FEATHERWEIGHT JAVA

Classes \( C ::= \text{class } c \text{ extends } c \{ e_1; k; d \} \)

Constructors \( k ::= c \{ \text{super}(x); \text{this.}f = x; \} \)

Methods \( d ::= c \text{ m}(e) \{ \text{return } e; \} \)

Types \( \tau ::= c \)

Expressions \( e ::= x | e.f | e.m(e) | \text{new } c(e) | (c) e \)

Notation:

\( f \): field name

\( c \): class name

\( m \): method name

\( x \): variable name

\( \{ \) is an abbreviation for \( e_1, e_2, \ldots, e_k \)

\( \ldots \) is an abbreviation for \( c_1, x_1; \ldots \)

Class expressions:

\( \text{class } c \text{ extends } c' \{ c_1 f_1; c_2 f_2; \ldots; \} \)

\( d \)

\( \ldots \}

declares class \n
\( c \) to be a subclass of \( c' \)

with additional fields \( c_i, f_i \)

a single constructor \( k \)

methods \( \ldots \)

Constructor expressions:

\( c(c_1, c_2, \ldots, c_1 x_1, c_2 x_2, \ldots)\{ \text{super}(x); \text{this.f} = x; \} \)

\( \text{this.f} = x_1; \text{this.f} = x_2; \}

declares a constructor for class \( c \)

with arguments \( c_i, x_i \) corresponding to fields of the superclass

and arguments \( c_i, x_i \) corresponding to new fields of the subclass

\( x_i \) are initialised via the superclass

\( \text{this.f} = x; \) fields initialised in the subclass

Method expressions:

\( c \text{ m}(c_1 x_1, c_2 x_2, \ldots) \{ \text{return } e; \} \)

declares a method \( m \)

which returns a value of class \( c \)

with arguments \( x_i \) of class \( c_i \)

and a body returning the value of expression \( e \)
Field selection:

\[ e.f \]

select field \( f \) from instance \( e \)

Method invocation:

\[ e.m(e_1, e_2, \ldots) \]

invoke method \( m \) of instance \( e \) with arguments \( e_1, e_2 \)

Instance creation:

\[ \text{new } c(e_1, e_2, \ldots) \]

creates a new instance of class \( c \) with arguments \( e_1, e_2 \)

Casting:

\[ (c) e \]

casts value \( e \) to class \( c \)

Slide 5

Slide 7

the set of types is limited to the set of class names

• in examples, we may assume the presence of the types like int, but we will not discuss the semantics of operations on these types

there is a class Object

• all other classes are subclasses of Object

a special variable \textit{this} referring to the instance itself

Subclasses and Subtypes

the subclass/superclass relationship is similar to the subtype/supertype relationship with coercion interpretation

if \( c \) is a subclass of \( c' \), the \( c \) has at least as many fields as \( c' \)

objects of class \( c \) can be coerced (cast) to \( c' \) by deleting the additional fields

we therefore write \( c <: c' \)

Slide 6

Slide 8

Subclasses and Subtypes
class Pt extends Object {
    int x;
    int y;
    Pt (int x, int y) {
        super (); this.x = x; this.y = y;
    }
}
class CPt extends Pt {
    color c;
    CPt (int x, it y, color c) {
        super (x,y); this.c = c;
    }
    color getc () {return this.c;}
}

### Static Semantics

The static semantics is defined by the following judgements:

- $\tau < : \tau'$: subtyping
- $\Gamma \vdash e : \tau$: expression typing
- $d \text{ ok in } c$: class ok
- $C \text{ ok}$: well formed class
- $T \text{ ok}$: well formed class table
- $\text{fields}(c) = \{ c_1 f_1, c_2 f_2 \ldots \}$: field lookup
- $type(m, c) = \leq \rightarrow c$: method type

A program consists of a class table $T$ (sequence of class declarations) and an expression $e$.

We use the class table as implicit parameter in the following.

We only look at some interesting aspects of the static semantics.

Subclass relationship:

$$\tau < : \tau \quad \Rightarrow \quad \frac{\tau_1 < : \tau_2 \quad \tau_2 < : \tau_3}{\tau_1 < : \tau_3}$$

$T(c) = \text{class } c \text{ extends } c'(\ldots) \quad \Rightarrow \quad c < : c'$

Every variable has to be declared

$$x : \tau \in \Gamma \quad \Rightarrow \quad \frac{}{\Gamma \vdash x : \tau}$$

Types of fields are defined in the class table:

$$\Gamma \vdash e_0 : \alpha_0 \quad \text{ fields}(\alpha_0) = c_1 f_1 \ldots \quad \Rightarrow \quad \frac{}{\Gamma \vdash e_0.f_1 : c_1}$$
Argument and result types of methods are defined in the class table:

- methods of superclass can be applied, without explicit cast

\[ \Gamma \vdash e_0 : c_0 \quad \Gamma \vdash e : c \quad \text{type}(m, c_0) = c' \rightarrow c_1 \quad e < ; c' \]

All casts are statically valid:

\[ \Gamma \vdash e_0 : c' \]

\[ \Gamma \vdash (e) e_0 : c \]

Method overriding:

New subclass method must have
- the same argument types
- the same result type as superclass method

\[ T(c) = \text{class } c \text{ extends } c' \{ \ldots \} \quad \text{type}(m, c') = c \rightarrow c_0 \quad e : c \quad \text{this : } c \vdash e_0 : c_0 \]

\[ c_0 \quad m \quad (e) \quad (\text{return } e_0) \quad \text{ok in } c \]

Properties:

- casts may fail at run time; checks required
- method invocation is statically checked
- field selection is statically checked
**Featherweight Java: Dynamic Semantics**

We discuss the single step or structural operational semantics of Featherweight Java.

**Values:** an instance is a value if all of its arguments are values

\[
\text{value} \quad \text{new } c \{ v \} \quad \text{value}
\]

In essence, a instance is just a collection of named fields, labeled with the class name.

**Field Selection:**
- \( c', f' \): fields of the superclass
- \( c, f \): new fields of the class
- retrieve value of field from either superclass or class itself

\[
\text{fields}(c) = c', f' \quad \text{new } c \{ v, \ell \} \quad f_i \mapsto v_i'
\]

\[
\text{fields}(c) = c', f' \quad \text{new } c \{ v, \ell \} \quad f_i \mapsto v_i
\]

Note that in Featherweight Java, the content of fields cannot be changed after initialization.

We could extend the language using techniques described previously.

**Method Invocation:**
- method invocation relies on an auxiliary predicate, body, which provides the list of formal arguments and the body of a method
- replace all formal parameters \( z \) by actual parameters \( z' \)
- replace every occurrence of this by the instance itself

\[
\text{body}(m, c) = z \mapsto e_0
\]

\[
\text{new } c \{ z' \} \mapsto \{ z' / z \} \{ \text{new } c \{ z \} / \text{this} \} e_0
\]

**Type Cast:**
- if \( c' \) is supertype of \( c \), then cast instance by “ignoring” fields that are not present in \( c' \)
- otherwise, cause a checked run-time error

\[
\text{if } c \prec c' \quad \text{then } \text{new } c \{ z \} \mapsto \text{new } c \{ z \}
\]

\[
\text{if } c \not\prec c' \quad \text{then } \text{error}
\]

Note the interplay with the static semantic rules for type cast.
Dynamic Dispatch:
To find the body \( \text{body}(m, c) \) of a method \( m \) of a class \( c \), we have to search upwards through the class hierarchy for the first definition of \( m \).

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots \} \\
\text{body}(m, c) = x \rightarrow e
\]

\[
T(c) = \text{class } c \text{ extends } c' \{ \ldots \} \\
\text{body}(m, c) = x \rightarrow e
\]

And finally, we need rules to determine the evaluation order.

Subtyping Revisited
For records (or n-tuples) two types of subtyping:

- **Width Subtyping**
  \[
  \{ l_1 : \tau_1, \ldots, l_m : \tau_m \} <: \{ l_1 : \sigma_1, \ldots, l_n : \sigma_n \}
  \]
  Note: ordering of fields in records immaterial, rule allows us to “drop” arbitrary subset of fields

- **Depth Subtyping**
  \[
  \sigma_1 <: \tau_1, \sigma_2 <: \tau_2, \ldots, \sigma_n <: \tau_n \\
  \{ l_1 : \sigma_1, \ldots, l_n : \sigma_n \} <: \{ l_1 : \tau_1, \ldots, l_n : \tau_n \}
  \]

Implementation of Subtyping
For the implementation, we need to synthesize the coercion functions if we want to avoid runtime checks.

We define a new relation \( \tau_1 <: \tau_2 \rightarrow f \)

- \( \tau_1 \) is subclass of \( \tau_2 \), and \( f \) is the coercion function of type \( \tau_1 \rightarrow \tau_2 \)

\[
\begin{align*}
\tau_1 \rightarrow \tau_2 \rightarrow f \\
\sigma_1 \rightarrow \sigma_2 \rightarrow f, \text{ where } f(g(x) = f_0(g(f_1(x)))
\end{align*}
\]

Implementation of Subtyping
11
Records and Subtyping
12
Records and Subtyping

- Subtyping complicates efficient implementation of field selection
- Record type no longer reveals fields and types of fields of record
- Possible solution: coercion creates a new copy of the record with adjusted number of fields
  - `drop`: create a copy of record, delete fields
  - `copy`: create a copy of record, apply respective coercion function to all components
- Problematic for mutable records
- Possible solution: separation of view from contents