Types in Programming Languages

Types in Programming Languages

Organization of untyped values

- At the lowest level, everything is a sequence of bits
- Need higher-level view: categorize these bit sequences based on usage and behavior

Type = set of run-time values with uniform behavior

Type-related constraints to enforce correctness, e.g.

- Should not try to multiply two strings
- Should not use a character value as a condition of an if-statement
- Should not use an integer as a pointer

Static Typing

Statically typed languages: expressions in the code have static types

- static type = promise about possible 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

- E.g., at compile time Java checks that the [] operator is applied to a value of type "array"

Dynamic Typing

Dynamically-typed languages: entities in the code do not have static types

- Examples: Lisp, Scheme, CLOS, Smalltalk, Perl, Python
- Entities in the code do not have declared types, and the compiler does not try to infer/check types for them

Dynamic type checking

- Before an operation is performed at run time
- E.g., in Scheme: **(+ 5 #t)** fails at run time, when the evaluation expects to see two numeric values as operands of +

Examples of Types

Integers

Arrays of integers

Pointers to integers

Records with fields **int x** and **int y**

e.g., “struct” in C

Objects of class C or a subclass of C

e.g., C++, Java, C#

Functions from any list to integers

Numeric Types [no need to remember this]

C does not specify the ranges of numeric types

- Integer types: char, short, int, long, long long
  - Includes “unsigned” versions of these
- Floating-point types: float, double, long double

Java specifies the ranges of numeric types

- byte: 8-bit signed two's complement integer [-128,+127]
- short: 16-bit signed two's complement integer [-32768,+32,767]
- int: 32-bit signed two's complement integer [-2147483648,+2147483647]
- long: 64-bit signed two's complement integer [-9223372036854775808, +9223372036854775807]
- float/double: single/double-precision 32-bit IEEE 754 floating point
- char: single 16-bit Unicode character; minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65535)

Enumeration Types [no need to remember this]

- C: a set of named integer constant values
  - Example from the C specification

    enum hue { chartreuse, burgundy, claret=20, winedark };

    /* the set of integer constant values is { 0, 1, 20, 21 } */
- Java: a fixed set of named items (not integers)

    enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

  - In reality, it is like a class: e.g., it can contain methods

Types as Sets of Run-Time Values

32-bit integers: the set of numbers than can be represented in 32 bits in signed two's-complement representation

- "**type int**" = { -2^{31}, …, $2^{31} - 1$ }

Class type C: all instances of class C and its transitive subclasses

- instance of a class C: object created by **new C**
- "**type C**" (or "**class type C**") = set of all instances of C or of any transitive subclass of C [not the same as "**class C**", which is just a blueprint for creating objects]

Static type of an expression/variable: "at run time, the expression/variable values will be from this set"

Subtypes are subsets: T2 is a subtype of T1 if T2's set of values is a subset of T1's set of values

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

Typical example: subtype polymorphism

- E.g., **class X**, **class Y extends X**, **class Z extends Y**
- Class type X = all instances of X, Y, and Z
- Class type Y = all instances of Y and Z [subtype of type X]
- An instance of class Z belongs to all three class types
- Variable of static type X is really of type "*reference to values of type X*", so it can refer to X, Y, or Z instances
  - E.g., in Java, when we say **X var;**

More Polymorphism

Parametric polymorphism

- Use a **type parameter** T; define type based on T
- Typical example: generics in C++/Java – e.g. Map<K,V>
- ML and similar functional languages

Coercion: values of one type are silently converted

- e.g., addition: **3.0 + 4** : converts **4** to **4.0**
- When the compiler sees a situation where the type of an expression is not appropriate:
  - either an automatic coercion 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** (numeric types)
- e.g., subtype to supertype (class types):
  - **class X; class Y extends X; Y var2; X var1 = var2;**

Narrowing

- coercing a value into a “smaller” type
- could lose information, e.g., **float** to **int**

Widening Primitive Conversions in Java [no need to remember this]

Widening primitive conversions: 19 cases

- 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

Does not lose information about the overall magnitude of a numeric value in the following cases, where the numeric value is preserved exactly:

- from an integral type to another integral type
- from byte, short, or char to a floating-point type
- from int to double
- from float to double

Widening Primitive Conversions in Java [no need to remember this]

[Java Lang Spec=JLS, Section 5.1.2]

A widening primitive conversion from int to float, or from long to float, or from long to double, may result in loss of precision, that is, 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 the round to nearest rounding policy [IEEE 754 standard for converting from an integer format to a floating-point format]

JSL example:

int big = 1234567890;

float approx = big;

System.out.println(big - (int)approx);

Prints: -46

indicating that information was lost during the conversion from type int to type float because values of type float are not precise to nine significant digits.

Contexts for Widening Conversions

Assignment conversion: when the value of an expression is assigned to a variable

Method call 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 [no need to remember this]
  - 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 [no need to remember this]

Narrowing primitive conversions in Java: 22 cases; some examples below:

- long to byte, short, char, or int
- float to byte, short, char, int, or long
- double to byte, short, char, int, long, or float

One example: [JLS, Section 5.1.3]: A narrowing conversion of a signed integer to an integral type T simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type T. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

Type Systems

Type System and Type Checking

Based on the set of types, a type system can prove that programs are "good" without running them

- A well-typed program will not "go wrong" at run time

Works only for some run-time errors, not all:

- E.g., cannot assure the absence of "division by zero" or "array index out of bounds" - they depend on particular values from a type
- But can catch type-related errors such as "multiplication of two booleans" error

In reality, it is a simple form of abstract interpretation

Simple Language (from the programming projects)

<expr> ::= **const** | **id** [only consider integer vars/consts; in the project also do float]

| <expr> + <expr> | <expr> - <expr>

| <expr> * <expr> | <expr> / <expr>

| (<expr>)

<cond> ::= **true** | **false** | <expr> < <expr> [also <=, >, >=, ==, !=]

| <cond> && <cond> | <cond> || <cond>

| ! <cond> | (<cond>)

Simple Abstract State

Abstract state: a map $\boldsymbol{\sigma_a}$ from vars to abstract values

A summarization of many possible concrete states

$\boldsymbol{\sigma_a}$ **: Vars →** { *Int, Float* } These are just *AnyInt* and *AnyFloat* from earlier

There is only one abstract state, defined by the declarations in the program

State never changes: e.g., if we declare a variable to be of type int, it is of type int everywhere, in all executions

Compare this with the more refined abstract interpretation from earlier, where a variable can have different abstract values at different program points

Abstract Evaluation

Abstract evaluation relation for arithmetic expressions: triples <**ae**, σ_a> → v_a

- **ae** is a parse subtree derived from <expr>
- σ_a is the abstract state defined by declarations
- v_a is an abstract value $\in$ { *Int*, *Float* }

Meaning of <**ae**, σ_a> → v_a : the evaluation of **ae** from any concrete state σ, if it completes successfully, will produce a concrete value *v* abstracted by v_a

Example: <**x+y+1**, [x↦*Int*, y↦*Int*]> → *Int*

Example: <**x*y-1.2**, [x↦*Float*, y↦*Float*]> → *Float*

Evaluation for Arithmetic Expressions

Syntax: **id** | **const** | <expr> **+** <expr> | …

$$\boxed{\langle \textbf{const}, \sigma_a \rangle \rightarrow \mathit{Int}}$$

if const.lexval is an integer constant; similarly for *Float*

$$\boxed{\langle \textbf{id}, \sigma_a \rangle \rightarrow \sigma_a(\text{id})}$$

static error if $\sigma_a(\text{id})$ is undefined; use of undeclared variable

$$\boxed{\frac{\langle ae_1, \sigma_a \rangle \rightarrow v_{a1} \quad \langle ae_2, \sigma_a \rangle \rightarrow v_{a2}}{\langle ae_1 + ae_2, \sigma_a \rangle \rightarrow v_a}}$$

$$v_a = v_{a1} +_a v_{a2}$$

Here we use abstract addition operator $+_a$ working on abstract values

Evaluation for Expressions

$+_a$	*Int*	*Float*
Int	*Int*	⚠
Float	⚠	*Float*

Same for arithmetic ops -, *, /

For boolean expressions: introduce abstract value *Bool*

$<=_a$	*Int*	*Float*
Int	*Bool*	⚠
Float	⚠	*Bool*

Same for comparison ops <, >, >=, ==, !=

For boolean ops &&, II, !: the context-free grammar already makes sure that their operands are of type *Bool*; no need for checking rules

Typed Expressions

As with other static checking: without evaluating an expression, can we guarantee that its evaluation will not produce a run-time error? (**static** a.k.a. **compile-time** analysis)

For our simple language

Type *Int* = set of all expressions that are guaranteed to evaluate to an integer value in the concrete operational semantics (no matter what the concrete state σ actually looks like at run time)

Similarly for *Float*

Typing Relation

Typing relation for arithmetic expressions: binary relation, a simplified version of the evaluation relation

If $\langle \textbf{ae}, \sigma_a \rangle \to T$ we will write $\mathrm{ae} : T$ [here *T* is *Int* or *Float*]

Type checking is defined by inference rules for the typing relation

$\boxed{\textbf{const} : T}$ if const.lexval is a constant of type *T*

$\boxed{\textbf{id} : T}$ if id is declared of type *T*

$$\frac{\mathrm{ae}_1 : T \quad \mathrm{ae}_2 : T}{\mathrm{ae}_1 + \mathrm{ae}_2 : T}$$

here *T* denotes the same type in all three expressions

$$\frac{\mathrm{ae}_1 : T \quad \mathrm{ae}_2 : T}{\mathrm{ae}_1 < \mathrm{ae}_2 : Bool}$$

If there is a derivation tree for $\mathrm{ae} : T$, the expression is well-typed and will produce a value of type *T* at run time

Static Type Safety

If a program is well-typed, we can guarantee the absence of certain type-related errors

Static type safety: **all** bad behaviors of certain type-related kinds are excluded - e.g., Java, but not C

Example: C is not type safe

```
double pi = 3.14;
double* ptr1 = &pi;
int* ptr2 = (int*) ptr1;
int x = *ptr2;
```

This program will be type checked successfully – but typecasting "pointer to float" into "pointer to int" at run time will produce a garbage value in **x**

Language Safety

Want more than static type safety – want language safety

- Cannot "break" the abstractions of the language (type-related and otherwise); e.g., no buffer overflows, segmentation faults, return address overriding, garbage values, etc.

Example: C is unsafe for many reasons, one of which is the lack of static type safety

- Other reasons: null pointers lead to segmentation faults (OS concept, not PL concept); buffer overflows lead to stack smashing or garbage values
  - Interesting follow up: **CSE 5474 (Software Security):** dedicated lecture and lab on **stack smashing and code injection**; Course scope: common software vulnerabilities, memory exploits, vulnerability analysis (e.g., reverse engineering, fuzzing, and symbolic execution), defenses against common vulnerabilities

Language Safety

Example: Java is safe – combination of static type safety & run-time checks

- Static type safety ensures that a well-typed program will not do type-related "bad" things
- Run-time checks catch things that cannot be caught statically via types: e.g., null pointers, array index out of bounds, division by zero

Example: Lisp is safe – dynamic checks for type-related correctness ("operands of PLUS must be numbers") and special "bad" values (e.g., "trying to get an element out of an empty list")