

Type Systems
(Largely based on a paper by Luca Cardelli)

Lecture 14-16
CS263

CS 263

1

Types

- A program variable can assume a range of values during the execution of a program
- An upper bound of such a range is called a type of the variable
 - A variable of type "bool" is supposed to assume only boolean values
 - If x has type "bool" then the boolean expression "not(x)" has a sensible meaning during every run of the program

CS 263

3

Execution Errors

- The foremost purpose of types is to prevent certain types of execution errors
- Trapped execution errors
 - Cause the computation to stop immediately
 - Well-specified behavior
 - Usually enforced by hardware
 - E.g., Division by zero
 - E.g., Invoking a floating point operation with a NaN
 - E.g., Dereferencing the address 0 (on most systems)

CS 263

5

Review

- λ -calculus is as expressive as a Turing machine
- We can encode a multitude of data types in the untyped λ -calculus
- To simplify programming it is useful to add types to the language
- We now start the study of type systems in the context of the typed λ -calculus

CS 263

2

Typed and Untyped Languages

- Untyped languages
 - Do not restrict the range of values for a given variable
 - Operations might be applied to inappropriate arguments. The behavior in such cases might be unspecified
 - The pure λ -calculus is an extreme case of an untyped language (however, its behavior is completely specified)
- Typed languages
 - Variables are assigned (non-trivial) types
 - A type system keeps track of types
 - Types might or might not appear in the program itself
 - Languages can be explicitly typed or implicitly typed

CS 263

4

Execution Errors (II)

- Untrapped execution errors
 - Behavior is unspecified (depends on the state of the machine)
 - Accessing past the end of an array
 - Jumping to an address in the data segment
- A program is deemed safe if it does not cause untrapped errors
 - Languages in which all programs are safe are safe languages
- For a given language we designate a set of forbidden errors
 - A superset of the untrapped errors
 - Includes some trapped errors as well
 - E.g., null pointer dereference
 - To ensure portability across architectures

CS 263

6

Caveats

- The definition of a safe language depends on what is considered a forbidden execution error
- Many classes of execution errors are beyond the capability of (decidable) type systems
- Type systems can be designed to help catch a variety of errors
 - *traced* and *untraced* (memory management errors)
 - *static* and *dynamic* (for run-time code generation errors)
 - *unclassified* and *secret* (information flow violations)

CS 263

7

Preventing Forbidden Errors - Static Checking

- Forbidden errors can be caught by a combination of static and run-time checking
- Static checking
 - Detects errors early, before testing
 - Types provide the necessary static information for static checking
 - E.g., ML, Modula-3, Java
 - Detecting certain errors statically is undecidable in most languages

CS 263

8

Preventing Forbidden Errors - Dynamic Checking

- Required when static checking is undecidable
 - e.g., array-bounds checking
- Run-time encoding of types are still used (e.g. Lisp)
- Should be limited since it delays the manifestation of errors
- Can be done in hardware (e.g. null-pointer)

CS 263

9

Safe Languages

- There are typed languages that are not safe (weakly typed languages)
- All safe languages use types (either statically or dynamically)

	Typed		Untyped
	Static	Dynamic	
Safe	ML, Java, ...	Lisp, Scheme	λ -calculus
Unsafe	C, C++, ...	?	Assembly

- We will be concerned mainly with statically typed languages

CS 263

10

Why Typed Languages?

- Development
 - Type checking catches early many mistakes
 - Reduced debugging time
 - Typed signatures are a powerful basis for design
 - Typed signatures enable separate compilation
- Maintenance
 - Types act as checked specifications
 - Types can enforce abstraction
- Execution
 - Static checking reduces the need for dynamic checking
 - Safe languages are easier to analyze statically
 - the compiler can generate better code

CS 263

11

Why Not Typed Languages?

- Static type checking imposes constraints on the programmer
 - Some valid programs might be rejected
 - But often they can be made well-typed easily
 - Hard to step outside the language (e.g. OO programming in a non-OO language)
- Dynamic safety checks can be costly
 - 50% is a possible cost of bounds-checking in a tight loop
 - In practice, the overall cost is much smaller
 - Memory management must be automatic \Rightarrow need a garbage collector with the associated run-time costs
 - Some applications are justified to use weakly-typed languages

CS 263

12

Properties of Type Systems

- How do types differ from other program annotations
 - Types are more precise than comments
 - Types are more easily mechanizable than program specifications
- Expected properties of type systems:
 - Types should be enforceable
 - Types should be checkable algorithmically
 - Typing rules should be transparent
 - It should be easy to see why a program is not well-typed

CS 263

13

Why Formal Type Systems?

- Many typed languages have informal descriptions of the type systems (e.g., in language reference manuals)
- A fair amount of careful analysis is required to avoid false claims of type safety
- A formal presentation of a type system is a precise specification of the type checker
 - And allows formal proofs of type safety
- But even informal knowledge of the principles of type systems help

CS 263

14

Formalizing a Type System

A multi-step process

1. Syntax
 - Of expressions (programs)
 - Of types
 - Issues of binding and scoping
2. Static semantics (typing rules)
 - Define the typing judgment and its derivation rules
3. Dynamic semantics (e.g., operational)
 - Define the evaluation judgment and its derivation rules
4. Type soundness
 - Relates the static and dynamic semantics
 - State and prove the soundness theorem

CS 263

15

Typing Judgments

- Judgments
 - A statement J about certain formal entities
 - Has a truth value $\models J$
 - Has a derivation $\vdash J$
- A common form of the typing judgment: $\Gamma \vdash e : \tau$ (e is an expression and τ is a type)
- Γ is a set of type assignments for the free variables of e
 - Defined by the grammar $\Gamma ::= \cdot \mid \Gamma, x : \tau$
 - Usually viewed as a set of type assignments
 - Type assignments for variables not free in e are not relevant
 - E.g., $x : \text{int}, y : \text{int} \vdash x + y : \text{int}$

CS 263

16

Typing rules

- Typing rules are used to derive typing judgments
- Examples:

$$\frac{\Gamma \vdash 1 : \text{int}}{x : \tau \in \Gamma \quad \Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

CS 263

17

Typing Derivations

- A typing derivation is a derivation of a typing judgment
- Example:

$$\frac{\frac{\frac{}{x : \text{int} \vdash x : \text{int}} \quad \frac{}{x : \text{int} \vdash 1 : \text{int}}}{x : \text{int} \vdash x + 1 : \text{int}}}{x : \text{int} \vdash x + (x + 1) : \text{int}}}$$
- We say that $\Gamma \vdash e : \tau$ to denote that there is a derivation of this typing judgment
- Type checking: given Γ , e and τ find a derivation
- Type inference: given Γ and e , find τ and a derivation

CS 263

18

Proving Type Soundness

- A typing judgment has a truth value
- Define what it means for a value to have a type
 $v \in \|\tau\|$
 (e.g. $5 \in \|\text{int}\|$ and $\text{true} \in \|\text{bool}\|$)
- Define what it means for an expression to have a type
 $e \in \|\tau\|$ iff $\forall v. (e \Downarrow v \Rightarrow v \in \|\tau\|)$
- Prove type soundness
 If $\cdot \vdash e : \tau$ then $e \in \|\tau\|$
 or equivalently
 If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $v \in \|\tau\|$
- This implies safe execution (since the result of a unsafe execution is not in $\|\tau\|$ for any τ)

CS 263

19

First-Order Type Systems

CS 263

21

Static Semantics of F_1

- The typing judgment
 $\Gamma \vdash e : \tau$
- The typing rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

CS 263

23

Next

- We will give formal description of first-order type systems (no type variables)
 - Function types (simply typed λ -calculus)
 - Simple types (integers and booleans)
 - Structured types (products and sums)
 - Imperative types (references and exceptions)
 - Recursive types
- The type systems of most common languages are first-order
- The we move to second-order type systems
 - Polymorphism and abstract types

CS 263

20

Simply-Typed Lambda Calculus

- Syntax:
 - Terms $e ::= x \mid \lambda x : \tau. e \mid e_1 e_2$
 $\mid n \mid e_1 + e_2 \mid \text{iszero } e$
 $\mid \text{true} \mid \text{false} \mid \text{not } e \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$
 - Types $\tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$
- $\tau_1 \rightarrow \tau_2$ is the function type
- \rightarrow associates to the right
- Arguments have typing annotations
- This language is also called F_1

CS 263

22

Static Semantics of F_1 (Cont.)

- More typing rules

$$\frac{}{\Gamma \vdash n : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{not } e : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_t \text{ else } e_f : \tau}$$

CS 263

24

Typing Derivation in F₁

- Consider the term

$\lambda x : \text{int}. \lambda b : \text{bool}. \text{if } b \text{ then } f \ x \ \text{else } x$

- With the initial typing assignment $f : \text{int} \rightarrow \text{int}$

$$\frac{\frac{\frac{\Gamma \vdash f : \text{int} \rightarrow \text{int} \quad \Gamma \vdash x : \text{int}}{\Gamma \vdash b : \text{bool}} \quad \Gamma \vdash f \ x : \text{int} \quad \Gamma \vdash x : \text{int}}{f : \text{int} \rightarrow \text{int}, x : \text{int}, b : \text{bool} \vdash \text{if } b \text{ then } f \ x \ \text{else } x : \text{int}}}{f : \text{int} \rightarrow \text{int}, x : \text{int} \vdash \lambda b : \text{bool}. \text{if } b \text{ then } f \ x \ \text{else } x : \text{bool} \rightarrow \text{int}}}{f : \text{int} \rightarrow \text{int} \vdash \lambda x : \text{int}. \lambda b : \text{bool}. \text{if } b \text{ then } f \ x \ \text{else } x : \text{int} \rightarrow \text{bool} \rightarrow \text{int}}$$

Where $\Gamma = f : \text{int} \rightarrow \text{int}, x : \text{int}, b : \text{bool}$

CS 263

25

Type Checking in F₁

- Type checking is easy because

- Typing rules are syntax directed
- Typing rules are compositional
- All local variables are annotated with types

- In fact, type inference is also easy for F₁

- Without type annotations an expression does not have a unique type

$\cdot \vdash \lambda x. x : \text{int} \rightarrow \text{int}$

$\cdot \vdash \lambda x. x : \text{bool} \rightarrow \text{bool}$

CS 263

26

Operational Semantics of F₁

- Judgment:

$$e \Downarrow v$$

- Values

$$v ::= n \mid \text{true} \mid \text{false} \mid \lambda x : \tau. e$$

- The evaluation rules ...

CS 263

27

Operational Semantics of F₁ (Cont.)

- Call-by-value evaluation rules (sample)

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

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{n \Downarrow n} \quad \frac{e_1 \Downarrow \text{true} \quad e_t \Downarrow v}{\text{if } e_1 \text{ then } e_t \text{ else } e_f \Downarrow v}$$

$$\frac{e_1 \Downarrow \text{false} \quad e_f \Downarrow v}{\text{if } e_1 \text{ then } e_t \text{ else } e_f \Downarrow v}$$

Evaluation undefined for ill-typed programs !

CS 263

28

Type Soundness for F₁

- Theorem:

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

- Also called, subject reduction theorem, type preservation theorem

- 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!

CS 263

29

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 $e_1 \Downarrow \dots$ we have $\cdot, x : \tau_2 \vdash e'_1 : \tau$

- From IH on $e_2 \Downarrow \dots$ 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)

CS 263

30

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
- Homework: The execution always terminates in F_1

CS 263

31

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

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

$$\begin{aligned} n_1 + n_2 &\rightarrow n_1 \text{ plus } n_2 \\ \text{if true then } e_1 \text{ else } e_2 &\rightarrow e_1 \\ \text{if false then } e_1 \text{ else } e_2 &\rightarrow e_2 \\ (\lambda x:\tau. e_1) v_2 &\rightarrow [v_2/x]e_1 \end{aligned}$$
- and one global reduction rule

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

CS 263

32

Contextual Semantics for F_1

- Decomposition lemmas:
 - If $\cdot \vdash e : \tau$ and e is not a 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 $r \rightarrow e'$ 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

CS 263

33

Contextual Semantics of F_1

- Type preservation theorem
 - If $\cdot \vdash e : \tau$ and $e \rightarrow e'$ 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: $e \rightarrow e'$
- Progress theorem says that execution can make progress on a well typed expression
- Furthermore, due to type preservation we know that the execution of a well typed expression never gets stuck
 - this is a common way to state and prove type safety of a language

CS 263

34

Product Types - Static Semantics

- Extend the syntax with (binary) tuples

$$\begin{aligned} e &::= \dots \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e \\ \tau &::= \dots \mid \tau_1 \times \tau_2 \end{aligned}$$
 - This language is sometimes called F_1^*
- Same typing judgment $\Gamma \vdash e : \tau$

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

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2}$$

CS 263

35

Product Types: Dynamic Semantics and Soundness

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

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

$$\frac{e \Downarrow (v_1, v_2)}{\text{fst } e \Downarrow v_1} \quad \frac{e \Downarrow (v_1, v_2)}{\text{snd } e \Downarrow v_2}$$

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

$$\begin{aligned} \text{fst } (v_1, v_2) &\rightarrow v_1 \\ \text{snd } (v_1, v_2) &\rightarrow v_2 \end{aligned}$$
- Type soundness holds just as before

CS 263

36

Records

- Records are like tuples with labels
- New form of expressions

$$e ::= \dots \mid \{L_1 = e_1, \dots, L_n = e_n\} \mid e.L$$
- New form of values

$$v ::= \{L_1 = v_1, \dots, L_n = v_n\}$$
- New form of types

$$\tau ::= \dots \mid \{L_1 : \tau_1, \dots, L_n : \tau_n\}$$
- ... follows the model of F_1^\times
 - typing rules
 - derivation rules
 - type soundness

CS 263

37

Sum Types

- We need types of the form
 - either an int or a float
 - either 0 or a pointer
 - either true or false
 - These are called disjoint union types
- New form of expressions and types

$$e ::= \dots \mid \text{injl } e \mid \text{inj } e \mid \text{case } e \text{ of } \text{injl } x \rightarrow e_1 \mid \text{inj } y \rightarrow e_2$$

$$\tau ::= \dots \mid \tau_1 + \tau_2$$
 - 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: x is bound in e_1 and y is bound in e_2

CS 263

38

Examples with Sum Types

- Consider the type "unit" with a single element called *
- The type "optional integer" defined as "unit + int"
 - Useful for optional arguments or return values
 - No argument: $\text{injl } *$
 - Argument is 5: $\text{inj } 5$
 - To use the argument you must test the kind of argument
 - case arg of $\text{injl } x \Rightarrow \text{"no_arg_case"} \mid \text{inj } y \Rightarrow \text{"...y..."}$
 - injl and inj are tags and case is tag checking
- Bool is a union type: $\text{bool} = \text{unit} + \text{int}$
 - true is $\text{injl } *$
 - false is $\text{inj } *$
 - if e then e_1 else e_2 is $\text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{inj } y \Rightarrow e_2$
 - Check the equivalence of the static and dynamic semantics

CS 263

39

Static Semantics of Sum Types

- New typing rules

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{injl } e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inj } e : \tau_1 + \tau_2}$$

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

- Types are not unique anymore
 - $\text{injl } 1 : \text{int} + \text{bool}$
 - $\text{injl } 1 : \text{int} + (\text{int} \rightarrow \text{int})$
 - this complicates type checking, but still doable

CS 263

40

Dynamic Semantics of Sum Types

- New values $v ::= \dots \mid \text{injl } v \mid \text{inj } v$
- New evaluation rules

$$\frac{e \Downarrow v}{\text{injl } e \Downarrow \text{injl } v} \quad \frac{e \Downarrow v}{\text{inj } e \Downarrow \text{inj } v}$$

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

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

CS 263

41

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

CS 263

42

Types for Imperative Features

- We looked at types for pure functional languages
- Now we look at types for imperative features
- Such types are used to characterize non-local effects
 - assignments
 - exceptions
- Contextual semantics is useful here

CS 263

43

Reference Types

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

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

$$\tau ::= \dots \mid \tau \text{ ref}$$
 - $\text{ref } e$ - 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

CS 263

44

Global Effects with Reference Cells

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

$$(\lambda f : \text{int} \rightarrow \text{int ref. } !(f 5)) (\lambda x : \text{int. ref } x : \text{int})$$
- 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
- To model reference cells we must extend the evaluation model

CS 263

45

Modeling References

- A heap is a mapping from addresses to values

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

$$p ::= \text{heap } h \text{ in } e$$
 - The initial program is "heap \emptyset 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

CS 263

46

Static Semantics of References

- Typing rules for expressions:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e) : \tau \text{ ref}} \quad \frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}$$
- and for programs

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

where $\Gamma = a_1 : \tau_1 \text{ ref}, \dots, a_n : \tau_n \text{ ref}$
 and $h = a_1 \leftarrow v_1 : \tau_1, \dots, a_n \leftarrow v_n : \tau_n$

CS 263

47

Contextual Semantics for References

- Addresses are values: $v ::= \dots \mid a$
- 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
 - $\text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, a \leftarrow v : \tau \text{ in } H[a]$
 - where a is fresh
 - $\text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v]$
 - where $a \leftarrow v : \tau \in h$
 - $\text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \leftarrow v] \text{ in } H[*]$
 - where $h[a \leftarrow v]$ means a heap like h except that the part " $a \leftarrow v_1 : \tau^1$ " in h is replaced by " $a \leftarrow v : \tau$ "
- Global rules are used to propagate the effects of a write to the entire program

CS 263

48

Example with References

- Consider the evaluation (the redex is underlined)
 - heap · in $(\lambda f:\text{int} \rightarrow \text{int} \text{ ref. } \underline{!(f\ 5)}) (\lambda x:\text{int} \text{ ref } x : \text{int})$
 - heap · in $!(\lambda x:\text{int} \text{ ref } x : \text{int}) \underline{5}$
 - heap · in $!(\text{ref } 5 : \text{int})$
 - heap $a = 5 : \text{int}$ in $!a$
 - heap $a = 5 : \text{int}$ in 5
- The resulting program has a useless memory cell
- An equivalent result would be
heap \emptyset in 5
- This is a simple way to model garbage collection

CS 263

49

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
- 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

CS 263

50

Modeling Exceptions

- Syntax
 - $e ::= \dots \mid \text{raise } e \mid \text{try } e_1 \text{ handle } x \Rightarrow e_2$
 - $\tau ::= \dots \mid \text{exn}$
- 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
 - $1 + \text{raise } 2$ 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?

CS 263

51

Example with Exceptions

- A (strange) factorial function
 - let $f = \lambda x:\text{int} \lambda \text{res}:\text{int} \text{ if } x = 0 \text{ then}$
 - raise res
 - else
 - $f (x - 1) (\text{res} * x)$
 - in try f 5 1 handle $x \Rightarrow x$
- The function returns in one step from the recursion
- The top-level handler catches the exception and turns it into a regular result

CS 263

52

Typing Exceptions

- New typing rules
$$\frac{\Gamma \vdash e : \text{exn}}{\Gamma \vdash \text{raise } e : \tau}$$
$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma, x : \text{exn} \vdash e_2 : \tau}{\Gamma \vdash \text{try } e_1 \text{ handle } x \Rightarrow e_2 : \tau}$$
- 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

CS 263

53

Dynamics of Exceptions

- The result of evaluation can be an uncaught exception
 - Evaluation answers: $a ::= v \mid \text{uncaught } v$
 - "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

CS 263

54

Contextual Semantics for Exceptions

- Contexts
 - $H ::= \bullet \mid H e \mid v H \mid \text{raise } H \mid \text{try } H \text{ handle } x \Rightarrow e$
- Propagating contexts
 - Contexts that propagate exceptions to their own enclosing contexts
 - $P ::= \bullet \mid P e \mid v P \mid \text{raise } P$
- 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:
 - $H[(\lambda x:\tau. e) v]$
 - $H[\text{try } v \text{ handle } x \Rightarrow e]$
 - $H[\text{try } P[\text{raise } v] \text{ handle } x \Rightarrow e]$
 - $P[\text{raise } v]$

CS 263

55

Contextual Semantics for Exceptions (Cont.)

- Small-step reduction rules
 - $H[(\lambda x:\tau. e) v] \rightarrow H[[v/x] e]$
 - $H[\text{try } v \text{ handle } x \Rightarrow e] \rightarrow H[v]$
 - $H[\text{try } P[\text{raise } v] \text{ handle } x \Rightarrow e] \rightarrow H[[v/x] e]$
 - $P[\text{raise } v] \rightarrow \text{uncaught } v$
- The handler is ignored if the body of try completes normally
- A raised exception propagates (in one step) to the closer enclosing handler or to the top of the program

CS 263

56

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

CS 263

57

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
- 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

CS 263

58

Continuation Uses in Real Life

- Consider that you walk down the street to a fork
- You save a continuation "right" for going right
- But you go left (with the "right" continuation in hand)
- You encounter a mean dog. Dog bites you badly.
- You save a continuation "bitten" for calling the ambulance, etc.
- You decide to invoke the "right" continuation
- So, you go right (unbitten, but with the "bitten" continuation in hand)
- A train hits you!
- With your last breath, you invoke the "bitten" continuation

CS 263

59

Continuations

- Syntax:
 - $e ::= \text{callcc } k \text{ in } e \mid \text{throw } e_1 e_2$
 - $\tau ::= \dots \mid \tau \text{ cont}$
- $\tau \text{ cont}$ - the type of a continuation that expects a τ
- $\text{callcc } k \text{ in } e$ - 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 " $\text{callcc } k \text{ in } e$ " returns
 - k is bound in e
- $\text{throw } e_1 e_2$ - evaluates e_1 to a continuation, e_2 to a value and invokes the continuation with the value of e_2

CS 263

60

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 than exceptions
 - The destination is not tied to the call stack

CS 263

61

Static Semantics of Continuations

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

$$\frac{\Gamma \vdash e_1 : \tau \text{ cont} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{throw } e_1 \ e_2 : \tau'}$$

- 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

CS 263

62

Dynamic Semantics of Continuations

- Use contextual semantics
 - Contexts are now manipulated directly
 - Contexts are values of type τ cont
- Contexts

$$H ::= \bullet \mid H \ e \mid v \ H \mid \text{throw } H_1 \ e_2 \mid \text{throw } v_1 \ H_2$$
- Evaluation rules
 - $H[(\lambda x.e) v] \rightarrow H[[v/x] e]$
 - $H[\text{callcc } k \text{ in } e] \rightarrow H[[H/k] e]$
 - $H[\text{throw } H_1 \ v_2] \rightarrow H_1[v_2]$
- Callcc duplicates the current continuation
- Note that throw abandons its own context

CS 263

63

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)

CS 263

64

Continuations. 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)

CS 263

65

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, λrg.h (rg, k)) }
```
- Advantages:
 - interesting compilation scheme (supports callcc easily)
 - no need for a stack, can have multiple return addresses (e.g., for an error case)
 - interesting programming style

CS 263

66

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 $\text{cps}(e, k)$ as the code that computes e in CPS and passes the result to continuation k
 - $\text{cps}(x, k) = k \ x$
 - $\text{cps}(n, k) = k \ n$
 - $\text{cps}(e_1 + e_2, k) = \text{cps}(e_1, \lambda n_1. \text{cps}(e_2, \lambda n_2. k \ (n_1 + n_2)))$
 - $\text{cps}(\lambda x.e, k) = k \ (\lambda x \lambda k'. \text{cps}(e, k'))$
 - $\text{cps}(e_1 e_2, k) = \text{cps}(e_1, \lambda f_1. \text{cps}(e_2, \lambda v_2. f_1 \ v_2 \ k))$
- Example: $\text{cps}(\text{h}(\text{g}(5)), k) = \text{g}(5, \lambda x. \text{h} \ x \ k)$
 - Notice the order of evaluation being explicit

CS 263

67