CS 4110

Programming Languages & Logics

Lecture 30
Featherweight Java and Object Encodings

11 November 2016

Properties

Lemma (Preservation)

If $\Gamma \vdash e : C$ *and* $e \to e'$ *then there exists a type* C' *such that* $\Gamma \vdash e' : C'$ *and* $C' \leq C$.

Lemma (Progress)

Let e *be an expression such that* $\vdash e : C$. *Then either:*

1. e *is a value,*
2. *there exists an expression* e' *such that* $e \to e'$, *or*
3. $e = E[(B)\ (\mathit{new}\ A(\overline{v}))]$ *with* $A \not\leq B$.

Lemmas

Lemma (Method Typing)

If $mtype(m, C) = \overline{D} \to D$ *and* $mbody(m, C) = (\overline{x}, e)$ *then there exists types* C' *and* D' *such that* $\overline{x} : \overline{D}, \mathtt{this} : C' \vdash e : D'$ *and* $D' \leq D$.

Lemmas

Lemma (Method Typing)

If $mtype(m, C) = \overline{D} \rightarrow D$ *and* $mbody(m, C) = (\overline{x}, e)$ *then there exists types* C' *and* D' *such that* $\overline{x} : \overline{D}, \mathtt{this} : C' \vdash e : D'$ *and* $D' \leq D$.

Lemma (Substitution)

If $\Gamma, \overline{x} : \overline{B} \vdash e : C$ *and* $\Gamma \vdash \overline{u} : \overline{B'}$ *with* $\overline{B'} \leq \overline{B}$ *then there exists* C' *such that* $\Gamma \vdash [\overline{x} \mapsto \overline{u}]e : C'$ *and* $C' \leq C$.

Lemmas

Lemma (Method Typing)

If $mtype(m, C) = \overline{D} \to D$ *and* $mbody(m, C) = (\overline{x}, e)$ *then there exists types* C' *and* D' *such that* $\overline{x} : \overline{D}, \mathtt{this} : C' \vdash e : D'$ *and* $D' \leq D$.

Lemma (Substitution)

If $\Gamma, \overline{x} : \overline{B} \vdash e : C$ *and* $\Gamma \vdash \overline{u} : \overline{B'}$ *with* $\overline{B'} \leq \overline{B}$ *then there exists* C' *such that* $\Gamma \vdash [\overline{x} \mapsto \overline{u}]e : C'$ *and* $C' \leq C$.

Lemma (Weakening)

If $\Gamma \vdash e : C$ *then* $\Gamma, x : B \vdash e : C$.

Lemmas

Lemma (Decomposition)

If $\Gamma \vdash E[e] : C$ *then there exists a type* B *such that* $\Gamma \vdash e : B$

Lemmas

Lemma (Decomposition)

If $\Gamma \vdash E[e] : C$ *then there exists a type* B *such that* $\Gamma \vdash e : B$

Lemma (Context)

If $\Gamma \vdash E[e] : C$ *and* $\Gamma \vdash e : B$ *and* $\Gamma \vdash e' : B'$ *with* $B' \leq B$ *then there exists a type* C' *such that* $\Gamma \vdash E[e'] : C'$ *and* $C' \leq C$.

Operational Semantics

$$E ::= [\cdot] \mid E.f \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ C(\overline{v}, E, \overline{e}) \mid (C)\, E$$

Operational Semantics

$$E ::= [\cdot] \mid E.f \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\, C(\overline{v}, E, \overline{e}) \mid (C)\, E$$

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']} \ \textsc{E-Context}$$

Operational Semantics

$$E ::= [\cdot] \mid E.f \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ C(\overline{v}, E, \overline{e}) \mid (C)\ E$$

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']} \ \textsc{E-Context}$$

$$\frac{\mathit{fields}(C) = \overline{C\ f}}{\texttt{new}\ C(\overline{v}).f_i \rightarrow v_i} \ \textsc{E-Proj}$$

Operational Semantics

$$E ::= [\cdot] \mid E.f \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ C(\overline{v}, E, \overline{e}) \mid (C)\, E$$

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']} \ \textsc{E-Context}$$

$$\frac{\mathit{fields}(C) = \overline{C\, f}}{\texttt{new}\ C(\overline{v}).f_i \rightarrow v_i} \ \textsc{E-Proj}$$

$$\frac{\mathit{mbody}(m, C) = (\overline{x}, e)}{\texttt{new}\ C(\overline{v}).m(\overline{u}) \rightarrow [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C(\overline{v})]e} \ \textsc{E-Invk}$$

Operational Semantics

$$E ::= [\cdot] \mid E.f \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid \texttt{new}\ C(\overline{v}, E, \overline{e}) \mid (C)\ E$$

$$\frac{e \to e'}{E[e] \to E[e']}\ \textsc{E-Context}$$

$$\frac{\mathit{fields}(C) = \overline{C\ f}}{\texttt{new}\ C(\overline{v}).f_i \to v_i}\ \textsc{E-Proj}$$

$$\frac{\mathit{mbody}(m, C) = (\overline{x}, e)}{\texttt{new}\ C(\overline{v}).m(\overline{u}) \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C(\overline{v})]e}\ \textsc{E-Invk}$$

$$\frac{C \leq D}{(D)\ \texttt{new}\ C(\overline{v}) \to \texttt{new}\ C(\overline{v})}\ \textsc{E-Cast}$$

Lemmas

Lemma (Canonical Forms)

If $\vdash v : C$ *then* $v = \mathtt{new}\, C(\overline{v})$.

Lemmas

Lemma (Canonical Forms)

If $\vdash v : C$ *then* $v = \mathtt{new}\, C(\overline{v})$.

Lemma (Inversion)

1. *If* $\vdash (\mathtt{new}\, C(\overline{v})).f_i : C_i$ *then* $\mathtt{fields}(C) = \overline{C\, f}$ *and* $f_i \in \overline{f}$.
2. *If* $\vdash (\mathtt{new}\, C(\overline{v})).m(\overline{u}) : C$ *then* $\mathtt{mbody}(m, C) = (\overline{x}, e)$ *and* $|\overline{u}| = |\overline{e}|$.

Typing Rules

$$\frac{\Gamma(x) = C}{\Gamma \vdash x : C}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e : C \qquad \mathit{fields}(C) = \overline{C\,f}}{\Gamma \vdash e.f_i : C_i}\ \text{T-FIELD}$$

$$\frac{\Gamma \vdash e : C \qquad \mathit{mtype}(m, C) = \overline{B} \to B \qquad \Gamma \vdash \overline{e} : \overline{A} \qquad \overline{A} \leq \overline{B}}{\Gamma \vdash e.m(\overline{e}) : B}\ \text{T-INVK}$$

$$\frac{\mathit{fields}(C) = \overline{C\,f} \qquad \Gamma \vdash \overline{e} : \overline{B} \qquad \overline{B} \leq \overline{C}}{\Gamma \vdash \texttt{new}\, C(\overline{e}) : C}\ \text{T-NEW}$$

$$\frac{\Gamma \vdash e : D \qquad D \leq C}{\Gamma \vdash (C)\, e : C}\ \text{T-UCAST} \qquad \frac{\Gamma \vdash e : D \qquad C \leq D \qquad C \neq D}{\Gamma \vdash (C)\, e : C}\ \text{T-DCAST}$$

$$\frac{\Gamma \vdash e : D \qquad C \not\leq D \qquad D \not\leq C \qquad \mathit{stupid\ warning}}{\Gamma \vdash (C)\, e : C}\ \text{T-SCAST}$$

Object Encodings

Object-Oriented Features

- Dynamic dispatch
- Encapsulation
- Subtyping
- Inheritance
- Open recursion

Dynamic Dispatch

```
interface Shape {
  ...
  void draw() { ... }
}

class Circle extends Shape {
  ...
  void draw() { ... }
}

class Square extends Shape {
...
void draw() { ... }
}

/*could be a circle, square, or something else */
Shape s = ...;
s.draw();
```

Encapsulation

```
class Circle extends Shape {
  private Point center;
  private int radius;
  ...
  public Point getX() { return center.x }
  public Point getY() { return center.y }
}
```

Subtyping

Subtyping fits naturally with object-oriented languages because (ignoring languages such as Java that allow certain objects to manipulate instance variables directly) the only way to interact with an object is to invoke a method

As a result, an object that supports the same methods as another object can be used wherever the second is expected

Example: a method that takes an object of type `Shape` can be passed a `Circle`, `Square`, or any other subtype of `Shape`, because they each support the methods listed in the `Shape` interface

Inheritance

```
class A {
  public int f(...) { ... g(...) ... }
  public bool g(...) { ... }
}

class B extends A {
  public bool g(...) { ... }
}
...
new B.f(...)
```

Open Recursion

Many object-oriented languages allow objects to invoke their own methods using the special keyword `this` (or `self`)

Implementing `this` in the presence of inheritance requires deferring the binding of `this` until the object is actually created

We will see an example of this next...

Record Encoding

```
type pointRep = { x:int ref; y:int ref }
```

Record Encoding

```
type pointRep = { x:int ref; y:int ref }

type point = { movex:int -> unit;
               movey:int -> unit }
```

Record Encoding

```
type pointRep = { x:int ref; y:int ref }

type point = { movex:int -> unit;
               movey:int -> unit }

let pointClass : pointRep -> point =
  (fun (r:pointRep) ->
    { movex = (fun d -> r.x := !(r.x) + d);
      movey = (fun d -> r.y := !(r.x) + d) })
```

Record Encoding

```
type pointRep = { x:int ref; y:int ref }

type point = { movex:int -> unit;
               movey:int -> unit }

let pointClass : pointRep -> point =
  (fun (r:pointRep) ->
    { movex = (fun d -> r.x := !(r.x) + d);
      movey = (fun d -> r.y := !(r.x) + d) })

let newPoint : int -> int -> point =
  (fun (x:int) ->
    (fun (y:int) ->
      pointClass { x=ref x; y = ref y }))
```

Inheritance

```
type point3D = { movex:int -> unit;
                 movey:int -> unit;
                 movez:int -> unit }
```

Inheritance

```
type point3D = { movex:int -> unit;
                 movey:int -> unit;
                 movez:int -> unit }

let point3DClass : point3DRep -> point3D =
  (fun (r:point3DRep) ->
    let super = pointClass r in
    { movex = super.movex;
      movey = super.movey;
      movez = (fun d -> r.z := !(r.x) + d) } )
```

Inheritance

```
type point3D = { movex:int -> unit;
                 movey:int -> unit;
                 movez:int -> unit }

let point3DClass : point3DRep -> point3D =
  (fun (r:point3DRep) ->
    let super = pointClass r in
    { movex = super.movex;
      movey = super.movey;
      movez = (fun d -> r.z := !(r.x) + d) } )

let newPoint3D : int -> int -> int -> point3D =
  (fun (x:int) ->
    (fun (y:int) ->
      (fun (z:int) ->
        point3DClass { x=ref x; y = ref y; z = ref z })))
```

Open Recursion With Self

```
type altPointRep = { x:int ref; y:int ref }
```

Open Recursion With Self

```
type altPointRep = { x:int ref; y:int ref }

type altPoint = { movex:int -> unit;
                  movey:int -> unit;
                  move:  int -> int -> unit }
```

Open Recursion With Self

```
type altPointRep = { x:int ref; y:int ref }

type altPoint = { movex:int -> unit;
                  movey:int -> unit;
                  move:  int -> int -> unit }

let altPointClass : altPointRep -> altPoint ref -> altPoint =
  (fun (r:altPointRep) ->
    (fun (self:altPoint ref) ->
      { movex = (fun d -> r.x := !(r.x) + d);
        movey = (fun d -> r.y := !(r.y) + d);
        move = (fun dx dy -> (!self.movex) dx;
                             (!self.movey) dy) }))
```

Open Recursion with Self

```
let dummyAltPoint : altPoint =
  { movex = (fun d -> ());
    movey = (fun d -> ());
    move = (fun dx dy -> ()) }
```

Open Recursion with Self

```
let dummyAltPoint : altPoint =
  { movex = (fun d -> ());
    movey = (fun d -> ());
    move = (fun dx dy -> ()) }

let newAltPoint : int -> int -> altPoint =
  (fun (x:int) ->
    (fun (y:int) ->
      let r = { x=ref x; y = ref y } in
      let cref = ref dummyAltPoint in
      cref := altPointClass r cref;
      !cref ))
```