

Types in Programming Languages

- Stansifer Ch. 4
- Cardelli Sec. 1

1

Types

- Organization of untyped values
 - Untyped universes: bit strings, S-expr, ...
 - Categorize based on usage and behavior
- Type = set of computational objects with uniform behavior
- Constraints to enforce correctness
 - Check the applicability of operations
 - Should not try to multiply two strings
 - Should not use an integer as a pointer

2

Examples of Type Checking

- Built-in operators should get operands of correct types
- Type of left-hand side must agree with the value on the right-hand side
- Procedure calls: number and type of actual arguments
- Return type should match returned value

3

Static Typing

- Statically typed languages: expressions in the code have **static types**
 - static type = claim about run-time values
 - Types are either **declared** or **inferred**
 - Examples: C, C++, Java, ML, Pascal, Modula-3
- A statically typed language typically does some form of static type checking
 - May also do dynamic type checking
 - e.g. Java checks for array indices out of bounds and for null pointers

4

Dynamic Typing

- Dynamically-typed languages: expressions in the code do not have static types
 - Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
- Dynamic type checking
 - Before an operation is performed at run time
 - Typical implementation: keep program values "tagged" with a type, and check the type tag before a value is used in an operation

5

Strongly vs. Weakly Typed

- **Strongly typed** languages: type-incorrect operations are not performed at run time
 - Things cannot "go wrong": no type errors
 - Certain run-time errors are possible but clearly marked as such
 - i.e. array index out of bounds, null pointer
 - C/C++: weakly typed, Java: strongly typed
- Independent of static vs. dynamic
 - Lisp is strongly, dynamically typed
 - Forth is weakly, dynamically typed

6

Examples of Types

- Integers
- Arrays of Integers
- Pointers to Integers
- Records with fields `int x` and `int y`
- Objects of class `C` or a subclass of `C`
- Functions from any list to integers

7

Types as Sets of Values

- Integers
 - Any number that can be represented in 32 bits in signed two's-complement
 - $\text{int} = \{-2^{32} \dots 2^{32} - 1\}$
- Class type
 - Any object of class `C` or a subclass of `C`
 - `C` = set of all instances of `C` or of a subclass of `C`
- Subtypes are subsets

8

Types as Interfaces

- Integers
 - `int` = set of all objects to which one can apply integer addition, multiplication, etc.
- Class types
 - `C` = set of all objects that understand the public methods of class `C`
- Types in Java or C++
 - **Implementation types**: integers, other base types, class types
 - **Interface types**: abstract class types, interface types in Java

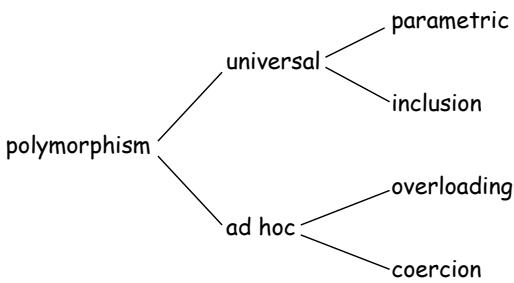
9

Monomorphism vs. Polymorphism

- Greek:
 - mono = single
 - poly = many
 - morph = form
- Monomorphism
 - every value belongs to exactly one type
- Polymorphism
 - a value can belong to multiple types

10

Types of Polymorphism



11

Coercion

- Values of one type are silently converted to another type
 - e.g. addition: $3.0 + 4$: converts 4 to 4.0
 - $\text{int} \times \text{int} \rightarrow \text{int}$ or $\text{real} \times \text{real} \rightarrow \text{real}$
- In a context where the type of an expression is not appropriate
 - either an automatic coercion (conversion) to another type is performed automatically
 - or if not possible: compile-time error

12

Coercions

- Widening
 - coercing a value into a "larger" type
 - e.g., **int** to **float**, subclass to superclass
- Narrowing
 - coercing a value into a "smaller" type
 - loses information, e.g., **float** to **int**
 - PL/I: $1/3 + 25$ has value **5.3333333333**

13

Widening Primitive Conversions in Java

- Widening primitive conversions
 - byte to short, int, long, float, or double
 - short to int, long, float, or double
 - char to int, long, float, or double
 - int to long, float, or double
 - long to float or double
 - float to double
- "integral type to integral type" and "float to double" do not lose any information

14

Widening Primitive Conversions in Java

- Language Spec says
 - Conversion of an int or long value to float, or of a long value to double, may result in loss of precision
 - The result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode

15

Contexts for Widening Conversions

- **Assignment conversion:** when the value of an expression is assigned to a variable
- **Method invocation conversion:** applied to each argument value in a method or constructor invocation
 - The type of the argument expression must be converted to the type of the corresponding formal parameter
- **Casting conversion:** applied to the operand of a cast operator: (float) 5

16

Contexts for Widening Conversions

- **Numeric Promotion:** converts operands of a numeric operator to a common type
- **Example: binary numeric promotion**
 - e.g. +, -, *, etc.
 - If either operand is double, the other is converted to double
 - Otherwise, if either operand is of type float, the other is converted to float
 - Otherwise, if either operand is of type long, the other is converted to long
 - Otherwise, both are converted to type int

17

Narrowing Conversions

- **Narrowing primitive conversions in Java**
 - e.g. long to byte, short, char, or int
 - float to byte, short, char, int, or long
 - double to byte, short, char, int, long, or float
- **Examples of loss of information**
 - int to short loses high bits
 - int not fitting in byte changes sign and magnitude
 - double too small for float underflows to zero

18

Overloading

- Multiple definitions of the same name
- E.g. name + for several operations
 - double × double → double (binary plus)
 - float × float → float
 - long × long → long
 - int × int → int
 - double → double (unary plus)
 - float → float
 - long → long
 - int → int

19

Overloading

- Method selection at compile time

```
class D extends C { ... }  
int foo (C x) { return 0; }  
int foo (D x) { return 1; }
```

```
C p = new D();  
int i = foo(p); // executes C.foo
```

20

Multimethods

- Method selection at run time
 - e.g. CLOS, Dylan, Cecil, Brew

```
class D extends C { ... }  
int foo (C x) { return 0; }  
int foo (D x) { return 1; }
```

```
C p = new D();  
int i = foo(p); // executes D.foo
```

21

Overloading vs. Overriding in Java

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; } }
class RealPoint extends Point {
    float x = 0.0, y = 0.0;
    void move(int dx, int dy)
        { move((float)dx, (float)dy); }
    void move(float dx, float dy)
        { x += dx; y += dy; } }
```

22

Overloading vs. Overriding in Java

```
public static void main(String[] args) {
    RealPoint rp = new RealPoint();
    // compile-time resolution: the most specific
    // target method
    rp.move(1.5, 1.5); → RealPoint.move(float,float)
    rp.move(2,2); → RealPoint.move(int,int)
    Point p = rp;
    p.move(3,3); → compile time: Point.move(int,int)
                 → run time: RealPoint.move(int,int)
}
```

23

Overloading: Most Specific Method

```
class Test {
    static void test(RealPoint p, Point q) { ... }
    static void test(Point p, RealPoint q) { ... }
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        test(rp,rp); // compile-time error
    }
}
```

24

Parametric Polymorphism

- Idea: the same function can work on parameters of different types
 - i.e. the function has multiple (function) types
- Example: identity function in ML
 - `fun id x = x;`
- has types
 - `id : int → int`
 - `id : real → real`
 - ...

25

Parametric Polymorphism

- Example: "length" function in ML
 - `fun length x =
 if null(x) then 0
 else 1 + length (tl x);`
- has types
 - `length : int list → int`
 - `length : real list → int`
 - `length : int list list → int`
 - ...

26

Generics in Ada

```
generic
  type T is private;
function Id(X : in T) return T is
begin
  return X;
end;
function IntId is new Id (INTEGER);
function FloatId is new Id (FLOAT);
```

27

ML-Style Parametric Polymorphism

- Unlike with Ada generics, we do not "instantiate" with a specific type
 - i.e. a ML polymorphic function is a real function, not a template for real functions
- Universally quantified types
 - $\text{id} : \forall t . (t \rightarrow t)$
 - $\text{length} : \forall t . (t \text{ list} \rightarrow \text{int})$
- ML syntax
 - $\text{id} : 'a \rightarrow 'a$
 - $\text{length} : 'a \text{ list} \rightarrow \text{int}$

28

Mini-ML

- Stansifer, Sect 4.5.3
- $\langle \text{expr} \rangle ::= \langle \text{id} \rangle \mid \langle \text{int} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{expr} \rangle \langle \text{expr} \rangle \mid$
 $(\text{fn } \langle \text{id} \rangle \Rightarrow \langle \text{expr} \rangle) \mid$
 $\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$
- $\langle \text{expr} \rangle \langle \text{expr} \rangle$ is function application
 - $(\text{fn } \dots)$ is definition of an anonymous function (i.e., lambda expression)
 - e.g. $(\text{fn } x \Rightarrow x)$ is the anonymous identity fun
 - $(\text{fn } x \Rightarrow x) 5$ evaluates to 5

29

Types in Mini-ML

- $\langle \text{type} \rangle ::= \langle \text{type variable} \rangle \mid \text{int} \mid \text{bool} \mid$
 $\langle \text{type} \rangle \rightarrow \langle \text{type} \rangle$
- Goal: define the type of each expression
 - An expression may have infinite # of types
 - $(\text{fn } x \Rightarrow x)$ has types $\text{int} \rightarrow \text{int}$, $\text{bool} \rightarrow \text{bool}$, and $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$, ...
 - Solution: define a proof system
 - Any type derivable ("provable") in the proof system is a valid type for the expression

30

Typing Judgments

- Typing judgments: $A \vdash e : t$
 - Just notation: remember $\langle ae, \sigma \rangle \rightarrow n$ from operational semantics?
- A is a type assignment
 - set of pairs (identifier, type)
 - Similar to state σ in operational semantics
- The judgment says: under type assignment A , expression e has type t

31

Typing Rules

_____ if A assigns a type to identifier x
 $A \vdash x : A(x)$

$A[x \leftarrow t1] \vdash e1 : t2$

$A \vdash (\text{fn } x \Rightarrow e1) : t1 \rightarrow t2$

$A[x \leftarrow t1]$ is $\{(x, t1)\} \cup \{(y, A(y)) \mid y \neq x\}$; we used similar notation in operational semantics

Example: $\emptyset \vdash (\text{fn } x \Rightarrow x) : \text{int} \rightarrow \text{int}$

because $\emptyset[x \leftarrow \text{int}] \vdash x : \text{int}$

32

Typing Rules

$A \vdash e1 : t2 \rightarrow t$ $A \vdash e2 : t2$

$A \vdash e1 \ e2 : t$

$A \vdash e1 : \text{bool}$ $A \vdash e2 : t$ $A \vdash e3 : t$

$A \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t$

And of course

$A \vdash \text{true} : \text{bool}$ $A \vdash \text{false} : \text{bool}$

$A \vdash 0 : \text{int}$ $A \vdash 1 : \text{int}$ $A \vdash 2 : \text{int}$...

33

Example

$\emptyset \vdash (\text{fn } x \Rightarrow \text{if true then } 0 \text{ else } 1) : ?$

Consider $\emptyset[x \leftarrow 'a] = \{(x, 'a)\}$

$\{(x, 'a)\} \vdash \text{if true then } 0 \text{ else } 1 : \text{int}$

$\emptyset \vdash (\text{fn } x \Rightarrow \text{if true then } 0 \text{ else } 1) : 'a \rightarrow \text{int}$

Of course, we can also derive, for example,

$\emptyset \vdash (\text{fn } x \Rightarrow \text{if true then } 0 \text{ else } 1) : \text{int} \rightarrow \text{int}$

$\emptyset \vdash (\text{fn } x \Rightarrow \text{if true then } 0 \text{ else } 1) : \text{bool} \rightarrow \text{int}$

$\emptyset \vdash (\text{fn } x \Rightarrow \text{if true then } 0 \text{ else } 1) : (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$

34

Summary of Type Rules

- If an expression does not have a type, there will be no derivations in the system
 - Example: $(\text{fn } x \Rightarrow x \ x)$; problem ...
- If more than one type can be derived for an expression: **polymorphic** expression
- If only one type can be derived, it is a **monomorphic** expression

35

Type Inference

- Given some expression, can we infer
 - any type?
 - the most general type?
 - $'a \rightarrow 'a$ is more general than $\text{int} \rightarrow \text{int}$
- Milner, 1978: algorithm W
 - **Claim 1:** if $W(A, e)$ succeeds with t , then $A \vdash e : t$ is derivable in the type system
 - **Claim 2:** If $A \vdash e : t$ is derivable in the type system, $W(A, e)$ produces a type t_2 such that t is an instance of t_2

36

More Examples

- $(\text{fn } x \Rightarrow \text{if } x \text{ then } 4 \text{ else true})$
 - bad news
- $(\text{fn } f \Rightarrow f \ 2)$
 - $(\text{int} \rightarrow 'a) \rightarrow 'a$
- $(\text{fn } f \Rightarrow f \ 2) (\text{fn } x \Rightarrow x)$
 - type int
- $(\text{fn } x \Rightarrow (\text{fn } y \Rightarrow x \ y))$
 - $('a \rightarrow 'b) \rightarrow ('a \rightarrow 'b)$

37

Pairs

- Addition to the language:
 $\langle \text{expr} \rangle ::= \dots \mid (\langle \text{expr} \rangle , \langle \text{expr} \rangle)$
- Addition to the type system
 - e.g. $\text{int} * \text{bool}$: pair of int + bool
- $\langle \text{type} \rangle ::= \dots \mid \langle \text{type} \rangle * \langle \text{type} \rangle$
- Type rule
$$\frac{A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2}{A \vdash (e_1 , e_2) : t_1 * t_2}$$

38

let

- **let** $\langle \text{id} \rangle = \langle \text{expr} \rangle_1$ **in** $\langle \text{expr} \rangle_2$ **end**
 - e.g. **let** $f = (\text{fn } x \Rightarrow x)$ **in** $f \ 2$ **end**
- Computationally equivalent to
 $(\text{fn } \langle \text{id} \rangle \Rightarrow \langle \text{expr} \rangle_2) (\langle \text{expr} \rangle_1)$
 - $(\text{fn } f \Rightarrow f \ 2) (\text{fn } x \Rightarrow x)$
- Type rule
$$\frac{A \vdash e_1 : t_1 \quad A[x \leftarrow t_1] \vdash e_2 : t_2}{A \vdash \text{let } \text{id} = e_1 \text{ in } e_2 \text{ end} : t_2}$$

39

Example

- `let f = (fn x => x) in (f 2, f true) end`
 - the type is `int*bool`, and here is why:
- $\emptyset \vdash (fn\ x \Rightarrow x) : 'a \rightarrow 'a$
- $\{(f, 'a \rightarrow 'a)\} \vdash f\ 2 : int$
- $\{(f, 'a \rightarrow 'a)\} \vdash f\ true : bool$
- $\{(f, 'a \rightarrow 'a)\} \vdash (f\ 2, f\ true) : int * bool$
- $\emptyset \vdash \text{let } \dots \text{ end} : int * bool$

40

Different Typing

- `(fn f => (f 2, f true)) (fn x => x)`
 - operationally equivalent to the last slide
- There is no type for this expression!
- Problem
 - To infer a type, we need to infer a function type for `(fn ...)`
 - What is the domain of this function type?
 - i.e., the type of `f`
- Convince yourself that it does not work

41

Inclusion Polymorphism

- Subtype relationships among types
- A computational object of a subtype may be used in any context that expects an object of a supertype
- Typical examples
 - Imperative languages: record types
 - Object-oriented languages: class types

42

Subtyping in Java

- Subtyping between class types
class B { int foo () { ... } }
class C **extends** B { int foo () { ... } }
B p = new C();
int i = p.foo();
- Interface types
 - interface X { bool bar(); }
 - class A **implements** X { bool bar() { ... } }
 - X x = new A(); bool b = x.bar();

43

Structural Subtyping in Amber

- Toy research language
- $t1 \leq t2$ means "t1 is a subtype of t2"
- Record types (similar to struct in C)
{k: Int, l: String}
- Variant record types (similar to C unions)
[k: Int, l: String, m: Bool]
- Function types
{k: Int, l: String} \rightarrow {m: Int}

44

Structural Subtyping for Records

- Record types t1 and t2
- $t1 \leq t2$ iff for every field $m:s2$ in t2, there is a field $m:s1$ in t1 with $s1 \leq s2$
 - {x: Int, y: String} \leq {x: Int}
- Variant record types t1 and t2
- $t1 \leq t2$ iff for every field $m:s1$ in t1, there is a field $m:s2$ in t2 with $s1 \leq s2$
 - [x: Int] \leq [x: Int, y: String]

45

Subtyping for Functions

- Function types $s1 \rightarrow t1$ and $s2 \rightarrow t2$
- $s1 \rightarrow t1 \leq s2 \rightarrow t2$ iff $s2 \leq s1$ and $t1 \leq t2$
 - contravariant argument types
 - covariant return types

$\{a:\text{Int}\} \rightarrow \{c:\text{Int},d:\text{String}\} \leq$

$\{a:\text{Int},b:\text{String}\} \rightarrow \{c:\text{Int}\}$

46
