Virtual Machines

Lecture 7-8

–

CoreJava Type System and Third Assignment

Overview

1. Discussion about the Second Assignment – CoreJava Dynamic Semantics – Questions on the Interpreter

2. Introduction in Static Semantics -- Type Systems

3. Third Assignment - CoreJava Type System

Introduction in Static Semantics

--

Type Systems

Static semantics

We use the simpler expression language to introduce the type system. For example the language contains Booleans, conjunction, and **`if`** expressions :

```
e ::= ... | b | e1 && e2
     | if e1 then e2 else e3
v ::= ... | b
```

We could get nonsensical expressions, e.g.,

```
5 + false
if 5 then true else 0
```

Need *static semantics* (type checking) to rule those out...

if expressions

Syntax:

if e1 then e2 else e3

Type checking:

if **e1** has type **bool** and **e2** has type **t** and **e3** has type **t**
then **if e1 then e2else e3** has type **t**

Static semantics

Defined by a *judgement*:

`T |-e : t`

- Read as in typing context `T`, expression `e` has type `t`
- Turnstile `|-` can be read as "proves" or "shows"
- You're already used to `e : t`, because Ocaml utop uses that notation
- *Typing context* is a dictionary mapping variable names to types
- The typing context is a new idea, but obviously needed to give types of variables in scope

Static semantics

e.g.,

```
x:int |-  x+2 : int
x:int,y:int |-  x<y : bool
|-  5+2 : int
```

Static semantics of expr. lang.

```
T |- i : int
T |- b : bool
T,x:t  |-   x: t
```

Static semantics of expr. lang.

T |- e1+ e2 : int

if T |- e1 : int

and T |- e2 : int

```
T |- e1 && e2 : bool
  if  T |- e1 : bool
  and T |- e2 : bool
```

Static semantics of expr. lang.

```
T |- if e1 then e2 else e3 : t
  if  T |- e1 : bool
  and T |- e2 : t
  and T |- e3 : t


T |- let x:t1 = e1 in e2 : t2
if     T |- e1  : t1
  and T,x:t1     |- e2     : t2
```

Interpreter for expr. lang.

See **`interp3-full.ml`** code attached to this lecture

1. Type checks expression
2. Evaluates expression

Purpose of type system

Ensure **type safety**: well-typed programs don't get *stuck:*

- haven't reached a value, and
- unable to evaluate further

Lemmas:

Progress: if `e` has type `t`, then either `e` is a value or `e` can take a step.
Preservation: if `e` has type `t`, and if `e` takes a step to `e'`, then `e'` has type `t`.

Type safety = progress + preservation

Proving type safety is more difficult and therefore we ignore it in this course. Type safety MUST always be proved, since the compiler MUST be correct.

Third Assignment

–

CoreJava Type System

Third Assignment – 25% of the final grade

Please implement in Ocaml a type checker for CoreJava language according to the CoreJava type system.

The type system (or the static semantics) of CoreJava is described in the following slides

CoreJava Type System

- In the following we present the type checking rules of all CoreJava.
- The presentation is not so formal as in the literature
- The judgements have the following form

conditions to be met

----

context |- type rule

<==>

(IF conds to be met)

THEN

for the given context

The type rule is true

CoreJava Type System

- The type system is presented top-down
- It consists of the following judgements for:
  - A well-typed program
  - A well-typed class declaration
  - Well-typed field declarations
  - A well-typed method declaration
  - A well-typed expression
  - Subtyping (you can also use this subtyping definition in the operational semnatics where we defined not so rigorous)

Well-typed program

|- WellFoundedClasses(P) and P=clsD1;...;clsDn and

For each class declaration clsDi we have:

|- methsOnce(clsDi) and |-fieldsOnce(clsDi) and

P |- inheritanceOK(clsDi) and P |-def- clsDi

|- P

- A program is well-typed if:
  - WellFoundedClasses: no duplicate definitions of the clases, no cycle in the class hierarchy and last class contains the main method
  - MethsOnce: no methods duplication in a class
  - FieldsOnce: no field duplication in a class
  - InheritanceOk: method overriding is sound
  - Each class is well typed

Well-typed class declaration

ClsD= class cn extends cn' {fldD1...fldDn # mthD1...mthDn} and

For each method declaration mthDi we have:

P, {this:cn} |-mth- mthDi

--

P |-def- clsD

- A class is well typed if:
  - Each method from the class is well typed
  - {this:cn} denotes the initial type environment
  - A type environment is a dictionary containing mappings from the variable name to the type associated to that variable
  - Type environment is working as a stack where we continously push new mappings

Well-typed method declaration

P, {v1:t1,..., vn:tn}+TE |- e : t   and P |- t <: tr

--

P,TE |-mth- tr mn(t1 v1, ..., tn vn) {e}

- A method is well typed if:
  - The method body is well typed
  - TE denotes the type environment
  - {v1:t1,..., vn:tn}+TE denotes the extension of a type environment TE with new mappings {v1:t1,...,vn:tn} corresponding to the formal parameters of the method
  - The judgement P,TE |- e:t says that the type of the expression e is t with respect to the program P and type environment TE
  - The type of the method body must be a subtype of the declared return type of the method

Subtyping Judgement

- In order to denote that a type t1 is a subtype of type t2 we used the following notation t1 <: t2
- The rules of the subtyping relation are enumerated in the following
- If none of the following rule is applicable that means that t1 is not subtype of t2
- Note that in Lecture 6 we presented a draft implementation of the subtyping relation

Subtyping Judgement

(inheritance rule)

```
Class cn1 extends c2 {...} is defined in P
--------------------------------------------------------------------
P |-  cn1 <: cn2
```

(reflexivity)

```
-------------------
P |- t<:t
```

(transitivity)

```
P |-  t1<:t2   and   P |- t2<:t3
------------------------------------
P |- t1 <:t3
```

Subtyping Judgement

Cn is a class in P

P |- bot <: cn

cn is a class in P

P |- cn <: Object

- Note that the above 5 rules directly imply the followings:
  - int <:int ,
  - float<:float ,
  - void <:void
  - bool <: bool

Well-typed expressions

P,TE |- null:bot

P,TE |- kint: int

P,TE |- kfloat:float

P,TE |- (): void

P,TE |- false:bool

P,TE |- true: bool

Well typed expressions

(v: t) is defined in TE

P,TE |- v : t

- The type of the variable v is the declared type of the variable v
- The declared type of a variable is stored in the type environment

Well typed expressions

P,TE |- v: cn and

(cn is a class defined in P) and

((f,t) is defined in fieldlist(P,cn))

--

P,TE |- v.f : t

- First we get the type of v, that type must be a class
- Second we get the type of the field f

fieldlist

---

fieldlist(P,Object) = []

class cn1 extends cn2 {t1 f1;...;tn fn # ....}

--

fieldlist(P,cn1)= fieldlist(P,cn2) ++ [(f1,t1);...(fn,tn))]

- It computes all fields of a class

Well typed expressions

P,TE |- v: t1 and

P,TE |- e : t2 and

P |- t2 <: t1

P,TE |- v=e : void

- The type t2 of the expression e must be a subtype of the variable v type t1

Well typed expressions

P,TE |- v.f :t1

P,TE |- e : t2 and

P |- t2 <: t1

P,TE |- v.f=e : void

Well typed expressions

P, {v:t} +TE |- e : t1

P,TE |- {(t v) e} : void

Well typed expressions

P,TE |- e1 :t1   and

P,TE |- e2 : t2

P,TE |- e1;e2 : t2

Well typed expressions

P,TE |- v :tv and P |- tv<:bool and

P,TE |- e1 : t1 and P,TE |- e2:t2 and

Find t such that

P |- t1 <: t and P |- t2 <: t

P,TE |- if v then e1 else e2 : t

Well typed expressions

P,TE |- v :tv   and P |- tv<:bool and

P,TE |- e1 : t1 and P,TE |- e2:t2 and

Find t such that

P |- t1 <: t   and P |- t2 <: t

P,TE |- if v then e1 else e2 : t

Well typed expressions

```
     P,TE |- v :tv    and P |- tv<:bool and
     P,TE |-  e1 : t1  and P,TE |- e2:t2 and
      Find t such that
          P |- t1 <: t   and  P |-  t2 <: t
-----------------------------------------------------
            P,TE |-  if v then e1 else e2 : t
```

Well typed expressions

P,TE |- e1 :t1   and P |- t1<:int and

P,TE |- e2 : t2 and P |- t2<:int

--

P,TE |- e1 opint e2 : int

Well typed expressions

P,TE |- e1 :t1   and P |- t1<:float and

P,TE |- e2 : t2 and P |- t2<:float

P,TE |- e1 opfloat e2 : float

Well typed expressions

(opbool is either && or ||) and

P,TE |- e1 :t1   and   P |- t1<:bool and

P,TE |- e2 : t2   and   P |- t2<:bool

--

P,TE |- e1 opbool e2 : bool

P,TE |- e : t   and   P |- t<:bool

--

P,TE |- !e : bool

Well typed expressions

(opcmp is either < or <= or == or != or > or >=) and

P,TE |- e1 :t1 and P,TE |- e2 : t2 and

t1==t2

P,TE |- e1 opcmp e2 : bool

Well typed expressions

(cn is a declared class in P) and

P,TE |- v :t   and

(P |- cn <: t   or   P |- t<: cn)

--

P,TE |- (cn) v : cn

Well typed expressions

(cn is a declared class in P) and

P,TE |- v :t

P,TE |- v instanceof cn : bool

Well typed expressions

```
(cn is a declared class in P) and
[(f1,t1),...,(fn,tn)]=fieldlist(P,cn) and
P,TE |- v1 :t1' and ... and P,TE |- vn:tn' and
P |- t1'<:t1 and ...  and P |- tn'<:tn
-----------------------------------------------------
P,TE |-  new cn(v1,...,vn) : cn
```

Well typed expressions

P,TE |- v0:t0 and (t0 is a declared class in P) and

P |-(tr mn(t1 v1,..., tn vn){e}) in t0   and

P,TE |- v1' :t1' and ... and P,TE |- vn':tn' and

P |- t1'<:t1 and ... and P |- tn'<:tn

--

P,TE |- v0.mn(v1',...,vn') : tr

Well typed expressions

```
P,TE |- v:t     and  P |- t <: bool and
        P,TE |- e : te
-----------------------------------------------------
     P,TE |- while v {e} : te
```

Auxilliary rules

clsD = class cn extends cn' {...# mthD1...mthDn}

For each i and j, 0<=i<=n and 0<=j<=n and i!=j

name(mthDi) != name(mthDj)

--

|- methsOnce(clsD)

- No method overloading/duplication in a class definition

Auxilliary rules

clsD = class cn extends cn' {fldD1...fldDn # ...}

For each i and j, 0<=i<=n and 0<=j<=n and i!=j

name(fldDi) != name(fldDj)

--

|- fieldsOnce(clsD)

- No field duplication in a class definition