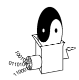


CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 20: Objects
14 Mar 01

Records

- Last time: modules approximated by records

  type: `{x:int, s: String, c,d,e: char, y: int }`

  terms:

  construction:

  `{x = 2, s = "hi", c = 'x', … y = 10 }`

  selection:

  `r.f`

Modules + abstract types

- Module is no longer a record: interface also contains list of abstract types
- Type:
  $\{\text{type } I_1 \ldots \text{type } I_n;\ v_1 : T_1 \ldots v_m : T_m\}$
- Stripped-down term syntax:
  $\text{module } \{ \text{ type } I_1 = T'_1, \ldots, I_n = T'_n \quad v_1 : T_1 = e_1 \ldots v_{m'} : T_{m'} = e_{m'} \}$

How to type-check?

- module must agree with own interface (everything implemented, with right type)
- Inside implementation, concrete types known: substitute (or put in symbol table)
- You already do a lot of this!

substitution

$$\frac{A, v_j : T_j \ ^{(j \in 1..m')} \vdash e_k\{T'_i / I_i \ ^{(i \in 1..n)}\} : T_k \quad ^{(k \in 1..m')} \qquad m' \geq m}{A \vdash \begin{array}{l}\text{type } I_1 = T'_1, \ldots, I_n = T'_n \\ v_1 : T_1 = e_1 \ldots v_{m'} : T_{m'} = e_{m'}\end{array} : \{ \begin{array}{l}\text{type } I_1 \ldots \text{type } I_n; \\ v_1 : T_1, \ldots, v_m : T_m\end{array} \}}$$

Multiple Implementations

- Non-OO languages: only one implementation of (module value for) any interface
- Linker ensures single implementation
- Doesn't scale to large systems—want multiple implementations of an interface
- Approach 1: *objects*
- Approach 2: *first-class module values* using *dependent module types* (*e.g.*, FX-91 language)

Using Objects as ADTs

- Another way to extend records into ADTs
- Source code for a class defines the concrete type (implementation)
- Interface defined by `public` variables and methods of class

```
class List {
   public static int length(List l);
   public static List cons(int, List);
   public static int first(List);
   public static List rest(List);
   private int len, head;
   private List next;
}
```

```
type T;
length(T): int
cons(int,T): T,
first(T): int
rest(T): T
```

Multiple implementations

- Can model using classes and methods:

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

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

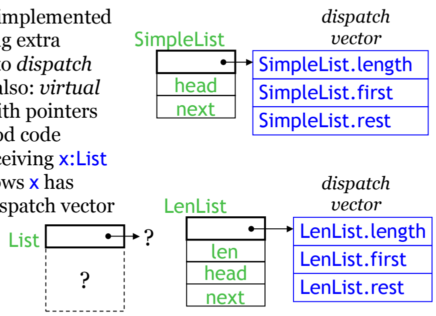
```
class LenList implements List {
   private int len, head;
   private LenList next;
   private LenList(int h,t) {...}
   public int length() { return len; }
   public List cons(int h)
       { return new LenList(h, this); } ...
```


The dispatching problem

- Problem: don't know what code to run at compile time.
  List a; a.length()
  ⇒ SimpleList.length or LenList.length?
- Objects must "know" their implementation at run time

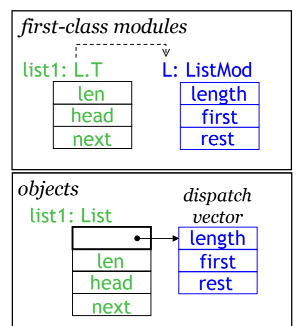
Compiling objects

- Objects implemented by adding extra pointer to *dispatch vector* (also: *virtual table*) with pointers to method code
- Code receiving x:List only knows x has initial dispatch vector pointer



Modules vs. objects

- Objects fold together functionality of records, abstract types and modules
- Both mechanisms allow forms of *polymorphism*: code can use values of more than one type
- Mechanisms have subtly different expressive power



Binary operations

- Advantage of abstract types: compare "LenList" in both styles, but with a binary "prepend" operation:

```
LenList: ListMod = {
   type T = {len: int, head:int, next: T}
   length(l: T): int = l.len
   cons(h: int, l: T): T = {len = l.len+1, ... }
   prepend(l1, l2: T): T = (if (l1.len == 0) l2
       else cons(l1.head, prepend(l1.next, l2)))
}
class LenList implements List {
   len, head: int, next: List
   length() = len
   prepend(l1: List) = ( if (l1.length() == 0) this else
       cons(l1.first(), prepend(l1.rest()))
```

Can't access l1 fields directly!

Heterogeneity

- Objects are better for *heterogenous* data structures containing different implementations of same interface
- Can mix different List impls in same list

- Abstract types are better for *homogeneous* data structures where we want to exploit same-type property

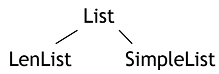
Type relationships

- Relationship of LenList module and List interface is relationship of a *value* to its *type*

  LenList, SimpleList : ListMod

- Relationship of classes and object interfaces is more complex... types related by *subtype* relationship
- Enables heterogeneous data structures

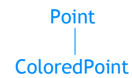
LenList <: List
SimpleList <: List



Subtypes

- Idea: one type can *extend* another by allowing more operations

```
interface Point {
   float x();
   float y();
}
interface ColoredPoint
      extends Point {
   float x();
   float y();
   Color color();
}
```



is a subtype of
ColoredPoint <: Point
(also: ≤)

Subtype properties

If type S is a subtype of type T ($S <: T$)

- 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
subtype *supertype*

- *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.

Subtypes in Java

interface I extends I_2 { ... }
class C implements I { ... }
class C extends C_2

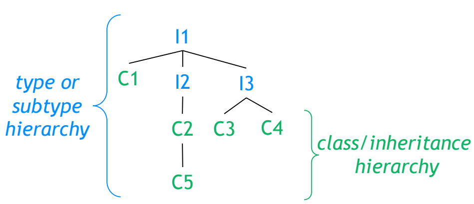
$I_1 <: I_2$

$C <: I$

$C_1 <: C_2$
C_1 inh C_2

Subtype hierarchy

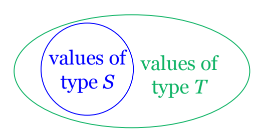
- Introduction of subtype relation creates a hierarchy of types: *subtype hierarchy*



Subtype ≈ Subset

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

$S <: T \rightarrow$
values(S) $\subseteq$ values(T)



Subtyping axioms

- Subtype relation is reflexive: $T <: T$
- Transitive: $$\frac{R <: S \quad S <: T}{R <: T}$$
- Usually anti-symmetric: $$T_1 <: T_2 \wedge T_2 <: T_1 \Rightarrow T_1 = T_2$$
- Defines an ordering on types (partial order)
- Language defines subtype judgement on various type kinds (primitives, records, &c)
- Java: C <: Object, C <: I

Subsumption

- *Subsumption rule* connects subtyping relation and ordinary typing judgements

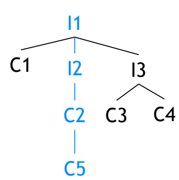
$$\frac{A \vdash E : S \quad S <: T}{A \vdash E : T}$$

$S <: T \rightarrow$ values(S) $\subseteq$ values(T)

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

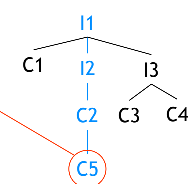
Implementing Type-checking

- Problem: static semantics is supposed to find a type for every expression, but expressions have (in general) many types
- Which type to pick?



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

Type-checking interface

- Old method for checking types:

```
abstract class Node {
    abstract Type typeCheck(SymTab A);
      // Return the principal type of this
      // statement or expression
}
```

- No changes in interface needed to support subtyping (except interpretation of result of typeCheck)

Type-checking rules

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

$$\frac{id : T \in A \quad A \vdash E : T}{A \vdash id = E \ : T}$$

What needs to change here?

Type-checking code

```
class Assignment extends ASTNode {
  String id; Expr E;
  Type typeCheck(SymTab A) {
      Type Tp = E.typeCheck(A);
      Type T = A.lookupVariable(id);
      if (Tp.subtypeOf(T)) return T;
      else throw new TypecheckError(E); }}
```

$$\frac{A \vdash E : T_p \quad T_p <: T \quad id : T \in A}{A \vdash E : T} \;=\; \frac{A \vdash E : S \quad S <: T}{A \vdash E : T} \;+\; \frac{id : T \in A \quad A \vdash E : T}{A \vdash id = E \; : T}$$

Combining rules

- Consider most general use of rules in typing derivation:

$$\dfrac{id : T \in A \qquad \dfrac{\dfrac{\dots}{A \vdash E : T_p} \quad \dfrac{\dots}{T_p <: T}}{A \vdash E : T}}{A \vdash id = E \; : T}$$

$$\dots$$

$$\dfrac{id : T \in A \quad \dfrac{\dots}{A \vdash E : T_p} \quad \dfrac{\dots}{T_p <: T}}{A \vdash id = E \; : T}$$

$$\dots$$

Unification

- Some rules more problematic: if
- Rule:

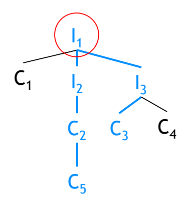
$$\frac{A \vdash E : \textbf{bool} \quad A \vdash S_1 : T \quad A \vdash S_2 : T}{A \vdash \text{if } (E)\ S_1 \text{ else } S_2 : T}$$

- Problem: suppose S_1 has principal type T_1, S_2 has principal type T_2. Old check: $T_1 = T_2$. New check: need principal type T. How to unify T_1 , T_2 ?

Unification in hierarchy

- Idea: unified principal type is least common ancestor in type hierarchy

$LCA(C_3, C_5) = I_1$



Logic: I_1 must be same as or subtype of any type that could be the type of both a value of type C_3 and a value of type C_5