Semantics of Programming Languages

Semantics of COOL

Prof. Peter Müller

Software Component Technology

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

5. Semantics of Cool

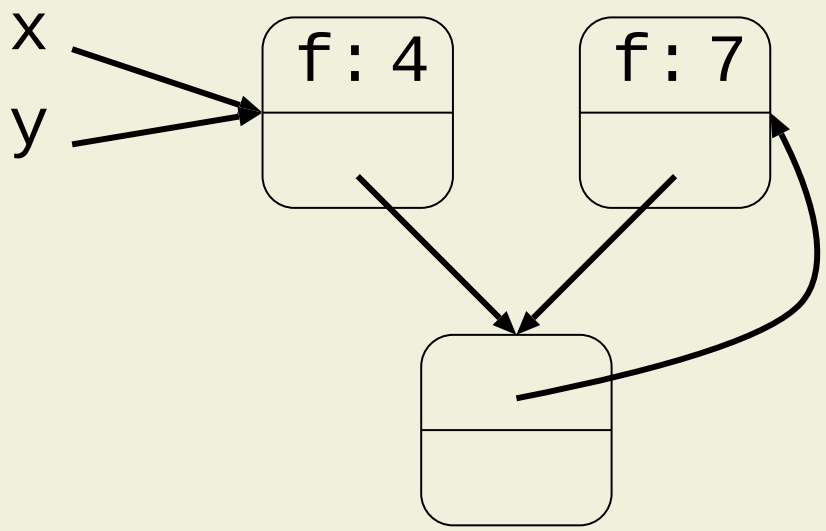
Motivation

- Realistic programming languages are significantly more complex than IMP

- For semantics, the main challenges of sequential object-oriented languages are:
  - Heap-allocated data structures (objects)
  - References (pointers)
  - Subtyping
  - Dynamic method binding

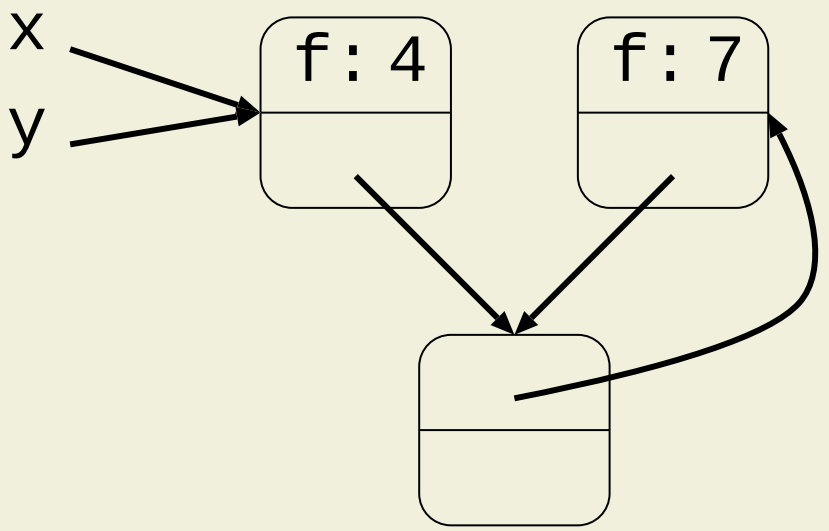
Objects and References

- Objects are accessed via references
- Static and dynamic aliasing
- Destructive Updates



Objects and References

- Objects are accessed via references
- Static and dynamic aliasing
- Destructive Updates



$$\{\mathsf{y} = V\}$$
$$\mathsf{x} := e$$
$$\{\mathsf{y} = V\}$$

$$\{\mathsf{y.f} = V\}$$
$$\mathsf{x.f} := e$$
$$\{\mathsf{y.f} = V\}$$

Not valid if x and y point to the same object

Subtyping

- Types of objects are not known statically
- At runtime, objects can have additional / different behavior

Without subtyping

$\{tt\}$
```
  var x: T := e1;
  var y: S := e2;
```
$\{x \neq y\}$

With subtyping

$\{tt\}$
```
  var x: T := e1;
  var y: S := e2;
```
$\{x \neq y\}$

Not valid if `S` and `T` are subtypes and `e1` and `e2` evaluate to the same value

Dynamic Method Binding

- Method implementation is selected at runtime
- No static connection between method call and method implementation

Static binding

$$\frac{\{\,\mathbf{P}\,\}\ \mathit{body}(p)\ \{\,\mathbf{Q}\,\}}{\{\,\mathbf{P}\,\}\ \texttt{call}\ p\ \{\,\mathbf{Q}\,\}}$$

Dynamic binding

$$\frac{\{\,\mathbf{P}\,\}\ \mathit{body}(m)\ \{\,\mathbf{Q}\,\}}{\{\,\mathbf{P}\,\}\ x.m()\ \{\,\mathbf{Q}\,\}}$$

$\mathit{body}(m)$ not known at compile time

The COOL Language

- COOL stands for Core Object-Oriented Language
- COOL is a subset of sequential Java
- It provides
  - Classes and interfaces
  - Fields and methods
  - Objects and values
- We do not consider arrays, exceptions, etc.

Example

```
interface Set {
 Set insert(int p);
}
```

```
class List
      implements Set {
 int  elem;
 List next;
```

```
boolean isElem(int p) {
 List ptr := this;
 while (ptr # null) {
  int e := ptr.elem;
  if (e = p)
   return true;
  ptr := ptr.next;
 }
 return false;
}

 List insert(int p) {
  ...
 }
}
```

Syntax of COOL

- The syntax for interfaces, classes, fields, and methods is identical to Java
  - ClassId and InterfaceId are the sets of globally unique class and interface names
  - TypeId $=$ ClassId $\cup$ InferfaceId
  - FieldId is the set of globally unique field names
- Expressions

$$\begin{aligned} \text{Exp} &= \text{Var} \mid \text{Integer} \mid \text{'}\texttt{null}\text{'} \\ &\quad \mid \text{Exp Op Exp} \mid \text{Unop Exp} \end{aligned}$$

  - Op and Unop are the usual unary and binary operators
  - Expressions are side effect free

Syntax of COOL: Statements

```
Stm  = ...
     | Var '.' FieldId ':=' Exp
     | Var ':=' Var '.' FieldId
     | Var ':=' '(' TypeId ')' Exp
     | Var ':=' 'new' ClassId
```

- Composition, conditional, etc. are like in IMP
- We will not discuss method invocations

Types and Subtyping

- Types

  $$\text{Type} = \text{boolT} \mid \text{intT} \mid \text{nullT} \mid \text{ClassId} \mid \text{InterfaceId}$$

  - nullT is the type of the `null` reference

- Subtyping

  $\preceq$: Type $\times$ Type $\longrightarrow$ Bool (subtype relation)

  $\prec$: Type $\times$ Type $\longrightarrow$ Bool (proper subtype relation)

- Examples

  - `List` $\preceq$ `List`, `List` $\preceq$ `Set` , nullT $\preceq$ `List`
  - `List` $\not\preceq$ `List`, intT $\not\preceq$ `List`

5. Semantics of Cool

5.1 Language

5.2 Object Stores

5.3 Axiomatic Semantics

Values, States, and Stores in IMP

- Values: Sort Val $= \mathbb{Z}$
  - Cool has integers, booleans, references
- States: Finite functions Var $\rightarrow$ Val
  - In Cool, values are not only stored in local variables, but also in fields of objects
- Stores: Finite functions Loc $\rightarrow$ Val
  - Stores were used to model static scope rules
  - We will use stores to model the heap memory

Recursive Data Types

$$
\begin{array}{l}
\textbf{datatype } \mathit{Sort} \\
\quad \mathit{cons}_1(\mathit{Sort}_1^1, \ldots, \mathit{Sort}_1^n) \\
\quad \ldots \\
\quad \mathit{cons}_m(\mathit{Sort}_m^1, \ldots, \mathit{Sort}_m^k) \\
\textbf{end}
\end{array}
$$

- Recursive data types declare
  - A sort (*Sort*)
  - The constructors for this sort ($\mathit{cons}_1, \ldots, \mathit{cons}_m$) and their arguments

Values

$\mathbf{datatype}$ Val

$b(\text{Bool})$

$i(\mathbb{Z})$

null

ref(ClassId, ObjId)

$\mathbf{end}$

typeof : Val $\rightarrow$ Type

$$\begin{aligned} \textit{typeof}(b(B)) &= \text{boolT} \\ \textit{typeof}(i(I)) &= \text{intT} \\ \textit{typeof}(\textit{null}) &= \text{nullT} \\ \textit{typeof}(\textit{ref}(C, X)) &= C \end{aligned}$$

- A value is a boolean value, an integer, the null reference, or an object reference
- Objects are identified by their class and an object identifier (sort ObjId)

Locations

- In the denotational semantics of IMP, we determined the state by a variable environment and a store

$$\text{Var} \xrightarrow{\Phi_V} \text{Loc} \xrightarrow{\$} \text{Val}$$

  - The variable environment statically maps variables to locations

- Fields are mappings from objects to locations

$$\text{Val} \xrightarrow{\text{FieldId}} \text{Loc} \xrightarrow{\$} \text{Val}$$

  - The location for field f of the object referenced by X is denoted by $X.f$

Stores

- Stores are modeled by sort Store and the following five operations:

$\_(\_)$	: Store $\times$ Loc $\longrightarrow$ Val
$\_\langle\_ := \_\rangle$	: Store $\times$ Loc $\times$ Val $\longrightarrow$ Store
new	: Store $\times$ ClassId $\longrightarrow$ Val
$\_\langle\_\rangle$	: Store $\times$ ClassId $\longrightarrow$ Store
alloc	: Val $\times$ Store $\longrightarrow$ Bool

- Object creation is modeled by two functions
  - $\_\langle\_\rangle$ yields new store
  - *new* yields reference to new object

Properties of Stores

st1: $L \neq K \Rightarrow OS\langle L := X\rangle(K) = OS(K)$

st2: $OS\langle X.f := Y\rangle(X.f) = Y$

st3: $OS\langle C\rangle(L) = OS(L)$

- A location update modifies only the updated location (**st1** and **st2**)
- Object creation does not change the values of locations (**st3**)

Properties of Stores (cont'd)

st4: $\mathit{alloc}(X, OS\langle L := Y\rangle) \Leftrightarrow \mathit{alloc}(X, OS)$

st5: $\mathit{alloc}(X, OS\langle C\rangle) \Leftrightarrow \mathit{alloc}(X, OS) \vee X = \mathit{new}(OS, C)$

st6: $\mathit{alloc}(OS(L), OS)$

- Updating a location does not change liveness (**st4**)
- An object is allocated in the store after an object creation iff it was allocated in the store before the creation or if it is the new object (**st5**)
- Locations never hold references to non-allocated objects (**st6**)

Properties of Stores (cont'd)

st7: $\neg \mathit{alloc}(\mathit{new}(OS, C), OS)$

st8: $\mathit{typeof}(\mathit{new}(OS, C)) = C$

- A new object is not allocated in the store in which it was created (**st7**)
- The type of a new object is the class specified in the *new* operation (**st8**)

Reachability

- Object can reach each other via **reference chains**

$$\mathit{reach}\_ : \mathbb{N} \times \mathsf{Val} \times \mathsf{Val} \times \mathsf{Store} \times \mathsf{FieldId} \to \mathsf{Bool}$$

$$\mathit{reach}_0(X, Y, \mathit{OS}, f) \quad \Leftrightarrow X = Y$$

$$\mathit{reach}_{N+1}(X, Y, \mathit{OS}, f) \quad \Leftrightarrow \exists Z : \mathit{OS}(X.f) = Z \wedge \mathit{reach}_N(Z, Y, \mathit{OS}, f)$$

$$\mathit{reach} : \mathsf{Val} \times \mathsf{Val} \times \mathsf{Store} \times \mathsf{FieldId} \to \mathsf{Bool}$$

$$\mathit{reach}(X, Y, \mathit{OS}, f) \quad \Leftrightarrow \exists N : \mathit{reach}_N(X, Y, \mathit{OS}, f)$$

Reachability: Example

- We can express that a singly-linked list is acyclic

```
class List implements Set {
 int  elem;
 List next;
 ...
```

$$\forall X, OS : \mathit{typeof}(X) \preceq \texttt{List} \Rightarrow \mathit{reach}(X, \mathit{null}, OS, \texttt{next})$$

- Such properties are often specified as class or object **invariant**

Abstraction Functions

- Abstraction functions map values, objects, or object structures to mathematical entities
- Important concept for specification and verification
  - Clients of a class only have to know how methods manipulate the abstract value of a data structure
  - Implementation details can be hidden from clients

- Abstraction function for integer values

$$aI : \mathsf{Val} \rightarrow \mathbb{Z}$$

$$aI(i(I)) = I$$

Abstraction Functions: Example

- Abstraction function for `List` structures
  - The empty list is represented by the null reference
  - Recursive definition is only well-defined if list is acyclic

$$aS : \text{Val} \times \text{Store} \to \mathcal{P}(\mathbb{Z})$$

$$aS(\mathit{null}, \mathit{OS}) = \emptyset$$

$$\mathit{typeof}(X) \preceq \texttt{List} \wedge X \neq \mathit{null} \wedge \mathit{reach}(X, \mathit{null}, \mathit{OS}, \texttt{next}) \Rightarrow$$

$$aS(X, \mathit{OS}) = aS(\mathit{OS}(X.\texttt{next}), \mathit{OS}) \cup \{\ aI(\mathit{OS}(X.\texttt{elem}))\ \}$$

- Subtyping
  - aS can also be used for interface `Set`
  - Defintion of aS depends on concrete subtype of `Set`

5. Semantics of Cool

5.1 Language

5.2 Object Stores

5.3 Axiomatic Semantics

Meaning of Assertions

- The meaning of $\{ \mathbf{P} \}\ s\ \{ \mathbf{Q} \}$ is a **refined** partial correctness

  If $\mathbf{P}$ holds in the initial state σ then the execution of s from σ

  1. terminates in a state in which $\mathbf{Q}$ will hold,
  2. leads to an out-of-memory exception, or
  3. loops

- Memory errors are outside the scope of language semantics

  - They require a model of the hardware and/or operating system

Field Read

- Assignment in IMP: $x := e$

$$\{ \mathbf{P}[x \mapsto \mathcal{A}[\![e]\!]] \}\ x := e\ \{ \mathbf{P} \}$$

- Field read: $x := y.f$
  - We use the store to determine the value of the right hand side
  - The receiver object has to be different from *null*

$$\{ y \neq \mathit{null} \wedge \mathbf{P}[x \mapsto \$(y.f)] \}\ x := y.f\ \{ \mathbf{P} \}$$

Field Update

- Assignment in IMP: $x := e$

$$\{ \mathbf{P}[x \mapsto \mathcal{A}[\![e]\!]] \}\ x := e\ \{ \mathbf{P} \}$$

- Field update: $x.f := e$
  - Updates modify the object store
  - The rule works like the assignment rule, but substitutes the store
  - $\mathcal{E}[\![e]\!]$ is the evaluation of expression e

$$\{ x \neq \mathit{null} \wedge \mathbf{P}[\$ \mapsto \$\langle x.f := \mathcal{E}[\![e]\!] \rangle] \}\ x.f := e\ \{ \mathbf{P} \}$$

Field Access: Example

- We prove termination of method `isElem`
- Loop variant:

$$\mathbf{V}(N) \equiv$$

$$\mathit{reach}_N(\texttt{ptr}, \texttt{null}, \$, \texttt{next})$$

- We show the following total correctness assertion

$$\{\, \mathit{reach}(\texttt{this}, \texttt{null}, \$, \texttt{next}) \,\}\ \mathit{body}(\texttt{isElem})\ \{\, \Downarrow tt \,\}$$

```
boolean isElem(int p) {
 List ptr := this;
 result := false;
 boolean c := true;
 while (ptr#null && c) {
  int e := ptr.elem;
  if (e = p) {
   result := true;
   c := false;
  }
 ptr := ptr.next;
 }
}
```

Object Creation

- The new statement allocates a new object and returns a reference to it
- The rules works like the assignment rule, but substitutes the store **and** the variable

$$\{\ \mathbf{P}[\$ \mapsto \$\langle C \rangle][x \mapsto \mathit{new}(\$, C)]\ \}\ x := \text{new}\ C\ \{\ \mathbf{P}\ \}$$

- Memory errors are not considered

Purity

- A statement or method is called **pure** if it does not modify the locations of objects that are allocated in the prestate
  - Objects can be created and initialized
  - Purity is important for sharing, contracts, thread synchronization, etc.

- To show purity of s, one has to prove for all X, f:

$$\{\ \mathit{alloc}(X, \$) \wedge \$(X.f) = V\ \}\ s\ \{\ \$(X.f) = V\ \}$$

Purity: Example

- We prove that method `insert` is pure

```
List insert(int p) {
 result := new List;
 result.elem := p;
 result.next := this;
}
```

$$\{\ \mathit{alloc}(X, \$) \wedge \$(X.f) = V\ \}\ \mathit{body}(\texttt{insert})\ \{\ \$(X.f) = V\ \}$$

Type Casts

- The cast statement converts the static type of an expression
- In COOL, casts are combined with an assignment: $x := (T)\, e$
- A runtime check guarantees that the type conversion is legal
  - The type of the value denoted by e must be a subtype of T

$$\{\ \mathit{typeof}(\mathcal{E}[\![e]\!]) \preceq T \wedge \mathbf{P}[x \mapsto \mathcal{E}[\![e]\!]]\ \}\ x := (T)\, e\ \{\ \mathbf{P}\ \}$$

Type Cast: Example

- We prove that the following statement terminates normally:

```
Set s := new List;
List l := (List) s;
l.next := null
```

$\{\ tt\ \}$ `s:=new List;l:=(List)s;l.next:=null` $\{\ tt\ \}$

Further Reading

- We have developed an axiomatic semantics for a large subset of sequential Java
  - A. Poetzsch-Heffter and P. Müller: *Logical Foundations for Typed Object-Oriented Languages*. In D. Gries and W. P. De Roever: Programming Concepts and Methods (PROCOMET), 1998.
  - A. Poetzsch-Heffter and P. Müller: *A Programming Logic for Sequential Java*. In S. D. Swierstra: Programming Languages and Systems (ESOP), Lecture Notes in Computer Science 1576, Springer-Verlag, 1999.
- The papers are available from the course web site

Soundness and Completeness

- Soundess is proven w.r.t. an operational Java semantics
- The logic is not complete

```
x := null;
x.f := e;
```

  - This statement would lead to a **stuck configuration** in an operational semantics
  - It is **not possible to prove anything** about the above statement in the axiomatic semantics
  - To achieve completeness, we would have to include exceptions in the axiomatic semantics