Lecture 30
Featherweight Java and Object Encodings

11 November 2016
Properties

Lemma (Preservation)

If $\Gamma \vdash e : C$ and $e \rightarrow e'$ 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 $e \rightarrow e'$, or
3. $e = E[(B) (new A(\overline{v}))]$ with $A \not\leq B$. 
Lemma (Method Typing)

If \( \text{mtype}(m, C) = \overline{D} \rightarrow D \) and \( \text{mbody}(m, C) = (\overline{x}, e) \) then there exists types \( C' \) and \( D' \) such that \( \overline{x} : \overline{D}, \ this : C' \models e : D' \) and \( D' \leq D \).
Lemma (Method Typing)

If \( \text{mtype}(m, C) = \overline{D} \rightarrow D \) and \( \text{mbody}(m, C) = (\overline{x}, e) \) then there exists types \( C' \) and \( D' \) such that \( \overline{x} : \overline{D} \), \( \text{this} : C' \vdash e : D' \) 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 (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 $\overline{x} : \overline{D}$, $\text{this} : C' \vdash e : D'$ 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 $\Gamma, x : B \vdash e : C$. 
Lemma (Decomposition)

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

\[
E ::= [] \mid E.f \mid E.m(\bar{e}) \mid \nu.m(\bar{v}, E, \bar{e}) \mid \text{new } C(\bar{v}, E, \bar{e}) \mid (C)\ E
\]

\[
e \rightarrow e' \\
\frac{}{E[e] \rightarrow E[e']} \quad \text{E-CONTEXT}
\]

\[
\text{fields}(C) = \overline{C} f \\
\frac{}{\text{new } C(\bar{v}).f_i \rightarrow \nu_i} \quad \text{E-PROJ}
\]

\[
\text{mbody}(m, C) = (\bar{x}, e) \\
\frac{}{\text{new } C(\bar{v}).m(\bar{u}) \rightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})]e} \quad \text{E-INVK}
\]

\[
C \leq D \\
\frac{}{(D) \text{new } C(\bar{v}) \rightarrow \text{new } C(\bar{v})} \quad \text{E-CAST}
\]
Lemma (Canonical Forms)

\[ \text{Lemma (Canonical Forms)} \]

\[ \text{If } \vdash v : C \text{ then } v = \text{new } C(\overline{v}). \]

Lemma (Inversion)

1. \[ \text{If } \vdash (\text{new } C(\overline{v})).f_i : C_i \text{ then } \text{fields}(C) = \overline{Cf} \text{ and } f_i \in \overline{f}. \]
2. \[ \text{If } \vdash (\text{new } C(\overline{v})).m(\overline{u}) : C \text{ then } m\text{body}(m, C) = (\overline{x}, e) \text{ and } |\overline{u}| = |\overline{e}|. \]
Typing Rules

\[ \frac{\Gamma(x) = C}{\Gamma \vdash x : C} \]  
\text{\textsc{T-VAR}}

\[ \frac{\Gamma \vdash e : C}{\Gamma \vdash e.f_i : C_i} \]  
\text{\textsc{T-FIELD}}

\[ \frac{\Gamma \vdash e : C \quad \text{fields}(C) = \overline{C} f}{\overline{\Gamma} \vdash e : \overline{f}} \]  
\text{\textsc{T-FIELD}}

\[ \frac{\Gamma \vdash e : C \quad \text{mtype}(m, C) = \overline{B} \rightarrow B}{\Gamma \vdash e.m(\overline{e}) : B} \]  
\text{\textsc{T-INVK}}

\[ \frac{\Gamma \vdash e : C \quad \text{fields}(C) = \overline{C} f}{\Gamma \vdash \text{new} C(\overline{e}) : C} \]  
\text{\textsc{T-NEW}}

\[ \frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash (C)e : C} \]  
\text{\textsc{T-UCAST}}

\[ \frac{\Gamma \vdash e : D \quad C \leq D \quad C \neq D}{\Gamma \vdash (C)e : C} \]  
\text{\textsc{T-DCAST}}

\[ \frac{\Gamma \vdash e : D \quad C \not\leq D \quad D \not\leq C}{\Gamma \vdash (C)e : C} \]  
\text{\textsc{T-SCAST}}

\emph{stupid warning}
Object Encodings
Object-Oriented Features

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

definition of pointRep

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

construction of point

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

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

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

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) })
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 }))
type point3DRep = { x:int ref; y:int ref; z:int ref }

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 })))

11
type point3DRep = { x:int ref; y:int ref; z:int ref } 

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

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

type point3DRep = { x:int ref; y:int ref; z:int ref }

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

define altPointRep

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

define altPoint

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

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) }))
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 ))
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 ))