  - This language is sometimes called $F_1^{\times}$
- Same typing judgment $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\ e : \tau_2}$$

#11

Product Types: Dynamic Semantics and Soundness

- New form of values: v ::= … | (v_1, v_2)
- New (big step) evaluation rules:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}$$

$$\frac{e \Downarrow (v_1, v_2)}{\mathtt{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\mathtt{snd}\ e \Downarrow v_2}$$

- New contexts: H ::= … | (H_1, e_2) | (v_1, H_2) | fst H | snd H
- New redexes:

  fst $(v_1, v_2) \to v_1$

  snd $(v_1, v_2) \to v_2$

- Type soundness holds just as before

#12

General PL Feature Plan

- The general plan for language feature design
- You invent a new feature (tuples)
- You add it to the lambda calculus
- You invent typing rules and opsem rules
- You extend the basic proof of type safety
- You declare moral victory, and milling throngs of cheering admirers wait to carry you on their shoulders to be knighted by the Queen, etc.

#13

Records

- Records are like tuples with labels (w00t!)
- New form of expressions
  e ::= … | $\{L_1 = e_1, \ldots, L_n = e_n\}$ | e.L
- New form of values
  v ::= $\{L_1 = v_1, \ldots, L_n = v_n\}$
- New form of types
  τ ::= … | $\{L_1 : \tau_1, \ldots, L_n : \tau_n\}$
- … follows the model of $F_1^{\times}$
  - typing rules
  - derivation rules
  - type soundness

#14

Sum Types

- We need disjoint union types of the form:
  - either an int or a float
  - either 0 or a pointer
  - either a (binary tree node with two children) or a (leaf)
- New expressions and types
  e ::= … | injl e | injr e |
  case e of injl x → e_1 | injr y → e_2
  τ ::= … | $\tau_1 + \tau_2$
  - A value of type $\tau_1 + \tau_2$ is *either* a τ_1 *or* a τ_2
  - Like union in C or Pascal, but safe
    - distinguishing between components is under compiler control
  - case is a binding operator (like "let"): x is bound in e_1 and y is bound in e_2 (like OCaml's "match … with")

#15

Examples with Sum Types

- Consider the type unit with a single element called * or ()
- The type integer option defined as "unit + int"
  - Useful for optional arguments or return values
    - No argument: injl * (OCaml's "None")
    - Argument is 5: injr 5 (OCaml's "Some(5)")
  - To use the argument you must test the kind of argument
  - case arg of injl x ⇒ "no_arg_case" | injr y ⇒ "…y…"
  - injl and injr are tags and case is tag checking
- bool is the union type "unit + unit"
  - true is injl *
  - false is injr *
  - if e then e_1 else e_2 is case e of injl x ⇒ e_1 | injr y ⇒ e_2

#16

Static Semantics of Sum Types

- New typing rules

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{injl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{injr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_l : \tau \quad \Gamma, y : \tau_2 \vdash e_r : \tau}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r : \tau}$$

- Types are *not unique* anymore
  injl 1 : int + bool
  injl 1 : int + (int → int)
  - this complicates type checking, but it is still doable

#17

Dynamic Semantics of Sum Types

- New values v ::= … | injl v | injr v
- New evaluation rules

$$\frac{e \Downarrow v}{\texttt{injl}\ e \Downarrow \texttt{injl}\ v} \qquad \frac{e \Downarrow v}{\texttt{injr}\ e \Downarrow \texttt{injr}\ v}$$

$$\frac{e \Downarrow \texttt{injl}\ v \quad [v/x]e_l \Downarrow v'}{\texttt{case}\ e\ \texttt{of injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r \Downarrow v'}$$

$$\frac{e \Downarrow \texttt{injr}\ v \quad [v/y]e_r \Downarrow v'}{\texttt{case}\ e\ \texttt{of injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r \Downarrow v'}$$

#18

Type Soundness for F_1^+

- Type soundness *still holds*
- No way to use a $\tau_1 + \tau_2$ inappropriately
- The key is that the only way to use a $\tau_1 + \tau_2$ is with case, which ensures that you are not using a τ_1 as a τ_2
- In C or Pascal checking the tag is the responsibility of the programmer!
  - Unsafe (yes, even Pascal!)

Types for Imperative Features

- So far: types for pure functional languages
- Now: types for imperative features
- Such types are used to characterize non-local effects
  - assignments
  - exceptions
  - typestate
- Contextual semantics is useful here
  - Just when you thought it was safe to forget it …

Reference Types

- Such types are used for mutable memory cells
- Syntax (as in ML)

  $e ::= \ldots \mid \text{ref } e : \tau \mid e_1 := e_2 \mid ! \, e$

  $\tau ::= \ldots \mid \tau \text{ ref}$

  - ref e - evaluates e, allocates a new memory cell, stores the value of e in it and returns the address of the memory cell
    - like malloc + initialization in C, or new in C++ and Java
  - $e_1 := e_2$, evaluates e_1 to a memory cell and updates its value with the value of e_2
  - ! e - evaluates e to a memory cell and returns its contents

Global Effects, Reference Cells

- A reference cell can escape the static scope where it was created

  $(\lambda f{:}\text{int} \to \text{int ref}.\ !(f\ 5)) \quad (\lambda x{:}\text{int}.\ \text{ref } x : \text{int})$
- The value stored in a reference cell *must be visible from the entire program*
- The "result" of an expression must now include the changes to the heap that it makes (cf. IMP's opsem)
- To model reference cells we must extend the evaluation model

Modeling References

- A heap is a mapping from addresses to values

  $h ::= \cdot \mid h, a \leftarrow v : \tau$
  - $a \in$ Addresses
  - We tag the heap cells with their types
  - Types are useful only for static semantics. They are not needed for the evaluation $\Rightarrow$ are not a part of the implementation
- We call a program an expression with a heap

  p ::= heap h in e
  - The initial program is "heap · in e"
  - Heap addresses act as bound variables in the expression
  - This is a trick that allows easy *reuse of properties of local variables for heap addresses*
    - e.g., we can rename the address and its occurrences at will

Static Semantics of References

- Typing rules for expressions:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{ref}\ e : \tau) : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

- and for programs

$$\frac{\Gamma \vdash v_i : \tau_i\ (i = 1 \,..\, n) \quad \Gamma \vdash e : \tau}{\vdash \texttt{heap}\ h\ \texttt{in}\ e : \tau}$$

where $\Gamma = a_1 : \tau_1\ \texttt{ref}, \ldots, a_n : \tau_n\ \texttt{ref}$

and $h = a_1 \leftarrow v_1 : \tau_1, \ldots, a_n \leftarrow v_n : \tau_n$

Contextual Semantics for References

- Addresses are values: v ::= ... | a
- New contexts: H ::= ref H | H_1 := e_2 | a_1 := H_2 | ! H
- No new *local* reduction rules
- But some new *global* reduction rules
  - heap h in H[ref v : τ] → heap h, a ← v : τ in H[a]
    - where a is fresh (this models allocation - the heap is extended)
  - heap h in H[! a] → heap h in H[v]
    - where a ← v : τ ∈ h (heap lookup - can we get stuck?)
  - heap h in H[a := v] → heap h[a ← v] in H[*]
    - where h[a ← v] means a heap like h except that the part "a ← v_1 : τ" in h is replaced by "a ← v : τ" (memory update)
- Global rules are used to propagate the effects of a write to the entire program (eval order matters!)

#25

Example with References

- Consider these (the redex is underlined)
  - heap · in (λf:int → int ref. !(f 5)) (λx:int. ref x : int)
  - heap · in !((λx:int. ref x : int) 5)
  - heap · in !(ref 5 : int)
  - heap a = 5 : int in !a
  - heap a = 5 : int in 5
- The resulting program has a useless memory cell
- An equivalent result would be

  heap · in 5
- This is a simple way to model garbage collection

#26

Exceptions

- A mechanism that allows non-local control flow
  - Useful for implementing the propagation of errors to caller
- Exceptions ensure* that errors are not ignored
  - Compare with the manual error handling in C
- Languages with exceptions:
  - C++, ML, Modula-3, Java, C#, ...
- We assume that there is a special type exn of exceptions
  - exn could be int to model error codes
  - In Java or C++, exn is a special object type

* *Supposedly.*

#27

Modeling Exceptions

- Syntax

  e ::= ... | raise e | try e_1 handle x ⇒ e_2

  τ ::= ... | exn
- We ignore here how exception values are created
  - In examples we will use integers as exception values
- The handler binds x in e_2 to the actual exception value
- The "raise" expression never returns to the immediately enclosing context
  - 1 + raise 2 is well-typed
  - if (raise 2) then 1 else 2 is also well-typed
  - (raise 2) 5 is also well-typed
  - *What should be the type of raise?*

#28

Example with Exceptions

- A (strange) factorial function

```
let f = λx:int.λres:int. if x = 0 then
                                raise res
                            else
                                f (x - 1) (res * x)
in  try f 5 1 handle x ⇒ x
```

- The function returns in one step from the recursion
- The top-level handler catches the exception and turns it into a regular result

#29

Typing Exceptions

- New typing rules

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{handle}\ x \Longrightarrow e_2 : \tau}$$

- A raise expression has an *arbitrary type*
  - This is a clear sign that the expression does not return to its evaluation context
- The type of the body of try and of the handler must match
  - Just like for conditionals

#30

Dynamics of Exceptions

- The result of evaluation can be an uncaught exception
  - Evaluation answers: a ::= v | uncaught v
  - “uncaught v” has an *arbitrary type*
- Raising an exception has global effects
- It is convenient to use contextual semantics
  - Exceptions propagate through some contexts but not through others
  - We distinguish the handling contexts that intercept exceptions

Contexts for Exceptions

- Contexts
  - H :: = • | H e | v H | raise H | try H handle x ⇒ e
- Propagating contexts
  - Contexts that propagate exceptions to their own enclosing contexts
  - P ::= • | P e | v P | raise P
- Decomposition theorem
  - If e is not a value and e is well-typed then it can be decomposed in exactly one of the following ways:
    - H[(λx:τ. e) v] (normal lambda calculus)
    - H[try v handle x ⇒ e] (handle it or not)
    - H[try P[raise v] handle x ⇒ e] (propagate!)
    - P[raise v] (uncaught exception)

Contextual Semantics for Exceptions

- Small-step reduction rules

H[(λx:τ. e) v]	→ H[[v/x] e]
H[try v handle x ⇒ e]	→ H[v]
H[try P[raise v] handle x ⇒ e]	→ H[[v/x] e]
P[raise v]	→ uncaught v

- The handler is ignored if the body of try completes normally
- A raised exception propagates (in one step) to the closest enclosing handler or to the top of the program

Exceptions. Comments.

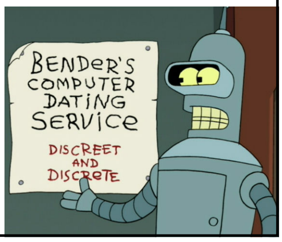
- The addition of exceptions preserves type soundness
- Exceptions are like *non-local goto*
- However, they cannot be used to implement recursion
  - Thus we still cannot write non-terminating programs
- There are a number of ways to implement exceptions (e.g., “zero-cost” exceptions)

Continuations

- Some languages have a mechanism for taking a snapshot of the execution and storing it for later use
  - Later the execution can be reinstated from the snapshot
  - Useful for implementing threads, for example
  - Examples: Scheme, LISP, ML, C (yes, really!)
- Consider the expression: $e_1 + e_2$ in a context C
  - How to express a snapshot of the execution right after evaluating e_1 but before evaluating e_2 and the rest of C ?
  - Idea: as a context $C_1 = C [\bullet + e_2]$
    - Alternatively, as $\lambda x_1 .\ C [x_1 + e_2]$
  - When we finish evaluating e_1 to v_1, we fill the context and continue with $C[v_1 + e_2]$
  - But the C_1 continuation is still available and we can continue several times, with different replacements for e_1

Continuation Uses in “Real Life”

- You’re walking and come to a fork in the road
- You save a continuation “right” for going right
- But you go left (with the “right” continuation in hand)
- You encounter Bender. Bender coerces you into joining his computer dating service.
- You save a continuation “bad-date” for going on the date.
- You decide to invoke the “right” continuation
- So, you go right (no evil date obligation, but with the “bad-date” continuation in hand)
- A train hits you!
- On your last breath, you invoke the “bad-date” continuation



Continuations

- Syntax:
  - e ::= callc k in e | throw e_1 e_2
  - τ ::= … | τ cont
- τ cont - the type of a continuation that expects a τ
- callcc k in e - sets k to the current context of the execution and then evaluates expression e
  - when e terminates, the whole callcc terminates
  - e can invoke the saved continuation (many times even)
  - When e invokes k it is as if "callcc k in e" returns
  - k is bound in e
- throw e_1 e_2 - evaluates e_1 to a continuation, e_2 to a value and invokes the continuation with the value of e_2 (just wait, we'll explain it!)

#37

Example with Continuations

- Example: another strange factorial

```
callcc k in
  let f = λx:int.λres:int. if x = 0 then throw k res
                                    else f (x - 1) (x * res)
  in f 5 1
```

- First we save the current context
  - This is the top-level context
  - A throw to k of value v means "pretend the whole callcc evaluates to v"
- This simulates exceptions
- Continuations are *strictly more powerful* that exceptions
  - The destination is not tied to the call stack

#38

Static Semantics of Continuations

$$\frac{\Gamma, k : \tau\ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{callcc}\ k\ \texttt{in}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{cont} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{throw}\ e_1\ e_2 : \tau'}$$

- Note that the result of callcc is of type τ
  "callcc k in e" returns in two possible situations
  1. e *throws* to k a value of type τ, or
  2. e *terminates normally* with a value of type τ
- Note that throw has any type τ'
  - Since it *never returns* to its enclosing context

#39

Dynamic Semantics of Continuations

- Use contextual semantics (wow, again!)
  - Contexts are now manipulated directly
  - Contexts are values of type τ cont
- Contexts
  - H ::= • | H e | v H | throw H_1 e_2 | throw v_1 H_2
- Evaluation rules
  - H[(λx.e) v] → H[[v/x] e]
  - H[callcc k in e] → H[[H/k] e]
  - H[throw H_1 v_2] → $H_1[v_2]$
- callcc duplicates the current continuation
- Note that throw abandons its own context

#40

Implementing Coroutines with Continuations

- Example:

```
let client = λk. let res = callcc k' in throw k k' in
                 print (fst res);
                 client (snd res)
```

  - "client k" will invoke "k" to get an integer and a continuation for obtaining more integers

```
let getnext =
   λL.λk. if L = nil then raise 0
          else getnext (cdr L) (callcc k' in throw k  (car L, k'))
```

  - "getnext L k" will send to "k" the first element of L along with a continuation that can be used to get more elements of L

```
getnext [0;1;2;3;4;5] (callcc k in client k)
```

#41

Continuation Comments

- In our semantics the continuation saves the entire context: program counter, local variables, call stack, and the heap!
- In actual implementations the *heap is not saved!*
- Saving the stack is done with various tricks, but it is expensive in general.
- Few languages implement continuations
  - Because their presence complicates the whole compiler considerably
  - Except if you use a continuation-passing-style of compilation (more on this next)

#42

Continuation Passing Style

- A style of compilation where evaluation of a function *never returns directly*: instead the function is *given a continuation to invoke with its result*.
- Instead of
  f(int a) { return h(g(e)); }
- we write
  f(int a, cont k) { g(e, λr. h(r, k)) }
- Advantages:
  - interesting compilation scheme (supports callcc easily)
  - no need for a stack, can have multiple return addresses (e.g., for an error case)
  - fast and safe (non-preemptive) multithreading

#43

Continuation Passing Style

- Let $e ::= x \mid n \mid e_1 + e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \lambda x.e \mid e_1\ e_2$
- Define cps(e, k) as the code that computes e in CPS and *passes the result to continuation* k

$$cps(x, k) = k\ x$$
$$cps(n, k) = k\ n$$
$$cps(e_1 + e_2, k) = cps(e_1, \lambda n_1.cps(e_2,\lambda n_2.k\ (n_1 + n_2)))$$
$$cps(\lambda x.e, k) = k\ (\lambda x \lambda k'.\ cps(e,k'))$$
$$cps(e_1\ e_2, k) = cps(e_1, \lambda f_1.cps(e_2,\lambda v_2.\ f_1\ v_2\ k))$$

- Example: cps (h(g(5)), k) = g(5, λx.h x k)
  - Notice the order of evaluation being explicit

#44

Homework

- Read Wright and Felleisen article
  - … that you didn't read on Tuesday.
- Soon: Class Survey #2
- Soon: Bonus Lecture #2
- Work on your projects!
  - Status Update Due: Thursday Mar 23

#45