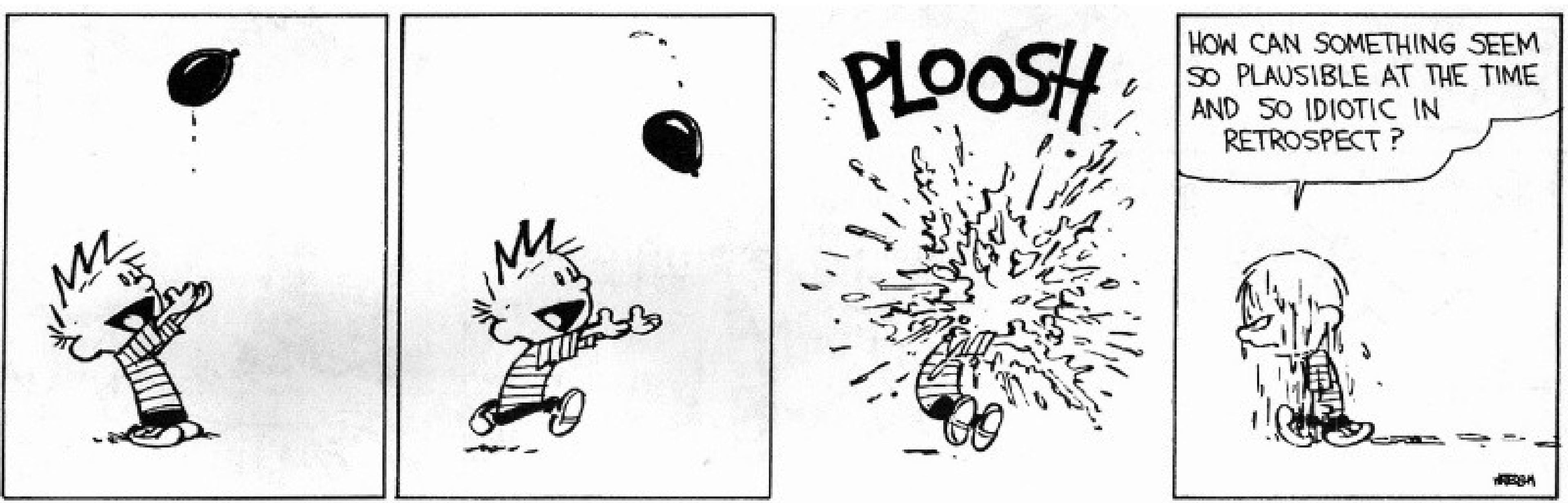
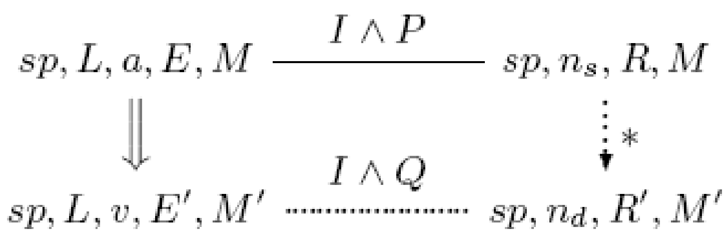
Monomorphic Type Systems



The Reading

- Explain the Xavier Leroy article to me …

The correctness of the translation follows from a simulation argument between the executions of the Cminor source and the RTL translation, proved by induction on the Cminor evaluation derivation. In the case of expressions, the simulation property is summarized by the following diagram:



On the choice of semantics We used big-step semantics for the source language, "mixed-step" semantics for the intermediate languages, and small-step semantics for the target language. A consequence of this choice is that our semantic preservation theorems hold only for terminating source programs: they all have premises of the form "if the source program evaluates to result r", which do not hold for non-terminating programs. This is unfortunate for

- How did he do register allocation?

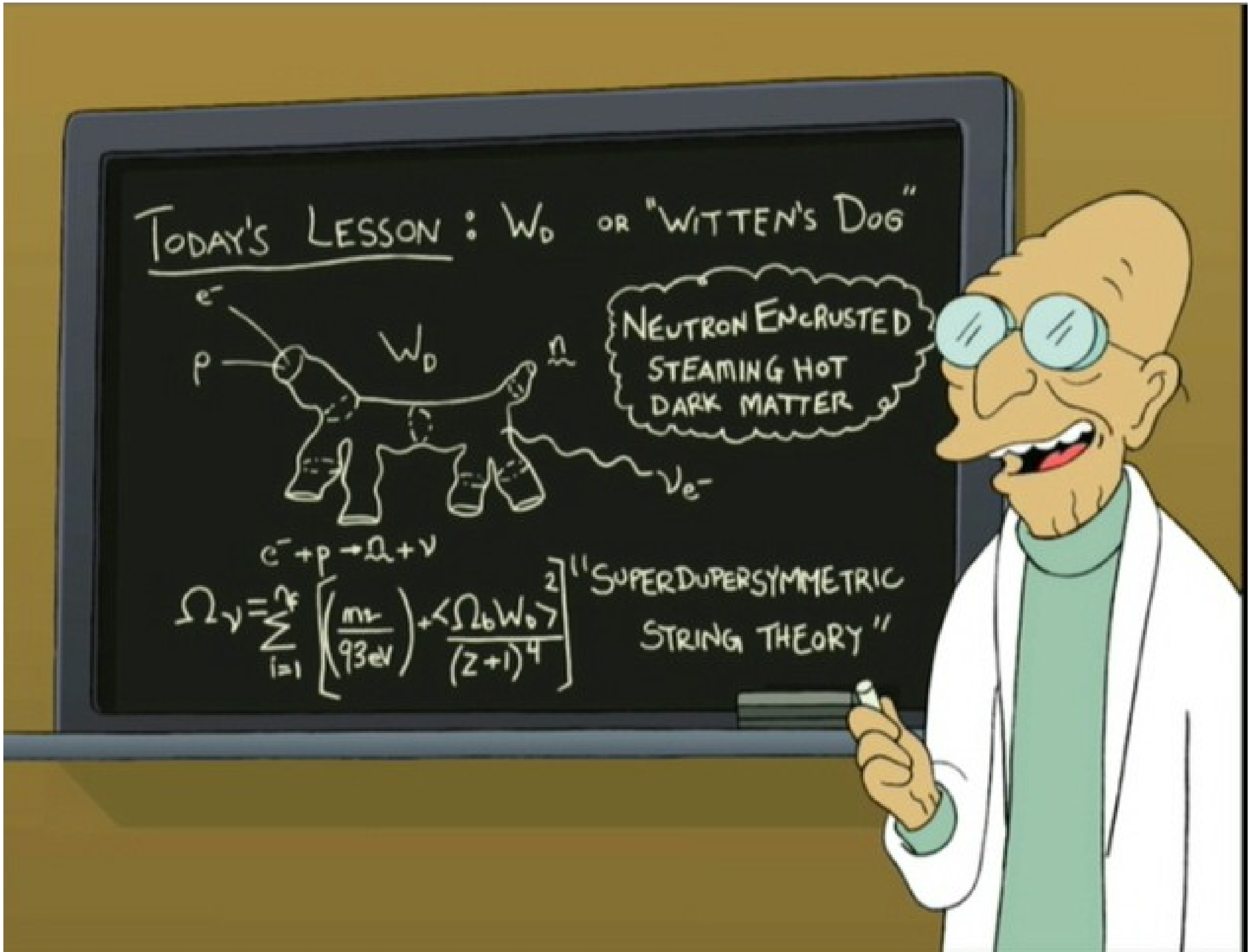
Type Soundness for F_1

What does this mean?

- 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 Might 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 $e_1\ e_2$
- ???

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 *function application* 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 <u>does not</u> 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
- What formal system of semantics do we use to reason about programs that might not terminate?

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

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

- and contexts

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

- and local reduction rules

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

- and one global reduction rule

  $$H[r] \rightarrow H[e] \quad \text{iff } r \rightarrow e$$

Decomposition Lemmas for F_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
- Furthermore, there exists τ' such that $\cdot \vdash r : \tau'$
  - the redex is closed and well typed
- Furthermore, there exists e' such that $r \to e'$ and $\cdot \vdash e' : \tau'$
  - local reduction is type preserving
- 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

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

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

$$e \quad ::= \ldots \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e$$

$$\tau \quad ::= \ldots \mid \tau_1 \times \tau_2$$

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

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

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

Dynamic Semantics and Soundness

- New form of values: $v ::= \ldots \mid (v_1, v_2)$
- New (big step) evaluation rules:

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

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

- New contexts: $H ::= \ldots \mid (H_1, e_2) \mid (v_1, H_2) \mid \text{fst } H \mid \text{snd } H$
- New redexes:

$$\text{fst } (v_1, v_2) \to v_1$$

$$\text{snd } (v_1, v_2) \to v_2$$

- Type soundness holds just as before

Q: General (454 / 842)

- In traditional logic this is an inference in which one proposition (the conclusion) necessarily follows from two others (the premises). An overused example is: *"All men are mortal. Socrates is a man. Therefore, Socrates is a mortal."*

Q: General (473 / 842)

- Which of the following chemical processes or reactions would be the most difficult to conduct in a high school chemistry lab?
  - Hall-Heroult (Aluminum Extraction) Process
  - Making Nitrocellulose (Guncotton)
  - Making Slime (disodium tetraborate)
  - Thermite Reaction (which reaches 5000(F))

Q: Games (534 / 842)

- Each face of this 1974 six-sided plastic puzzle is subdivided into nine smaller faces, each of which can be one of six colors.

Computer Science

- This American Turing award winner is known for foundational work in data structures and algorithms. Examples include off-line least common ancestors, strongly connected components, the Fibonacci heap, the splay tree, and the disjoint-set data structure. This prolific author has over 228 refereed journal and book chapters.

Q: Games (547 / 842)

- This viscoelastic silicone plastic "clay" came out of efforts to find a rubber substitute in World War II. It is now sold in plastic eggs as a toy for children. It bounces and can absorb the ink from newsprint. It was also used by the crew of Apollo 8 to secure tools in zero gravity.

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.

Records

- Records are like tuples with labels (w00t!)
- New form of expressions

$$e ::= \ldots \mid \{L_1 = e_1, \ldots, L_n = e_n\} \mid e.L$$

- New form of values

$$v ::= \{L_1 = v_1, \ldots, L_n = v_n\}$$

- New form of types

$$\tau ::= \ldots \mid \{L_1 : \tau_1, \ldots, L_n : \tau_n\}$$

- ... follows the model of $F_1^{\times}$
  - typing rules
  - derivation rules
  - type soundness

On the board!

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 $\rightarrow$ e_1 | injr y $\rightarrow$ e_2

  $\tau ::= \ldots \mid \tau_1 + \tau_2$

  - 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")

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

Static Semantics of Sum Types

- New typing rules

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

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

- Types are *not unique* anymore

  injl 1 : int + bool

  injl 1 : int + (int $\rightarrow$ int)

  – this complicates type checking, but it is still doable

Dynamic Semantics of Sum Types

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

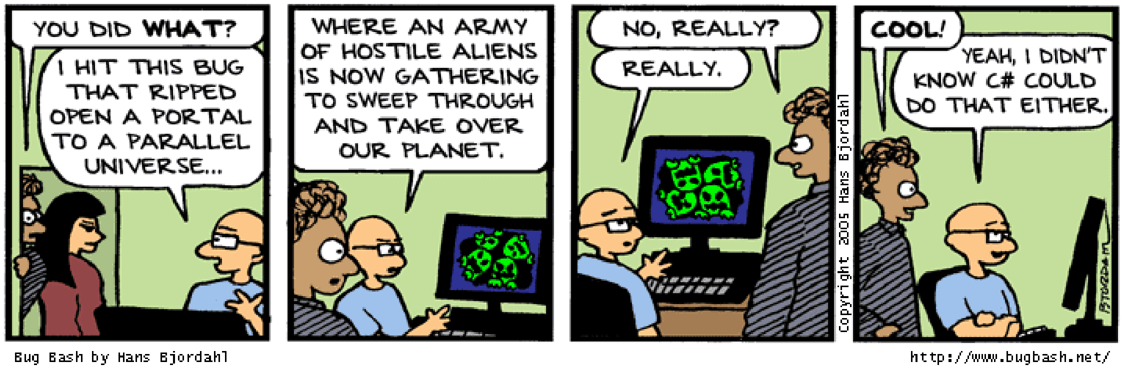
$$\frac{e \Downarrow v}{\texttt{injl}\ e \Downarrow \texttt{injl}\ v} \qquad \frac{e \Downarrow v}{\texttt{injr}\ e \Downarrow \texttt{injr}\ v}$$

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

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

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



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)

Why do I need : τ ?

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

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

  – ref e : τ - 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

  $(\lambda f{:}int \rightarrow int\ ref.\ !(f\ 5))\quad(\lambda x{:}int.\ ref\ x : int)$

- 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 (Addresses $\neq \mathbb{Z}$?)
  - 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 $\cdot$ 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:

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

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

- and for programs

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

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: $v ::= \ldots \mid a$
- New contexts: $H ::= \text{ref } H \mid H_1 := e_2 \mid a_1 := H_2 \mid \, ! \, H$
- 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!)

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

Homework

- HW 5 due Today or Thursday
- Project Status Update Due Thursday