

**CMSC 631 – Program Analysis and Understanding
Fall 2006**

Type Systems

The Need for a Type System

- Recall for homework 2 that not all programs accepted by the grammar of the language are sensible:
 - case 0 of nil => c1 or x,y => c2
 - The case statement expects a list, but we have given it a numeral. There is no rule to evaluate such a program, so we're "stuck."
- It would be great to rule out such non-sensical programs in advance, so we can a program can never reach a "stuck state."

CMSC 631

2

What is a Type System?

- A *type system* is some mechanism for distinguishing good programs from bad
 - Good programs = well typed
 - Bad programs = ill typed or not typable
- Examples:
 - `0 + 1` // well typed
 - `false 0` // ill-typed: can't apply a boolean
 - `1 + (if true then 0 else false)` // ill-typed: can't add boolean to integer

CMSC 631

3

A Definition of Type Systems

- "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."
 - — Benjamin Pierce, *Types and Programming Languages*

CMSC 631

4

Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x.t.e \mid e e$
 - Functions include the type t of their argument
 - We don't really need this, but it will come in handy
- $t ::= \text{int} \mid t \rightarrow t$
 - $t1 \rightarrow t2$ is the type of a function that, given an argument of type $t1$, returns a result of type $t2$
 - $t1$ is the *domain*, and $t2$ is the *range*

CMSC 631

5

Type Judgments

- Our type system will prove *judgments* of the form
 - $A \vdash e : t$
 - "In type environment A , expression e has type t "

CMSC 631

6

Type Environments

- A *type environment* (a.k.a. *context*) is a map from variables to types (a kind of symbol table)
 - \emptyset is the empty type environment
 - A closed term e is *well-typed* if $\emptyset \vdash e : t$ for some t
 - We'll abbreviate this as $\vdash e : t$
 - $A, x:t$ is just like A , except x now has type t
 - The type of x in $A, x:t$ is t
 - The type of $z \neq x$ in $A, x:t$ is the type of z in A
- When we see a variable in a program, we look in the type environment to find its type

CMSC 631

7

Type Rules

$$\frac{}{A \vdash n : \text{int}} \qquad \frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A, x:t \vdash e : t'}{A \vdash \lambda x:t. e : t \rightarrow t'} \qquad \frac{A \vdash e_1 : t \rightarrow t' \quad A \vdash e_2 : t}{A \vdash e_1 e_2 : t'}$$

CMSC 631

8

Example

$$\frac{\frac{- \in \text{dom}(A)}{A \vdash - : \text{int} \rightarrow \text{int}} \quad A \vdash 3 : \text{int}}{A \vdash - 3 : \text{int}}$$

CMSC 631

9

Another Example

$$\frac{\frac{\frac{+ \in \text{dom}(B) \quad x \in \text{dom}(B)}{B \vdash + : i \rightarrow i \rightarrow i \quad B \vdash x : i}}{B \vdash + x : \text{int} \rightarrow \text{int}} \quad B \vdash 3 : \text{int}}{B \vdash + x 3 : \text{int}}}{A \vdash (\lambda x:\text{int}. + x 3) : \text{int} \rightarrow \text{int}} \quad A \vdash 4 : \text{int}}{A \vdash (\lambda x:\text{int}. + x 3) 4 : \text{int}}$$

We'd usually use infix $x + 3$

CMSC 631

10

An Algorithm for Type Checking

- Our type rules are deterministic
 - For each syntactic form, only one possible rule
- They define a natural type checking algorithm
 - `TypeCheck`: $\text{type env} \times \text{expression} \rightarrow \text{type}$
 - `TypeCheck(A, n) = int`
 - `TypeCheck(A, x) = if x in dom(A) then A(x) else fail`
 - `TypeCheck(A, $\lambda x:t. e$) = TypeCheck((A, x:t), e)`
 - `TypeCheck(A, $e_1 e_2$) =`
 - `let t1 \rightarrow t2 = TypeCheck(A, e1) in`
 - `let t1' = TypeCheck(A, e2) in`
 - `if t1 = t1' then t2 else fail`

CMSC 631

11

Semantics

- Here is a small-step, call-by-value semantics
 - If an expression can't be evaluated any more and is not a value, then it is *stuck*

$$\frac{}{(\lambda x.e_1) v_2 \rightarrow e_1[v_2/x]} \qquad \frac{e_1 \rightarrow e_1' \quad e_2 \rightarrow e_2'}{e_1 e_2 \rightarrow e_1' e_2'}$$

$$\frac{e_2 \rightarrow e_2'}{v_1 e_2 \rightarrow v_1 e_2'}$$

$e ::= v \mid x \mid e e$

$v ::= n \mid \lambda x:t. e$ *values – not evaluated*

CMSC 631

12

Progress

- Suppose $\vdash e : t$. Then either e is a value, or there exists e' such that $e \rightarrow e'$
- Proof by induction on e
 - Base cases $n, \lambda x.e$ – these are values, so we're done
 - Base case x – can't happen (empty type environment)
 - Inductive case $e_1 e_2$ – If e_1 is not a value, then by induction we can evaluate it, so we're done, and similarly for e_2 . Otherwise both e_1 and e_2 are values. Inspection of the type rules shows that e_1 must have a function type, and therefore must be a lambda since it's a value. Therefore we can make progress.

CMSC 631

13

Preservation

- If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$
- Proof by induction on e
 - Base cases $n, x, \lambda x.e$ – Impossible, since these terms don't reduce
 - Induction. Assume $\vdash e_1 e_2 : t$ and $e_1 e_2 \rightarrow e'$. Then we have $\vdash e_1 : t' \rightarrow t$ and $\vdash e_2 : t'$. (Why?)
 - Then there are three cases.
 - If $e_1 \rightarrow e_1'$, then by induction $\vdash e_1' : t' \rightarrow t$, so $e_1' e_2$ has type t
 - If reduction inside e_2 , similar

CMSC 631

14

Preservation, cont'd

- Otherwise $(\lambda x.e) v \rightarrow e[v/x]$. Then we have

$$\frac{x : t' \vdash e : t}{\vdash \lambda x.e : t' \rightarrow t}$$

- Thus we have
 - $x : t' \vdash e : t$
 - $\vdash v : t'$
- Then by the substitution lemma (not shown) we have
 - $\vdash e[v/x] : t$
- And so we have preservation

CMSC 631

15

Substitution Lemma

- If $A \vdash v : t$ and $A, x:t \vdash e : t'$, then $A \vdash e[v/x] : t'$
- Proof: Induction on the structure of e

CMSC 631

16

Soundness

- So we have
 - Progress: Suppose $\vdash e : t$. Then either e is a value, or there exists e' such that $e \rightarrow e'$
 - Preservation: If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$
- Putting these together, we get soundness
 - If $\vdash e : t$ then either there exists a value v such that $e \rightarrow^* v$, or e diverges (doesn't terminate).
- What does this mean?
 - Semantics define bad things (evaluation getting stuck)
 - "Well-typed programs don't go wrong"

CMSC 631

17

Product Types (Tuples)

- $e ::= \dots \mid (e, e) \mid \text{fst } e \mid \text{snd } e$

$$\frac{\frac{A \vdash e_1 : t \quad A \vdash e_2 : t'}{A \vdash (e_1, e_2) : t \times t'}}{\frac{A \vdash e : t \times t'}{A \vdash \text{fst } e : t} \quad \frac{A \vdash e : t \times t'}{A \vdash \text{snd } e : t'}}$$

- Or, maybe, just add functions

- $\text{pair} : t \rightarrow t' \rightarrow t \times t'$
- $\text{fst} : t \times t' \rightarrow t$
- $\text{snd} : t \times t' \rightarrow t'$

CMSC 631

18

Sum Types (Tagged Unions)

- $e ::= \dots \mid \text{inL}_{t_1} e \mid \text{inR}_{t_2} e$
- $\mid (\text{case } e \text{ of } x_1:t_1 \rightarrow e_1 \mid x_2:t_2 \rightarrow e_2)$

$$\frac{A \vdash e : t_1}{A \vdash \text{inL}_{t_2} e : t_1 + t_2} \quad \frac{A \vdash e : t_2}{A \vdash \text{inR}_{t_1} e : t_1 + t_2}$$

$$\frac{A \vdash e : t_1 + t_2 \quad A, x_1:t_1 \vdash e_1 : t \quad A, x_2:t_2 \vdash e_2 : t}{A \vdash (\text{case } e \text{ of } x_1:t_1 \rightarrow e_1 \mid x_2:t_2 \rightarrow e_2) : t}$$

CMSC 631

19

Self Application and Types

- Self application is not checkable in our system

$$\frac{\frac{A, x:? \vdash x : t \rightarrow t' \quad A, x:? \vdash x : t}{A, x:? \vdash x x : \dots}}{A \vdash \lambda x:? . x x : \dots}$$

- It would require a type t such that $t = t \rightarrow t'$
– (We'll see this next, but so far...)

- The simply-typed lambda calculus is *strongly normalizing*

- Every program has a normal form
- I.e., every program halts!

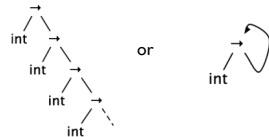
CMSC 631

20

Recursive Types

- We can type self application if we have a type to represent the solution to equations like $t = t \rightarrow t'$

- We define the type $\mu\alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
- Example: $\mu\alpha.\text{int} \rightarrow \alpha$



CMSC 631

21

Folding and Unfolding

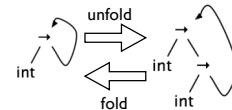
- We can check type equivalence with the previous definition

- Standard unification, omit occurs checks

- Alternative solution:

- The programmer puts in explicit *fold* and *unfold* operations to expand/contract one “level” of the type trees

- $\text{unfold } \mu\alpha.t = t[\mu\alpha.t/\alpha]$
- $\text{fold } t[\mu\alpha.t/\alpha] = \mu\alpha.t$



CMSC 631

22

Discussion

- In the pure lambda calculus, every term is typable with recursive types
 - (Pure = variables, functions, applications only)
- Most languages have some kind of “recursive” type
 - E.g., for data structures like lists, tree, etc.
- However, usually two recursive types that define the same structure but use a different name are considered different
 - E.g., `struct foo { int x; struct foo *next; }` is different from `struct bar { int x; struct bar *next; }`

CMSC 631

23

Recap

- We've discussed simple types so far
 - Integers, functions, pairs, unions
 - Extensions for recursive types and updatable refs
- Type systems have nice properties
 - Type checking is straightforward (needs annotations)
 - Well typed programs don't go “wrong”
 - They don't get stuck in the operational semantics
- But...We can't type check all good programs

CMSC 631

24

Up Next: Improving Types

- How can we build more flexible type systems?
 - More programs type check
 - Type checking is still tractable
- How can reduce the annotation burden?
 - Type inference

CMSC 631

25

Type Inference

- Let's consider the simply typed lambda calculus with integers
 - $e ::= n \mid x \mid \lambda x.e \mid e e$
 - (No parametric polymorphism)
- *Type inference*: Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?
 - Notice that lambda terms above have no type annotation

CMSC 631

26

Type Language

- Problem: Consider the rule for functions

$$\frac{A, x.t \vdash e : t'}{A \vdash \lambda x.t : t \rightarrow t'}$$
- Without type annotations, where do we get t ?
 - We'll use *type variables* for as-yet-unknown types
 - $t ::= \alpha \mid \text{int} \mid t \rightarrow t$
 - We'll generate *equality constraints* $t = t$ among the types and type variables
 - And then we'll solve the constraints to compute a typing

CMSC 631

27

Type Inference Rules

$$\frac{}{A \vdash n : \text{int}} \quad \frac{x \in \text{dom}(A)}{A \vdash x : A(x)}$$

$$\frac{A, x:\alpha \vdash e : t' \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \rightarrow t'} \quad \frac{A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \quad [t_1 = t_2 \rightarrow \beta] \quad \beta \text{ fresh}}{A \vdash e_1 e_2 : \beta}$$

“Generated” constraint

CMSC 631

28

Example

$$\frac{A, x:\alpha \vdash x:\alpha}{A \vdash (\lambda x.x) : \alpha \rightarrow \alpha} \quad A \vdash 3 : \text{int} \quad \alpha \rightarrow \alpha = \text{int} \rightarrow \beta}{A \vdash (\lambda x.x) 3 : \beta}$$

- We collect all constraints appearing in the derivation into some set C to be solved
- Here, C consists of one constraint $\alpha \rightarrow \alpha = \text{int} \rightarrow \beta$
 - Solution: $\alpha = \text{int} = \beta$
- Thus this program is typable, and we can derive a typing by replacing α and β by int in the proof tree

CMSC 631

29

Solving Equality Constraints

- We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set
 - $C \cup \{\text{int} = \text{int}\} \Rightarrow C$
 - $C \cup \{\alpha = t\} \Rightarrow C[t/\alpha]$
 - $C \cup \{t = \alpha\} \Rightarrow C[t/\alpha]$
 - $C \cup \{t_1 \rightarrow t_2 = t_1' \rightarrow t_2'\} \Rightarrow C \cup \{t_1 = t_1'\} \cup \{t_2 = t_2'\}$
 - $C \cup \{\text{int} = t_1 \rightarrow t_2\} \Rightarrow \text{unsatisfiable}$
 - $C \cup \{t_1 \rightarrow t_2 = \text{int}\} \Rightarrow \text{unsatisfiable}$

CMSC 631

30

Termination

- We can prove that the constraint solving algorithm terminates.
- For each rewriting rule, either
 - We reduce the size of the constraint set
 - We reduce the number of “arrow” constructors in the constraint set
- As a result, the constraint always gets “smaller” and eventually becomes empty
 - A similar argument is made for strong normalization in the simply-typed lambda calculus

CMSC 631

31

Occurs Check

- We don’t have recursive types, so we shouldn’t infer them
- In the operation $C[t\alpha]$, require that $\alpha \notin FV(t)$ (where $FV(t)$ is as expected; defined formally later)
 - Called the “occurs check”
- In practice, it may better to allow $\alpha \in FV(t)$ and do the occurs check at the end
 - But that can be awkward to implement

CMSC 631

32

Unifying a Variable and a Type

- Computing $C[t\alpha]$ by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
 - The types are in a union-find forest
 - When a variable and a type are equated, we union them so they have the same ECR
 - Want the ECR to be the concrete type with which variables have been unified, if one exists. Thus we can read off the solution by reading the ECR for each set.

CMSC 631

33

Example



$\alpha = \text{int} \rightarrow \beta$
 $\gamma = \text{int} \rightarrow \text{int}$
 $\alpha = \gamma$

CMSC 631

34

Unification

- The process of finding a solution to a set of equality constraints is called *unification*
 - Original algorithm due to Robinson
 - But his algorithm was inefficient
 - Often written out in different form
 - See Algorithm W
 - Constraints usually solved on-line
 - As type inference rules applied

CMSC 631

35

Discussion

- The algorithm we’ve given finds the *most general type* of a term
 - Any other valid type is “more specific,” e.g.,
 - $\lambda x.x : \text{int} \rightarrow \text{int}$
 - Formally, any other valid type can be gotten from the most general type by applying a substitution to the type variables
- This is still a *monomorphic* type system
 - α stands for “some particular type, but it doesn’t matter exactly which type it is”

CMSC 631

36

Parametric Polymorphism

- Observation: $\lambda x.x$ returns its argument exactly and places no constraints on the type of x
 - The identity function works for any argument type
- We can express this with *universal quantification*:
 - $\lambda x.x : \forall \alpha. \alpha \rightarrow \alpha$
 - For any type α , the identity function has type $\alpha \rightarrow \alpha$
 - This is also known as *parametric polymorphism*

CMSC 631

37

“Manual” use of polymorphism

- Let's extend our system in two simple ways:
 - $t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha. t$
 - $e ::= n \mid x \mid \lambda x.e \mid e e \mid e [t]$
- That is, we add polymorphic types, and we add explicit *type instantiations*
 - Explicitly annotated code locations at which a value of polymorphic type is used
- We also need to know when to introduce polymorphic types
 - Called *generalization*
 - For now, we'll just define suitable conditions, but won't be syntax-driven

CMSC 631

38

Instantiation

- When we use a parametric polymorphic type, we *instantiate* it with a particular type
 - Suppose the programmer specifies this by hand as
 - $(\lambda x.x)[t1] : t1 \rightarrow t1$
 - $(\lambda x.x)[t2] : t2 \rightarrow t2$
- This is where the term *parametric* comes from
 - The type $\forall \alpha. \alpha \rightarrow \alpha$ is a “function” in the domain of types, and it is passed a parameter at instantiation time

CMSC 631

39

Instantiation Rule

$$\frac{A \vdash e : \forall \alpha. t}{A \vdash e[t'] : t[t'\alpha]}$$

- Notice we can substitute in *any* type
 - That's what the forall means!

CMSC 631

40

Free Variables, Again

- We're going to need to perform substitutions on quantified types
 - So just like with lambda calculus, we need to worry about free variables and capture-free substitution
- Define the free variables of a type
 - $FV(\alpha) = \{\alpha\}$
 - $FV(c) = \emptyset$
 - $FV(t \rightarrow t') = FV(t) \cup FV(t')$
 - $FV(\forall \alpha. t) = FV(t) - \{\alpha\}$

– Look familiar?

CMSC 631

41

Substitution, Again

- Define $t[u\alpha]$ as
 - $\alpha[u\alpha] = u$
 - $\beta[u\alpha] = \beta$ where $\beta \neq \alpha$
 - $(t \rightarrow t')[u\alpha] = t[u\alpha] \rightarrow t'[u\alpha]$
 - $(\forall \beta. t)[u\alpha] = \forall \beta. (t[u\alpha])$ where $\beta \neq \alpha$ and $\beta \in FV(u)$
- Look familiar?

CMSC 631

42

Generalization

- Question: When is it safe to generalize (quantify) a type variable α in the type of expression e ?
- Answer: Whenever we can redo the typing proof for e , choosing α to be anything we want, and still have a valid typing proof.

CMSC 631

43

Examples

$$\frac{A, x:\alpha \vdash e : \alpha}{A \vdash \lambda x.x : \alpha \rightarrow \alpha} \quad \begin{array}{l} \xrightarrow{\text{}} \\ \xrightarrow{\text{}} \end{array} \quad \frac{A, x:\text{int} \vdash x : \text{int}}{A \vdash \lambda x.x : \text{int} \rightarrow \text{int}} \quad \frac{A, x:(i \rightarrow i) \vdash x : (i \rightarrow i)}{A \vdash \lambda x.x : (i \rightarrow i) \rightarrow (i \rightarrow i)}$$

- The choice of the type of x is purely local to type checking $\lambda x.x$
 - There is no interaction with the outside environment
 - Thus we can generalize the type of x

CMSC 631

44

Examples (cont'd)

$$\frac{A, x:\text{int} \vdash x : \text{int}}{A \vdash \lambda x.x+3 : \text{int} \rightarrow \text{int}}$$

- The function restricts the type of x , so we cannot introduce a type variable
 - Thus we cannot generalize the type of x
 - We can only generalize when the function doesn't "look at" its parameter

CMSC 631

45

Examples (cont'd)

$$\frac{A, y:\alpha, x:\alpha \vdash \text{if } p \text{ then } x \text{ else } y : \alpha}{A, y:\alpha \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y : \alpha \rightarrow \alpha}$$

$$\frac{A, y:\alpha, x:\text{int} \vdash \text{if } p \text{ then } x \text{ else } y : \text{int}}{A, y:\alpha \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y : \text{int} \rightarrow \text{int}}$$

- The choice of the type of x depends on the type environment (x must be α because y is)
 - In the first derivation, x and y have the same type; if we generalize the type of x , they could have different types
 - Thus we cannot generalize the type of x

CMSC 631

46

Generalization Rule

$$\frac{A \vdash e : t \quad \alpha \notin \text{FV}(A)}{A \vdash e : \forall \alpha.t}$$

- We can generalize any type variable that is unconstrained by the environment
 - Warning: This won't quite work with refs

CMSC 631

47

Another Justification

- Suppose we have
 - $A \vdash e : t$ and $\alpha \notin \text{FV}(A)$
- Then let u be any type. By induction, can show
 - $A[u/\alpha] \vdash e : t[u/\alpha]$
 - But then since $\alpha \notin \text{FV}(A)$, that's equivalent to
 - $A \vdash e : t[u/\alpha]$

CMSC 631

48

Kinds of Polymorphism

- We've just seen parametric polymorphism
 - A more restrictive variant is also called Hindley-Milner style polymorphism
- Another popular flavor is subtype polymorphism
 - As in OO programming
 - These two can be combined (e.g., Java Generics)
- Some languages also have *ad-hoc polymorphism*
 - E.g., + operator that works on ints and floats
 - E.g., overloading in Java

CMSC 631

49

Polymorphic Type Inference

- We'd like to extend our algorithm to polymorphic type inference
 - Perform generalization and instantiation automatically (and deterministically); remove explicit type instantiations
- Major problem: Our system for polymorphism is too expressive
 - In fact, type inference is undecidable when generalization and instantiation can be at arbitrary syntactic positions

CMSC 631

50

Hindley-Milner Polymorphism

- Restrict polymorphism to only the "top level"
 - Introduce polymorphism at `let`
 - Fully instantiate when we use a variable with a polymorphic type
- Here is our new language
 - $e ::= n \mid x \mid \lambda x.e \mid e \ e \mid \text{let } x = e \text{ in } e$
 - $t ::= \alpha \mid \text{int} \mid t \rightarrow t$
 - $s ::= t \mid \forall \alpha.s$
 - These are type schemes.
 - $A ::= \emptyset \mid A, x:s$
 - Notice that, according to the prior instantiation rule, we won't instantiate α with a scheme s , only a type t

CMSC 631

51

Old Type Inference Rules

$$\frac{}{A \vdash n : \text{int}}$$

$$\frac{A, x:\alpha \vdash e : t' \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \rightarrow t'}$$

$$\frac{A \vdash e1 : t1 \quad A \vdash e2 : t2 \quad t1 = t2 \rightarrow \beta \quad \beta \text{ fresh}}{A \vdash e1 \ e2 : \beta}$$

CMSC 631

52

New Type Inference Rules

- At `let`, generalize over all possible variables

$$\frac{A \vdash e1 : t1 \quad A, x:\forall \vec{\alpha}.t1 \vdash e2 : t2 \quad \vec{\alpha} = \text{FV}(t1) - \text{FV}(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

- At variable uses, instantiate to all fresh types

$$\frac{A(x) = \forall \vec{\alpha}.t \quad \vec{\beta} \text{ fresh}}{A \vdash x : t[\vec{\beta}/\vec{\alpha}]}$$

- Here the $\vec{\alpha}$ denotes a list of type variables

CMSC 631

53

Example

- Parametric polymorphic type inference

- $\text{let } x = \lambda x.x \text{ in } \quad // x : \forall \alpha.\alpha \rightarrow \alpha$
- $x \ 3; \quad // x : \beta \rightarrow \beta, \beta = \text{int}$
- $x (\lambda y.y) \quad // x : \gamma \rightarrow \gamma, \gamma = \delta \rightarrow \delta$

- This would be untypable in a monomorphic type system

CMSC 631

54

Algorithm W

- A type inference algorithm that explicitly solves the equality constraints on-line
- Instead of implicit global substitution (like we used before), threads the substitution through the inference
- In practice, use previous algorithm, plus generalize at let and instantiate at variable uses.
 - Solve for the type of e1, generalize it, then instantiate its solution when doing inference on e2

CMSC 631

55

An Imperative Language

- $e ::= x \mid \lambda x.e \mid e e$
- $\mid \text{ref } e$ allocation
- $\mid !e$ dereference
- $\mid e := e$ assignment
- $\mid e; e$ sequencing
- Notice that this is not C
 - Variables cannot be updated; only references can
 - I.e., there are no l-values or r-values
- This is a language with *updatable references*

CMSC 631

56

Examples

- $!(\text{ref } 0)$
- $\text{let } x = \text{ref } 0 \text{ in}$
- $x := !x + 1$
- $\text{let } x = \text{ref } 0 \text{ in}$
- $\lambda y. x := !x + 1; !x$

CMSC 631

57

Type Checking Rules

- $t ::= \dots \mid \text{ref } t$
- Note: in ML this type is written $t \text{ ref}$

$$\frac{A \vdash e : t}{A \vdash \text{ref } e : \text{ref } t} \qquad \frac{A \vdash e : \text{ref } t}{A \vdash !e : t}$$

$$\frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : t}$$

CMSC 631

58

Unit and the Unit Type

- Sometimes in imperative programs we write expressions that have some *side effect* but no interesting result
- To represent this directly, use *unit*:

- $e ::= \dots \mid ()$
- $t ::= \dots \mid \text{unit}$

$$\frac{}{A \vdash () : \text{unit}} \qquad \frac{A \vdash e1 : \text{ref } t \quad A \vdash e2 : t}{A \vdash e1 := e2 : \text{unit}}$$

CMSC 631

59

Operational Semantics

- Now we need to keep track of memory
 - State is a map from locations to values
 - Our redexes will be tuples $\langle \text{State}, \text{expression} \rangle$
 - As a consequence, order of evaluation matters
- As before, evaluation will yield a fully-evaluated term, also called a *value*
 - $v ::= x \mid \lambda x.e$
 - $e ::= v \mid e e \mid \text{ref } e \mid !e \mid e := e$

CMSC 631

60

Operational Semantics (cont'd)

$$\frac{}{\langle S, (\lambda x.e1) \rangle \rightarrow \langle S, (\lambda x.e1) \rangle}$$

$$\frac{\langle S, e1 \rangle \rightarrow \langle S', v1 \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v2 \rangle}{\langle S, e1; e2 \rangle \rightarrow \langle S'', v2 \rangle}$$

$$\frac{\langle S, e \rangle \rightarrow \langle S', v \rangle \quad \text{loc fresh}}{\langle S, \text{ref } e \rangle \rightarrow \langle S'[v|\text{loc}], \text{loc} \rangle}$$

CMSC 631

61

Operational Semantics (cont'd)

$$\frac{\langle S, e \rangle \rightarrow \langle S', \text{loc} \rangle}{\langle S, !e \rangle \rightarrow \langle S', S'(\text{loc}) \rangle}$$

$$\frac{\langle S, e1 \rangle \rightarrow \langle S', \text{loc} \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v \rangle}{\langle S, e1 := e2 \rangle \rightarrow \langle S''[v|\text{loc}], v \rangle}$$

$$\frac{\langle S, e1 \rangle \rightarrow \langle S', \lambda x.e \rangle \quad \langle S', e2 \rangle \rightarrow \langle S'', v \rangle \quad \langle S'', e[v|x] \rangle \rightarrow \langle S''', v' \rangle}{\langle S, e1 \text{ } e2 \rangle \rightarrow \langle S''', v' \rangle}$$

CMSC 631

62

Polymorphism and References

- Suppose we want polymorphism in our imperative language

- $e ::= x \mid n \mid \lambda x.e \mid e e \mid \text{ref } e \mid !e \mid e := e$
- $s ::= t \mid \forall \alpha.s$
- $t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \text{ref } t$

- What if we try our standard rule?

$$\frac{A \vdash e1 : t1 \quad A, x: \forall \alpha. t1 \vdash e2 : t2 \quad \alpha = \text{FV}(t1) - \text{FV}(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

CMSC 631

63

Naive Generalization is Unsound

- Example (due to Tofte)

- $\text{let } r = \text{ref } (\lambda x.x) \text{ in } \quad // r : \forall \alpha. \text{ref } (\alpha \rightarrow \alpha)$
- $r := \lambda x.x + 1; \quad // \text{ checks; use } r \text{ at } \text{ref}(\text{int} \rightarrow \text{int})$
- $(!r) \text{ true} \quad // \text{ oops! checks; use } r \text{ at } \text{ref}(\text{bool} \rightarrow \text{bool})$

- α should not be generalized, because later uses of r may place constraints on it

- Nobody realized this problem for a long time

CMSC 631

64

Solution: The Value Restriction

- Only allow *values* to be generalized

- $v ::= x \mid n \mid \lambda x.e$
- $e ::= v \mid e e \mid \text{ref } e \mid !e \mid e := e$

$$\frac{A \vdash v : t1 \quad A, x: \forall \alpha. t1 \vdash e2 : t2 \quad \alpha = \text{FV}(t1) - \text{FV}(A)}{A \vdash \text{let } x = v \text{ in } e2 : t2}$$

- Intuition: Values cannot later be updated
- This solution due to Wright and Felleisen
 - Tofte found a much more complicated solution

CMSC 631

65

Benefits of Type Inference

- Handles higher-order functions
- Handles data structures smoothly
- Works in infinite domains
 - Set of types is unlimited
- No forward/backward distinction
- Polymorphism provides context-sensitivity

CMSC 631

66

Drawbacks to Type Inference

- Flow-insensitive
 - Types are the same at all program points
 - May produce coarse results
 - Type inference failure can be hard to understand
- Polymorphism may not scale
 - Exponential in worst case
 - Seems fine in practice (witness ML)