Types in Programming Languages

- Stansifer Ch. 4
- Cardelli Sec. 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

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
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

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

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
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

Types as Sets of Values

- Integers
  - Any number than can be represented in 32 bits in signed two's-complement
    - `int = [-2^{31}...2^{31} - 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

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
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

Types of Polymorphism

- parametric
- universal
- inclusion
- overloading
- ad hoc
- coercion

Coercion

- Values of one type are silently converted to another type
  - e.g. addition: 3.0 + 4: converts 4 to 4.0
    - int \times int \rightarrow int \text{ or } real \times real \rightarrow 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
**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 5.33333333333

---

**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

---

**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
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`

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

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
Overloading

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

Overloading

- Method selection at compile time

```java
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
```

Multimethods

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

```java
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
```
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; }
}

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)
}

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
    }
}
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
    - ...

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
    - ...

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);
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
  - id : ∀ t . (t → t)
  - length : ∀ t . (t list → int)
- ML syntax
  - id : 'a → 'a
  - length : 'a list → int

Mini-ML

- Stansifer, Sect 4.5.3
- <expr> ::= <id> | <int> | <bool> | <expr> <expr> | (fn <id> => <expr>) | if (<expr>) then <expr> else <expr>
- <expr> <expr> is function application
- (fn ...) is definition of an anonymous function (i.e., lambda expression)
  - e.g. (fn x => x) is the anonymous identity function
  - (fn x => x) 5 evaluates to 5

Types in Mini-ML

- Goal: define the type of each expression
- An expression may have infinite # of types
  - (fn x => x) has types int → int, bool → bool,
    and (int → int) → (int → int), ...
- Solution: define a proof system
  - Any type derivable ("provable") in the proof system is a valid type for the expression
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 \( \sigma \) in operational semantics
- The judgment says: under type assignment \( A \), expression \( e \) has type \( t \)

Typing Rules

\[
\text{if } A \text{ assigns a type to identifier } x \\
A \vdash x : A(x) \\
A[x \leftarrow t1] \vdash e1 : t2 \\
A \vdash (\text{fn } x \rightarrow e1) : t1 \rightarrow t2
\]

\( A[x\leftarrow t1] \) 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 \( \emptyset[x \leftarrow \text{int}] \vdash x : \text{int} \)

Typing Rules

\[
\frac{A \vdash e1 : t2 \rightarrow t \quad A \vdash e2 : t2}{A \vdash e1 \ e2 : t} \\
\frac{A \vdash e1 : \text{bool} \quad A \vdash e2 : t \quad A \vdash e3 : t}{A \vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t}
\]

And of course
\[
\frac{A \vdash \text{true} : \text{bool} \quad A \vdash \text{false} : \text{bool}}{A \vdash 0 : \text{int} \quad A \vdash 1 : \text{int} \quad A \vdash 2 : \text{int} \;
\ldots
\}
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,

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

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

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

Summary of Type Rules

- If an expression does not have a type, there will be no derivations in the system
  - Example: \((\text{fn } x \Rightarrow x); \) 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

Type Inference

- Given some expression, can we infer any type?
  - the most general type?
    - 'a -> 'a is more general than int -> 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 \(t_2\)
### More Examples

- (fn x => if x then 4 else true)
- bad news
- (fn f => f 2)
- (int ⇒ 'a) ⇒ 'a
- (fn f => f 2) (fn x => x)
- type int
- (fn x => (fn y => x y))
- ('a ⇒ 'b) ⇒ ('a ⇒ 'b)

### Pairs

- Addition to the language:
  `<expr> ::= … | (expr, expr)`
- Addition to the type system
  - e.g. int*bool: pair of int + bool
  `<type> ::= … | type * type`
- Type rule
  \[ A ⊢ e_1 : t_1 \quad A ⊢ e_2 : t_2 \]
  \[ A ⊢ (e_1, e_2) : t_1 \times t_2 \]

### let

- let `<id>` = `<expr>_1` in `<expr>_2` end
  - e.g. let f = (fn x => x) in f 2 end
- Computationally equivalent to
  \( (fn \langle id \rangle => \langle expr>_2) \langle \langle expr>_1 \rangle \)
  \( (fn f => f 2) (fn x => x) \)
- Type rule
  \[ A ⊢ e_1 : t_1 \quad A[x\leftarrow t_1]-e_2 : t_2 \]
  \[ A ⊢ let \ id = e_1 in e_2 end : t_2 \]
Example

- let f = (fn x => x) in (f 2, f true) end
  - the type is int*bool, and here is why:
    - ∅ ⊢(fn x => x) : 'a → 'a
    - {}(f, 'a→'a)) ⊢ f 2 : int
    - {}(f, 'a→'a)) ⊢ f true : bool
    - {}(f, 'a→'a)) ⊢(f 2, f true) : int*bool
    - ∅ ⊢ let ... end : int*bool

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

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
**Subtyping in Java**

- Subtyping between class types
  
  ```java
  class B { int foo () { ... } }
  class C extends B { int foo () { ... } }
  B p = new C();
  int i = p.foo();
  ```

- Interface types
  
  ```java
  interface X { bool bar(); }
  class A implements X { bool bar() { … } }
  X x = new A(); bool b = x.bar();
  ```

**Structural Subtyping in Amber**

- Toy research language
- \( t_1 \leq t_2 \) means "\( t_1 \) is a subtype of \( t_2 \)"
- 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\}
  ```

**Structural Subtyping for Records**

- Record types \( t_1 \) and \( t_2 \)
  
  \( t_1 \leq t_2 \) iff for every field \( m:s_2 \) in \( t_2 \), there is a field \( m:s_1 \) in \( t_1 \) with \( s_1 \leq s_2 \)
  
  ```
  (x: Int, y: String) \leq (x: Int)
  ```

- Variant record types \( t_1 \) and \( t_2 \)
  
  \( t_1 \leq t_2 \) iff for every field \( m:s_1 \) in \( t_1 \), there is a field \( m:s_2 \) in \( t_2 \) with \( s_1 \leq s_2 \)
  
  ```
  [x:Int] \leq [x:Int, y:String]
  ```
Subtyping for Functions

- Function types $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$
- $s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2$ iff $s_2 \leq s_1$ and $t_1 \leq t_2$
  - contravariant argument types
  - covariant return types

\[
\{a: \text{Int}\} \rightarrow \{c: \text{Int}, d: \text{String}\} \leq \\
\{a: \text{Int}, b: \text{String}\} \rightarrow \{c: \text{Int}\}
\]