Module Mechanisms

- Last time: modules, ways to implement ADTs
- **Module**—collection of related values and types; mechanism for separate compilation, encapsulation, abstraction
- **Record**—set of named fields with types; modules similar to records; module interface defines type of module value
- **Abstract type**—allows encapsulation of values generated by module
- Implementation known only at link time -- clients are insulated from changes, but harder to optimize

Module Mechanisms

- Last time: modules, ways to implement ADTs
- **Module**—collection of related values and types; mechanism for separate compilation, encapsulation, abstraction
- **Record**—set of named fields with types; modules similar to records; module interface defines type of module value
- **Abstract type**—allows encapsulation of values generated by module
- Implementation known only at link time -- clients are insulated from changes, but harder to optimize

Abstract types

Iota+abstract types, Modula-3 style:

```plaintext
list.int: declaration of abstract type
type List;
length(l: List): int
cons(h: int, l: List): List
first(l: List): int
rest(l: List): List

list.mod:
type List = {len: int, head: int, next: List}
length(l: List): int = l.len
cons(h: int, l: List): List = List{len=l.len+1,head=h,l=l}
...```

How to type-check?

- Additional issues:
  - module must agree with own interface (everything implemented, with right type)
  - must recognize abstract types correctly: add to symbol table
  - You should already do most of this!

```
A + {I_i: type = T_i'} (i/G32 1..n) + {v_j: T_j} (j/G32 1..m') /G60
m' >= m

A \vdash\Box\text{module}(I_i...I_n) \{ v_j: T_j...v_m: T_m \} = type I_i... type I_n
v_j: T_j...v_m: T_m
```

Multiple Implementations

- Most (non-OO) languages: only one implementation of (module value for) any interface
- Doesn’t scale to large programs—want multiple modules implementing an interface
- Approach 1: **first-class module values** using dependent types (e.g. FX-91 language)
- Approach 2: **objects**
First-class module values

- List interface: `ListMod` = 
  
  ```
  type T; 
  length(T): int 
  cons(int,T): T, 
  first(T): int 
  rest(T): T
  ```

- Two implementations:
  
  **SimpleList**: `ListMod = {` 
  
  ```
  type T = {head: int, next: T}, 
  length(l: T): int = (/\* recurse \*/), ... }
  ```

  **LenList**: `ListMod = {` 
  
  ```
  type T = {len: int, head:int, next: T}, 
  length(l: T): int = l.len, ... }
  ```

Ambiguity

- Problem: from interface, don’t know which implementation we are dealing with.

```
uses List = ListMod.T 
list1: List

L: ListMod, list1: L.T 
L.length(list1) – calls what?
```}

Implications

- Must name module value explicitly rather than using name of interface: `SimpleList.length`, `LenList.T` instead of `ListMod.length`, `ListMod.T`

- Code written to use ADT must be passed module value too!

```
sum(list: ListMod, a: list.T): int = 
if (list.length(a) == 0) 0 
else list.first(a) + sum(list, list.rest(a))
```

vs.

```
sum(a: ListMod.T): int = 
if (ListMod.length(a) == 0) 0 
else ListMod.first(a) + sum(a,...)
```}

Using Module Values

- First-class module value is record: points to proper code and global variables of module
- For single implementation (2nd-class modules), linker makes module calls direct

```L: ListMod, list1: L.T;  L.length(list1)
```}

Compiling Multiple Impls

- Can’t stack allocate -- need to know the concrete type of a reference (as in C++)
- Don’t know what code to run when an operation (e.g. `length`) is invoked

```L: ListMod, list1: L.T 
L.length(list1) - calls what?
```}

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

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

The dispatching problem

- Same problem as with first-class modules: don’t know what code to run at compile time.

    ```
    List a; a.length()
    ListMod L; ListMod.T a; L.length(a) 
    ```

- Difference: objects “know” their implementation without separate module value (no L needed)

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

  ```
  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(), this)))
  }
  ```

  ```
  class SimpleList implements List {
      head: int, next: SimpleList
      length() = 1 + next.length()
  }
  ```

  ```
  dispatch vector
  ```

  ```
  len
  head
  next
  ```

  ```
  dispatch vector
  ```

  ```
  len
  head
  next
  ```

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:

    ```
    type T = [len: int, head: int, next: T]
    length(l: T) = l.len
    cons(h: int, l: T) = [len = l, len + 1]
    prepend(l1, l2: T)
    ```

    ```
    prepend(l1: List) = ( if (l1.length() == 0) this else 
        cons(l1.first(), prepend(l1.rest(), this)))
    ```

    ```
    Can’t access T fields directly!
    ```

Heterogeneity

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

![Heterogeneity Diagram]
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

Subtypes

- Idea: one interface can extend another by adding more operations

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

```
ColoredPoint <: Point
```

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;
  ```
  - Subtype polymorphism: code using T’s can also use S’s.

Subtypes in Java

```
interface I extends I2 { … }
class C implements I { … }
class C extends C2
```

```
I2
I1
I
C
C2
C1
C inh C2
```

Subtype hierarchy

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

```
<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
<th>I3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Subtype ≈ Subset

- “A value of type S may be used wherever a value of type T is expected”
- S <: T → values(S) ⊆ values(T)
Subtyping axioms

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

Subsumption

- Subsumption rule connects subtyping relation and ordinary typing judgements
  \[
  A \vdash E : S \\
  S <: T \\
  \therefore A \vdash E : T
  \]
  \( \text{values}(S) \subseteq \text{values}(T) \)
  
  "If expression \( E \) has type \( S \), it also has type \( T \) for every \( T \) such that \( S <: T \)"

Implementing Type-checking

- 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 \textit{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:

  ```java
  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 code

```java
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);
    }
}
```

Unification

- Some rules more problematic: if
- Rule:
  \[
  A \vdash E : \text{bool}
  A \vdash S_1 : T
  A \vdash S_2 : T
  \]
  \[
  A \vdash \text{if } (E) S_1 \text{ else } S_2 : T
  \]
  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 subtype hierarchy

- Idea: unified principal type is least common ancestor in type hierarchy
  \[
  
  \]
  Logic: \( J \) 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 \( C_5 \)

Explicit vs Structural subtypes

- Java: all subtypes explicitly declared, name equivalence for types. Subtype relationships inferred by transitive extension.
- Languages with structural equivalence (e.g., Modula-3): subtypes inferred based on structure of types; no extends declaration
- Same checking done in each case; explicitly declared subtypes must follow rules for recognizing subtypes implicitly

Testing subtype relation

- Subtyping for records
  \( S \leq T \) means \( S \) has at least the fields of \( T \)
  \[
  \{ x: \text{int}, y: \text{int}, c: \text{Color} \} \leq \{ x: \text{int}, y: \text{int} \}
  \]
  - Implementation:
    \[
    \begin{array}{c}
    x \\
    y \\
    c \\
    \end{array}
    \leq
    \begin{array}{c}
    x \\
    y \\
    \end{array}
    \]

Subtype rule for records

\[
\{ x: \text{int}, y: \text{int}, c: \text{Color} \} \leq \{ x: \text{int}, y: \text{int} \}
\]

- \( m \leq n \)
  \[
  A \vdash \{ a_1: T_1, \ldots, a_m: T_m \} \leq \{ a_1: T_1, \ldots, a_n: T_n \}
  \]
  - Similar to our rule for checking modules
  - What about allowing field types to vary?
  - If \( \text{Point} \leq \text{ColoredPoint} \), allow
    \[
    \begin{array}{c}
    p \\
    z \\
    \end{array}
    \leq
    \begin{array}{c}
    p \\
    z \\
    \end{array}
    \]

Field Invariance

Try \{ p: ColoredPoint \} <: \{ p: Point \}

\begin{align*}
  x &: \{ p: Point \} \\
  y &: \{ p: ColoredPoint \} \\
  x &= y; \\
  x.p &= \text{new 3DPoint( );}
\end{align*}

- Mutable (assignable) fields must be \textit{invariant} under subtyping

Covariance

- Immutable record fields \textit{may} change with subtyping (may be \textit{covariant})
- Suppose we allow variables to be declared \textit{final} -- \quad x : \text{final int}
- Safe:
  \{ p: \text{final ColoredPoint}, x : \text{int} \} \leq \{ p: \text{final Point}, x : \text{int} \}

Summary

- Multiple implementation of abstract types special case of subtyping
- Subtyping characterized by new judgement: \( S <: T \)
- Old judgement \( A \vdash e : T \) plus subsumption rule, defn. of subtype relation defines new type-checking process
- Mutable fields must be invariant in subtype relation; immutable fields may be covariant