CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 17: Types and Type-Checking
25 Feb 08

What Are Types?

- Types describe the values possibly computed during execution of the program
- Types are predicate on values
  - E.g., “int x” in Java means “$x \in [-2^{31}, 2^{31})$”
  - Think: “type = set of possible values”
- Type errors: improper, type-inconsistent operations during program execution
- Type-safety: absence of type errors at run time

How to Ensure Type-Safety?

- Bind (assign) types, then check types
  - Type binding: defines types for constructs in the program (e.g., variables, functions)
    - Can be either explicit (int x) or implicit (x = 1)
    - Type consistency (safety) = correctness with respect to the type bindings
  - Type checking: static semantic checks to enforce the type safety of the program
    - Enforce a set of type-checking rules

Static vs. Dynamic Typing

- Static and dynamic typing refer to type definitions (i.e., bindings of types to variables, expressions, etc.)
  - Statically typed language: types are defined and checked at compile-time, and do not change during the execution of the program
    - E.g., C, Java, Pascal
  - Dynamically typed language: types defined and checked at run-time, during program execution
    - E.g., Lisp, Scheme, Smalltalk

Strong vs. Weak Typing

- Strong and weak typing refer to how much type consistency is enforced
  - Strongly typed languages: guarantees that accepted programs are type-safe
  - Weakly typed languages: allow programs that contain type errors
- Can achieve strong typing using either static or dynamic typing

Soundness

- Sound type systems: can statically ensure that the program is type-safe
- Soundness implies strong typing
- Static type safety requires a conservative approximation of the values that may occur during all possible executions
  - May reject type-safe programs
  - Need to be expressive: reject as few type-safe programs as possible

Concept Summary

- Static vs dynamic typing
  - when to define/check types?
- Strong vs weak typing
  - how many type errors?
- Sound type systems
  - statically catch all type errors (and possibly reject some programs that have no type errors)

Classification

	Strong Typing	Weak Typing
Static Typing	ML Pascal Java Modula-3	C C++
Dynamic Typing	Scheme PostScript Smalltalk	assembly code

Why Static Checking?

- Efficient code
  - Dynamic checks slow down the program
- Guarantees that all executions will be safe
  - With dynamic checking, you never know when the next execution of the program will fail due to a type error
- But is conservative for sound systems
  - Needs to be expressive: reject few type-safe programs

Type Systems

- Type is predicate on value
- Type expressions: describe the possible types in the program: int, string, array[], Object, etc.
- Type system: defines types for language constructs (e.g., expressions, statements)

Type Expressions

- Languages have basic types
  (a.k.a. primitive types or ground types)
  - E.g., int, char, boolean
- Build type expressions using basic types:
  - Type constructors
  - Type aliases

Array Types

- Various kinds of array types in different programming languages

- array(T) : array with elements of type T and no bounds
  - C, Java: int [ ], Modula-3: array of integer

- array(T, S) : array with size
  - C: int[10], Modula-3: array[10] of integer
  - May be indexed 0..size-1

- array(T,L,U) : array with upper/lower bounds
  - Pascal or Ada: array[2 .. 5] of integer

- array(T, S_1, ..., S_n) : multi-dimensional arrays
  - FORTRAN: real(3,5)

Record Types

- A record is $\{id_1: T_1, \ldots, id_n: T_n\}$ for some identifiers id_i and types T_i
- Supports access operations on each field, with corresponding type
- C: struct { int a; float b; }
- Pascal: record a: integer; b: real; end
- Objects: generalize the notion of records

Pointer Types

- Pointer types characterize values that are addresses of variables of other types
- Pointer(T) : pointer to an object of type T
- C pointers: T* (e.g., int *x;)
- Pascal pointers: ^T (e.g., x: ^integer;)
- Java: object references

Function Types

- Type: $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- Function value can be invoked with some argument expressions with types T_i, returns return type T_r

- C functions: int pow(int x, int y)
  type: $\text{int} \times \text{int} \rightarrow \text{int}$
- Java: methods have function types

- Some languages have first-class functions
  - usually in functional languages, e.g., ML, LISP
  - C and C++ have function pointers
  - Java doesn't

Type Aliases

- Some languages allow type aliases (type definitions, equates)
  - C: typedef int int_array[ ];
  - Modula-3: type int_array = array of int;
  - Java doesn't allow type aliases
- Aliases are not type constructors!
  - int_array is the same type as int [ ]
- Different type expressions may denote the same type

Implementation

- Use a separate class hierarchy for type ASTs:

```
class BaseType extends Type { ... }
class IntType extends BaseType { ... }
class BoolType extends Base Type { ... }
class ArrayType extends Type { Type elemType; }
class FunctionType extends Type { ... }
```

- Translate type expressions to type objects during parsing

```
non terminal Type type
type ::= BOOLEAN  {: RESULT = new BoolType(); :}
    |  ARRAY LBRACKET type:t RBRACKET {: RESULT = new ArrayType(t); :}
    ;
```

- Bind names to type objects in symbol table during subsequent AST traversal

Processing Type Declarations

- Type declarations add new identifiers and their types in the symbol tables
- Class definitions must be added to symbol table:

    class_defn ::= CLASS ID:id { decls:d }

- Forward references require multiple passes over AST to collect legal names

    class A { B b; }

    class B { ... }

Type Comparison

- Option 1: implement a method T1.Equals(T2)
  - Must compare type objects for T1 and T2
  - For object-oriented language: also need sub-typing: T1.SubtypeOf(T2)

- Option 2: use unique objects for each distinct type
  - each type expression (e.g., array[int]) resolved to same type object everywhere
  - Faster type comparison: can use ==
  - Object-oriented: check subtyping of type objects

Type-Checking

- Type-checking = verify typing rules
- Implement by an AST visitor

```
class typeCheck implements Visitor {
        Object visit(Add e, Object symbolTable)  {
            Type t1 = (int) e.e1.accept(this, symbolTable);
            Type t2 = (int) e.e2.accept(this, symbolTable);
            if (t1 == Int && t2 == Int) return Int;
            else throw new TypeCheckError("+");
            }
        Object visit(Num e, Object symbolTable)   {
            return Int;
            }
        Object visit(Id e, Object symbolTable)   {
            return (SymbolTable)symbolTable.lookupType(e);
            }
}
```

CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 17part2: Classes and Objects
22 Feb 08

Records

- Objects combine features of records and abstract data types
- Records = aggregate data structures
  - Combine several variables into a higher-level structure
  - Type is essentially Cartesian product of element types
  - Need selection operator to access fields
  - Pascal records, C structures
- Example: struct {int x; float f; char a,b,c; int y } A;
  - Type: {int x; float f; char a,b,c; int y }
  - Selection: A.x = 1; n = A.y;

ADTs

- Abstract Data Types (ADT): separate implementation from specification
  - Specification: provide an abstract type for data
  - Implementation: must match abstract type
- Example: linked list

implementation

```
Cell = { int data; Cell next; }
List = {int len; Cell head, tail; }

int length() { return l.len; }
int first() { return head.data; }
List rest() { return head.next; }
List append(int d) { ... }
```

specification

```
int length();
List append (int d);
int first();
List rest();
```

Objects as Records

- Objects have fields
- ... in addition, they have methods = procedures that manipulate the data (fields) in the object
- Hence, objects combine data and computation

```
class List {
    int len;
    Cell head, tail;

    int length();
    List append(int d);
    int first();
    List rest();
}
```

Objects as ADTs

- Specification: signatures of public methods and fields of object
- Implementation: Source code for a class defines the concrete type (implementation)

```
class List {
    private int len;
    private Cell head, tail;

    public static int length() {…};
    public static List append(int d) {…};
    public static int first() {…} ;
    public static List rest() {…};
}
```

Objects

- What objects are:
  - Aggregate structures that combine data (fields) with computation (methods)
  - Fields have public/private qualifiers (can model ADTs)
- Need special support in many compilation stages:
  - Type checking
  - Static analysis and optimizations
  - Implementation, run-time support
- Features:
  - inheritance, subclassing, polymorphism, subtyping, overriding, overloading, dynamic dispatch, abstract classes, interfaces, etc.

Inheritance

- Inheritance = mechanism that exposes common features of different objects
- Class B extends class A = "B has the features of A, plus some additional ones", i.e., B inherits the features of A
  - B is subclass of A; and A is superclass of B

```
class Point {
    float x, y;
    float getx(){ ... };
    float gety(){ ... };
}
class ColoredPoint extends Point {
    int color;
    int getcolor(){ ... };
}
```

Single vs. Multiple Inheritance

- Single inheritance: inherit from at most one other object (Java)
- Multiple inheritance: may inherit from multiple objects (C++)

```
class A {
    int a;
    int geta(){...};
};
```

```
class B {
    int b;
    int getb(){...};
};
```

```
class C : A, B {
    int c;
    int getc(){...};
};
```

Inheritance and Scopes

- How do objects access fields and methods of:
  - Their own?
  - Their superclasses?
  - Other unrelated objects?

- Each class declaration introduces a scope
  - Contains declared fields and methods
  - Scopes of methods are sub-scopes

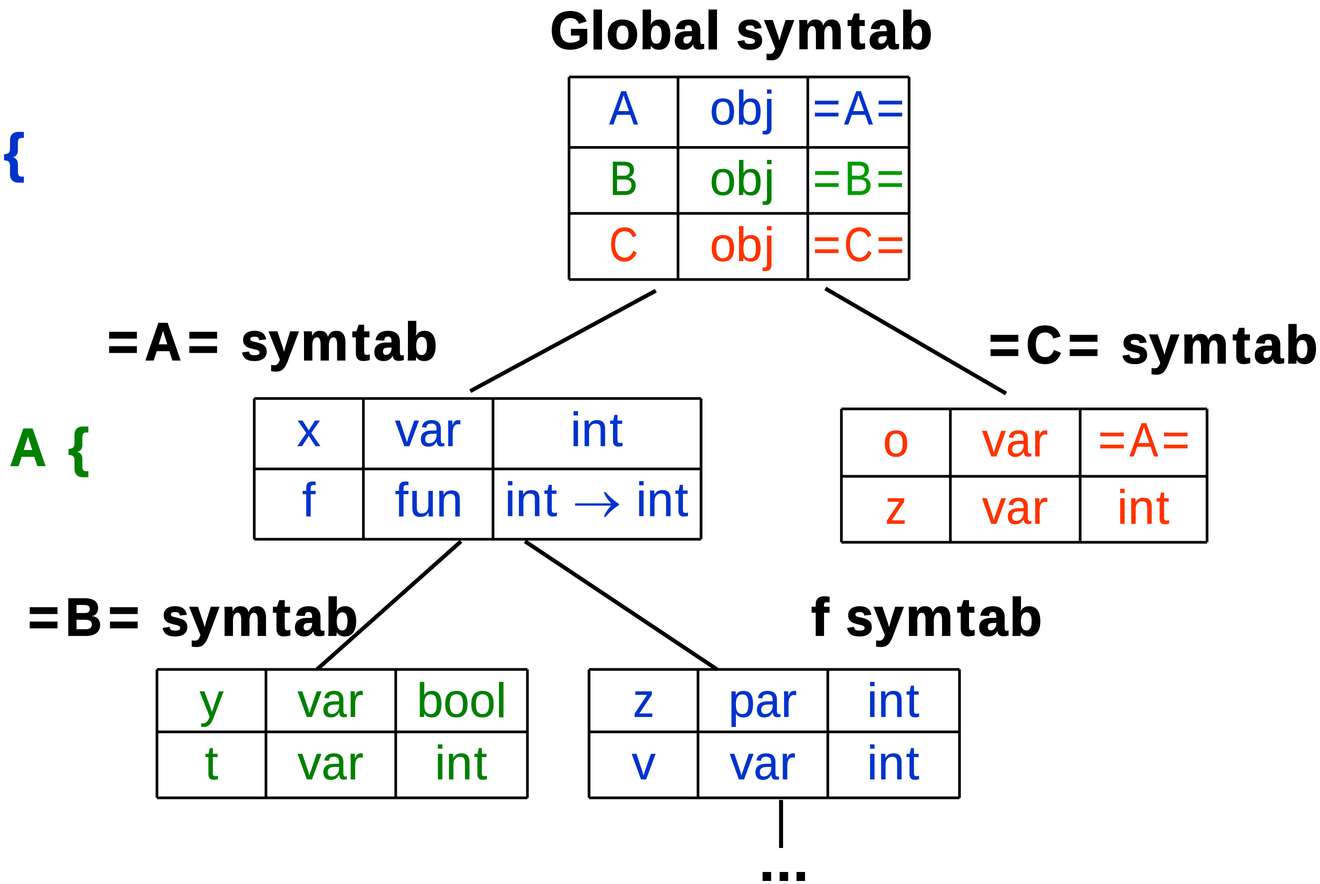
- Inheritance implies a hierarchy of class scopes
  - If B extends A, then scope of A is a parent scope for B

Example

```
class A {
    int x;
    int f(int z) {
        int v; ...
    }
}

class B extends A {
    bool y;
    int t;
}

class C {
    A o;
    int z;
}
```



Example

```
class A {
    int x;
    int f(int z) {
        int v; ...
    }
}

class B extends A {
    bool y;
    int t;
}

class C {
    A o;
    int z;
}
```

Global symtab

A	obj	=A=
B	obj	=B=
C	obj	=C=

=A= symtab

x	var	int
f	fun	int $\rightarrow$ int

=C= symtab

o	var	=A=
z	var	int

=B= symtab

y	var	bool
t	var	int

f symtab

z	par	int
v	var	int

...

Class Scopes

- Resolve an identifier occurrence in a method:
  - Look for symbols starting with the symbol table of the current block in that method

- Resolve qualified accesses:
  - Accesses o.f, where o is an object of class A
  - Walk the symbol table hierarchy starting with the symbol table of class A and look for identifier f
  - Special keyword this refers to the current object, start with the symbol table of the enclosing class

Class Scopes

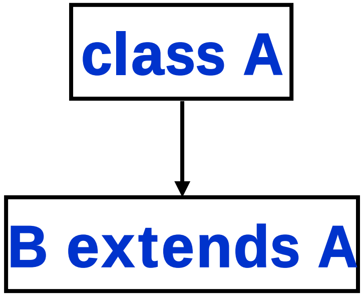
- Multiple inheritance:
  - A class scope has multiple parent scopes
  - Which should we search first?
  - Problem: may find symbol in both parent scopes!

- Overriding fields:
  - Fields defined in a class and in a subclass
  - Inner declaration shadows outer declaration
  - Symbol present in multiple scopes

Inheritance and Typing

- Classes have types
  - Type is Cartesian product of field and method types
  - Type name is the class name
- What is the relation between types of parent and inherited objects?

- Subtyping: if class B extends A then
  - Type B is a subtype of A
  - Type A is a supertype B

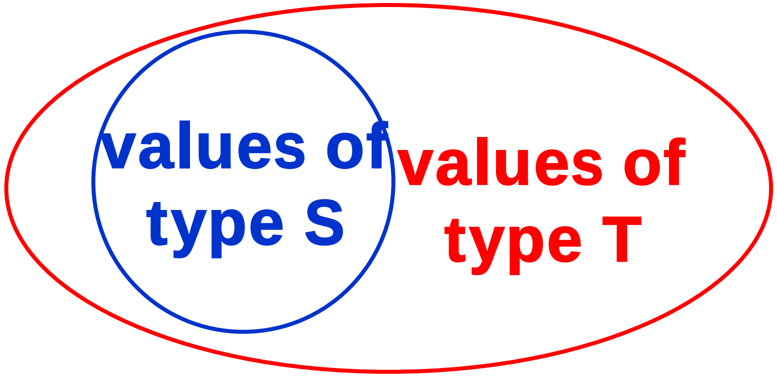


- Notation: B <: A

Subtype ≈ Subset

"A value of type S may be used wherever a value of type T is expected"

$$S <: T \quad \rightarrow \quad \text{values}(S) \subseteq \text{values}(T)$$



Subtype Properties

- If type S is a subtype of type T (S <: T), then:

  a value of type S may be used wherever a value of type T is expected (e.g., assignment to a variable, passed as argument, returned from method)

```
Point x;
ColoredPoint y;
x = y;
```

ColoredPoint <: Point

$\uparrow$ **subtype** $\uparrow$ **supertype**

- Polymorphism: a value is usable as several types
- Subtype polymorphism: code using T's can also use S's; S objects can be used as S's or T's.

Assignment Statements (Revisited)

$$\frac{A,\ id{:}T\ \vdash E : T}{A,\ id{:}T\ \vdash id = E : T}\ \textbf{(original)}$$

$$\frac{A,\ id{:}T\ \vdash E : S \quad \text{where } S <: T}{A,\ id{:}T\ \vdash id = E : T}\ \textbf{(with subtyping)}$$

How To Test the SubType Relation

```
class A {
    int x;
    int f(int z) {
        int v; ...
    }
}
```

```
class B extends A {
    bool y;
    int t;
}
```

```
class C {
    A a;
    B b;
    ...
    a = b;
}
```

Global symtab

A	obj	=A=
B	obj	=B=
C	obj	=C=

=A= symtab

x	var	int
f	fun	int $\rightarrow$ int

=C= symtab

a	var	=A=
b	var	=B=

=B= symtab

y	var	bool
t	var	int

f symtab

z	arg	int
v	var	int

...

Implications of Subtyping

- We don't statically know the types of object references
  - Can be the declared class or any subclass
  - Precise types of objects known only at run-time
- Problem: overriden fields / methods
  - Declared in multiple classes in hierarchy. Don't know statically which declaration to use at compile time
  - Java solution:
    - statically resolve fields using declared type of reference; no field overriding
    - dynamically resolve methods using the object's type (dynamic dispatch); in support of static type checking, a method m overrides m' only if the signatures are "nearly" identical --- the same number and types of parameters, and the return type of m a subtype of the return type of m'

Example

```
class A {
    int x;
  int f(int z) {
      int v; ...
    }
  }

class B extends A {
    bool y;
  int g(int z) {
      int w; ...
    }
  }

class C {
  A a = new B();
  B b = new B();
   ... a.x ...
   ... b.y ...
  }
```

Global symtab

A	obj	=A
B	obj	=B
C	obj	=C

=A= symtab

x	var	int
f	fun	int → int

C symtab

a	var	=A=
b	var	=B=

=B= symtab

y	var	bool
g	fun	int → int

f symtab

z	arg	int
v	var	int

...

g symtab

z	arg	int
w	var	int

Example

```
class A {
    int x;
  int f(int z) {
      int v; ...
    }
  }

class B extends A {
      bool x;
  int f(int z) {
      int w; ...
    }
  }

class C {
  A a = new B();
  B b = new B();
     ... a.x ...
     ... b.x ...
  }
```

Global symtab

A	obj	=A=
B	obj	=B=
C	obj	=C=

=A= symtab

x	var	int
f	fun	int → int

=C= symtab

a	var	=A=
b	var	=B=

=B= symtab

x	var	bool
f	fun	int → int

f symtab

z	arg	int
v	var	int

...

f symtab

z	arg	int
w	var	int

Example

```
class A {
    int x;
    int f(int z) {
        int v; ...
    }
}

class B extends A {
    bool x;
    int f(int z) {
        int w; ...
    }
}

class C {
    A a = new B();
    B b = new B();
    ... a.f(1) ...
    ... b.f(1) ...
}
```

Global symtab

A	obj	=A=
B	obj	=B=
C	obj	=C=

=A= symtab

x	var	int
f	fun	int → int

=C= symtab

a	var	=A=
b	var	=B=

=B= symtab

x	var	bool
f	fun	int → int

f symtab (of =A=)

z	arg	int
v	var	int

...

f symtab (of =B=)

z	arg	int
w	var	int

Objects and Typing

- Objects have types
  - ... but also have implementation code for methods
- ADT perspective:
  - Specification = typing
  - Implementation = method code, private fields
  - Objects mix specification with implementation
- Can we separate types from implementation?

Interfaces

- Interfaces are pure types; they don't give any implementation

implementation

```
class MyList implements List
{
  private int len;
  private Cell head, tail;

  public int length() {…};
  public List append(int d)
  {…};
  public int first() {…} ;
  public List rest() {…};
}
```

specification

```
interface List {
  int length();
  List append(int
  d);
  int first();
  List rest();
}
```

Multiple Implementations

- **Interfaces allow multiple implementations**

```
interface List {
    int length();
    List append(int);
    int first();
    List rest(); }
```

```
class SimpleList implements List {
    private int data;
    private SimpleList next;
    public int length()
    { return 1+next.length() } ...
```

```
class LenList implements List {
    private int len;
    private Cell head, tail;
    private LenList() {...}
    public List append(int d) {...}
    public int length() { return len; }
    ...
```

Implementations of Multiple Interfaces

```
interface A {
        int foo();
        }
interface B {
        int bar();
        }
class AB implements A, B {
        int foo(){ ... }
        int bar(){ ... }
        }
```

Subtyping vs. Subclassing

- Can use inheritance for interfaces
  - Build a hierarchy of interfaces

  interface A {...}

  interface B **extends** A {...}

  $\boxed{\mathbf{B <: A}}$

- Objects can implement interfaces

  class C **implements** A {...}

  $\boxed{\mathbf{C <: A}}$

- Subtyping: interface inheritance
- Subclassing: object (class) inheritance
  - Subclassing implies subtyping

Abstract Classes

- Classes define types and some values (methods)
- Interfaces are pure object types

- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class

Subtypes in Java

interface I_1 **extends** I_2 { … }

class C implements I { … }

class C_1 **extends** C_2

$$\begin{array}{c} I_2 \\ | \\ I_1 \end{array} \qquad \begin{array}{c} I \\ | \\ C \end{array} \qquad \begin{array}{c} C_2 \\ | \\ C_1 \end{array}$$

$I_1 <: I_2$

$C <: I$

$C_1 <: C_2$

Subtype Hierarchy

- Introduction of subtype relation creates a hierarchy of types: subtype hierarchy