CS412/413

Introduction to Compilers
Radu Rugina

Lecture 30: Objects
10 Apr 02

ADTs

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

```java
class List {
    private int len;
    private Cell head, tail;
    int length() { return len; }
    int first() { return head.data; }
    List rest() { return head.next; }
    List append(int d) { ... }
}
```

Records

- Objects combine features of records and abstract data types
- Records = aggregate data structures
  - Combine several variables (called fields) 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) A
  - Selection: A.x = 1; n = A.y;

Objects as Records

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

```java
class List {
    int len;
    Cell head, tail;
    int length();
    List append(int d);
    int first();
    List rest();
}
```

Objects as ADTs

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

```java
class List {
    private int len;
    private Cell head, tail;
    public static int length() { ... }
    public static List append(int d) { ... }
    public static List first() { ... }
    public static List rest() { ... }
}
```

Objects

- What objects are:
  - Aggregate structures which combine data (fields) with computation (methods)
  - Fields have public/private qualifiers (can model ADTs)
  - Also referred to as classes
- Objects interfere with almost all compilation stages:
  - Lexical and syntax analysis
  - Semantic analysis (type checking)
  - Analysis and optimizations
  - Implementation, run-time support
- Features:
  - Inheritance, subclasseing, subtyping, dynamic dispatch
Inheritance

- **Inheritance**: mechanism which exposes common features of different objects
- Object O1 inherits from O2 = “O1 has the features of O2, plus some additional ones”
  - Say that O2 extends O1

  ```java
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 only one other object
- **Multiple inheritance**: inherit from multiple objects

```java
class A {
  int a;
  int getInt();
}

class B {
  int b;
  int getInt();
}

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

Inheritance and Typing

- Inheritance defines a hierarchy between objects
- Objects have types:
  - Type is cartesian product of field and method types
- What is the relation between types of parent and inherited objects?

  - **Subtyping**: if O2 extends O1 then
    - Type(O2) is a **subtype** of Type(O1)
    - Type(O1) is a **supertype** of Type(O2)

  - Notation: Type(O2) \( \subseteq \) Type(O1)

Subtype = Subset

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

\[ S \subseteq T \rightarrow \text{values}(S) \subseteq \text{values}(T) \]

Subtype Properties

- If type S is a subtype of type T (\( S \subseteq T \)), 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)

```java
Point x;
ColoredPoint y;
ColoredPoint s : Point
x = y;
```

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

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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>class MyList implements List {</td>
<td>interface List {</td>
</tr>
<tr>
<td>private int len;</td>
<td>int length();</td>
</tr>
<tr>
<td>private Cell head, tail;</td>
<td></td>
</tr>
<tr>
<td>public int length(...);</td>
<td>List append(int);</td>
</tr>
<tr>
<td>public int first(...);</td>
<td>int first();</td>
</tr>
</tbody>
</table>
| public int rest(...); | 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 length();
  }
}

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

Subtyping vs. Subclassing

- Can use inheritance for interfaces
  - Build a hierarchy of interfaces
    interface A (...) interface B extends A (...) B <: A
  - Objects can implement interfaces
class C implements A (...) C <: A

- Subtyping: interface inheritance
- Subclassing: object (class) inheritance
  - Subtyping 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 I1 extends I2 (...) class C implements I (...) extends C2
I2 I C C2 I1 C1 C2 I1 <: I2 C <: I C1 <: C2

Subtyping Properties

- Subtype relation is reflexive: T <: T
- Transitive: R <:: S and S <: T implies R <: T
- Anti-symmetric:
  T1 <:: T2 and T2 <:: T1 => T1 = T2
- Defines a partial ordering on types!
- Use diagrams to describe typing relations
Subtype Hierarchy

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

![Subtype Hierarchy Diagram]

Type-checking

- **Problem:** what are the valid types for an object?
- **Subsumption rule** connects subtyping relation and ordinary typing judgements:

  \[
  A \vdash E : S \\
  S <: T \Rightarrow \text{values}(S) \subseteq \text{values}(T)
  \]

- "If expression \( E \) has type \( S \), it also has type \( T \) for every \( T \) such that \( S <: T \)."

Implementing Type-checking

- **Next problem:** static semantics is supposed to find a type for every expression, but objects may have (in general) many types

- Which type to pick?

  ![Implementing Type-checking Diagram]

Principal Type

- **Idea:** every expression has a principal type that is the most-specific type of the expression

- Can use subsumption rule to infer all supertypes if principal type is used

  ![Principal Type Diagram]

Type-checking Overview

- Rules for checking code must allow a subtype where a supertype was expected
- Old rule for assignment:

  \[
  \frac{
  \text{id} : T = A \\
  A \vdash E : T \\
  }{
  A \vdash \text{id} = E : T
  }
  \]

  What needs to change here?

- New rule for assignment:

  \[
  \frac{
  A \vdash E : T_p \\
  T_p <: T \\
  \text{id} : T \in A \\
  }{
  A \vdash \text{id} = E : T
  }
  \]

  \[
  \frac{
  A \vdash E : S \\
  S <: T \\
  }{
  A \vdash E : T
  }
  \]

  \[
  \frac{
  A \vdash E : T \\
  }{
  A \vdash \text{id} = E : T
  }
  \]
Type-checking Code

```java
class Assignment extends ASTNode {
    Variable var; ExprNode E;
    Type typeCheck() {
        Type Tp = E.typeCheck();
        Type T = var.getType();
        if (Tp.subtypeOf(T)) return T;
        else throw new TypeError(E);
    }
}
```

The Dispatching Problem

- Problem: don't know what code to run at compile time!
  ```java
  List a;
  if (cond) { a = new SimpleList(); }
  else { a = new LenList(); }
  a.length()
  ```
  or SimpleList.length() or LenList.length()?
- Objects must “know” their implementation at run time
- Method invocations must be resolved dynamically
- Dynamic dispatch: run-time mechanism to select the appropriate method, depending on the object type

Implementing Dynamic Dispatch

- Objects implemented by adding extra pointer to dispatch vector (also: virtual table) with pointers to method code
- Code receiving List only knows x has initial dispatch vector pointer
  ```java
  SimpleList dispatch vector
  SimpleListLength
  SimpleListFirst
  SimpleListRest
  ```
- LenList dispatch vector
  ```java
  LenListLength
  LenListFirst
  LenListRest
  ```

- Code receiving List only knows x has initial dispatch vector pointer