JOOS Action Semantics

David A. Watt[1]

Version 1: 1997–10–28 (10.34)

[1]Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: daw@dcs.gla.ac.uk.

Preface

This document is an action-semantic specification of the abstract syntax and dynamic semantics of JOOS. The abstract syntax was adapted from one written by Peter Mosses. The semantics was written by David Watt.

JOOS is a Java subset designed by Laurie Hendren. It demonstrates the main concepts of Java: classes, inheritance, dynamic method selection, and constructors. However, JOOS lacks static fields and methods, and automatic type conversions. All fields are private, and fields may not be hidden. All methods and constructors are public, and they may not be overloaded.

Chapter 1

JOOS/Abstract Syntax

needs: [**Mosses 1992**]/**Data Notation/Characters** .

grammar:

1.1 Expressions

- Assignment = ⟦ Identifier "=" Expression ⟧ .
- Instance-Creation = ⟦ "new" Identifier "(" Arguments ")" ⟧ .
- Method-Invocation = ⟦ Expression "." Identifier "(" Arguments ")" ⟧ **|**
  ⟦ "super" "." Identifier "(" Arguments ")" ⟧ .
- Arguments = ⟨ Expression ⟨ "," Expression ⟩* ⟩? .
- Expression = Literal **|**
  Identifier **|**
  "this" **|**
  ⟦ Prefix-Operator Expression ⟧ **|**
  ⟦ Expression Infix-Operator Expression ⟧ **|**
  ⟦ Expression ("||" **|** "&&") Expression ⟧ **|**
  ⟦ Expression "instanceof" Identifier ⟧ **|**
  ⟦ "(" Type ")" Expression ⟧ **|**
  ⟦ "(" Expression ")" ⟧ **|**
  Assignment **|**
  Instance-Creation **|**
  Method-Invocation .
- Prefix-Operator = "-" **|** "!" .
- Infix-Operator = "==" **|** "!=" **|** "<" **|** ">" **|** "<=" **|** ">=" **|**
  "+" **|** "-" **|** "*" **|** "/" **|** "%" .
- Literal = Boolean-Literal **|** Integer-Literal **|** "null" .
- Boolean-Literal = "true" **|** "false" .
- Integer-Literal = ⟦ digit+ ⟧ .
- Identifier = ⟦ letter (letter **|** digit)* ⟧ .

1.2 Statements

needs: Declarations , Expressions .

- Block-Statements = ⟦ Variable-Declaration* Statement* ⟧ .
- Statement = ⟦ ";" ⟧ **|**
  ⟦ "{" Block-Statements "}" ⟧ **|**
  ⟦ (Assignment **|** Instance-Creation **|**
  Method-Invocation) ";" ⟧ **|**
  ⟦ "`return`" Expression? ";" ⟧ **|**
  ⟦ "`if`" "(" Expression ")" Statement "`else`" Statement ⟧ **|**
  ⟦ "`while`" "(" Expression ")" Statement ⟧ .

1.3 Declarations

needs: Statements , Expressions .

- Class-Declaration = ⟦ "`public`" "`final`"? "`class`" Identifier
  "`extends`" Identifier "{"
  Field-Declaration*
  Constructor-Declaration
  Method-Declaration* "}" ⟧ .
- Field-Declaration = ⟦ "`private`" Type Identifier ";" ⟧ .
- Constructor-Declaration = ⟦ "`public`" Identifier "(" Formal-Parameters ")" "{"
  "`super`" "(" Arguments ")" ";"
  Block-Statements "}" ⟧ .
- Method-Declaration = ⟦ "`public`" ("`void`" **|** Type) Identifier
  "(" Formal-Parameters ")" "{"
  Block-Statements "}" ⟧ .
- Formal-Parameters = ⟨ Formal-Parameter ⟨ "," Formal-Parameter ⟩* ⟩? .
- Formal-Parameter = ⟦ Type Identifier ⟧ .
- Variable-Declaration = ⟦ Type Identifier ";" ⟧ .
- Type = "`int`" **|** "`boolean`" **|** Identifier .

Chapter 2

JOOS/Semantic Functions

needs: **JOOS/(Abstract Syntax, Semantic Entities)** ,
[Mosses 1992]/Action Notation .

2.1 Expressions

introduces: evaluate _ , respectively evaluate _ ,
apply-prefix _ , apply-infix _ ,
the value of _ .

- evaluate _ :: Assignment →
  action [giving a value | storing | diverging | escaping]
  [using current bindings | current storage] .

(1) evaluate ⟦ I:Identifier "=" E:Expression ⟧ =
evaluate E then
| store the given value in the variable bound to I and
| give the given value .

- evaluate _ :: Instance-Creation →
  action [giving an object | storing | diverging | escaping]
  [using current bindings | current storage] .

(2) evaluate ⟦ "`new`" I:Identifier "(" A:Arguments ")" ⟧ =
| allocate an object of the class bound to I and
| respectively evaluate A
then
| enact the application of the constructor of the class bound to I
| to the given (object, value*) and
| give the given object#1 .

- evaluate _ :: Method-Invocation →
  action [giving a value$^?$ | storing | diverging | escaping]
  [using current bindings | current storage] .

(3) evaluate ⟦ E:Expression "." I:Identifier "(" A:Arguments ")" ⟧ =
| evaluate E and
| respectively evaluate A
then
| enact the application of the method I of the class of the given object#1
| to the given (object, value*)
| or
| | check the given reference#1 is null then
| | escape with the null-reference-exception .

(4) evaluate ⟦ "`super`" "." I:Identifier "(" A:Arguments ")" ⟧ =
respectively evaluate A then
enact the application of
    the method I of the superclass of the class of
        the object bound to "`this`"
    to (the object bound to "`this`", the given value*) .

- respectively evaluate _ :: Arguments →
  action [giving value* | storing | diverging | escaping]
      [using current bindings | current storage] .

(5) respectively evaluate ⟨ ⟩ =
give () .

(6) respectively evaluate E:Expression =
evaluate E .

(7) respectively evaluate ⟨ E:Expression "," A:Arguments ⟩ =
evaluate E and respectively evaluate A .

- evaluate _ :: Expression →
  action [giving a value | storing | diverging | escaping]
      [using current bindings | current storage] .

(8) evaluate L:Literal =
give the value of L .

(9) evaluate I:Identifier =
give the value stored in the variable bound to I .

(10) evaluate "`this`" =
give the object bound to "`this`" .

(11) evaluate ⟦ O:Prefix-Operator E:Expression ⟧ =
evaluate E
then apply-prefix O .

(12) evaluate ⟦ E_1:Expression O:Infix-Operator E_2:Expression ⟧ =
| evaluate E_1 and evaluate E_2
then apply-infix O .

(13) evaluate ⟦ E_1:Expression "||" E_2:Expression ⟧ =
evaluate E_1 then
| | check (the given value is true) then give true
| or
| | check (the given value is false) then evaluate E_2 .

(14) evaluate ⟦ E_1:Expression "&&" E_2:Expression ⟧ =
evaluate E_1 then
| | check (the given value is true) then evaluate E_2
| or
| | check (the given value is false) then give false .

(15) evaluate ⟦ E:Expression "`instanceof`" I:Identifier ⟧ =
evaluate E then
give (the given reference is an instance of the class bound to I) .

(16) evaluate ⟦ "(" T:Type ")" E:Expression ⟧ =
evaluate E .

(17) evaluate ⟦ "(" E:Expression ")" ⟧ =
evaluate E .

- apply-prefix _ :: Prefix-Operator →
action [giving a value] [using the given value] .

(18) apply-prefix "!" =
give not (the given truth-value) .

- apply-infix _ :: Infix-Operator →
action [giving a value] [using the given (value, value)] .

(19) apply-infix "==" =
give (the given value#1 is the given value#2) .

(20) apply-infix "+" =
give the sum of (the given integer#1, the given integer#2) .

Other prefix and infix operators are similar.

- the value of _ :: Literal → value .

Semantics of literals is straightforward and is omitted here.

2.2 Statements

needs: **Declarations** , **Expressions** .

introduces: execute _ .

- execute _ :: Block-Statements →
action [storing | diverging | escaping]
[using current bindings | current storage] .

(1) execute ⟦ V:Variable-Declaration* S:Statement* ⟧ =
furthermore elaborate V
hence execute S .

- execute _ :: Statement* →
action [storing | diverging | escaping]
[using current bindings | current storage] .

(2) execute ⟨ ⟩ =
complete .

(3) execute ⟦ “`;`” ⟧ =
complete .

(4) execute ⟦ “`{`” B:Block-Statements “`}`” ⟧ =
execute B .

(5) execute ⟦ E:(Assignment | Instance-Creation | Method-Invocation) “`;`” ⟧ =
evaluate E then complete .

(6) execute ⟦ “`return`” “`;`” ⟧ =
escape with the return of () .

(7) execute ⟦ “`return`” E:Expression “`;`” ⟧ =
evaluate E then
escape with the return of the given value .

(8) execute ⟦ “`if`” “`(`” E:Expression “`)`” S_1:Statement “`else`” S_2:Statement ⟧ =
evaluate E then
| | check (the given value is true) then execute S_1
| or
| | check (the given value is false) then execute S_2 .

(9) execute ⟦ “`while`” “`(`” E:Expression “`)`” S:Statement ⟧ =
unfolding
| evaluate E then
| | | check (the given value is true) then execute S then unfold
| | or
| | | check (the given value is false) then complete .

(10) execute ⟨ S_1:Statement S_2:Statement$^+$ ⟩ =
execute S_1 and then execute S_2 .

2.3 Declarations

needs: **Statements** , **Expressions** .

introduces: elaborate _ , the type-bindings of _ ,
the constructor of _ , the method-bindings of _ ,
respectively formally bind _ , formally bind _ , the type denoted by _ .

- elaborate _ :: Class-Declaration →
action [binding] [using current bindings] .

(1) elaborate ⟦ “`public`” X:“`final`”$^?$ “`class`” I:Identifier
“`extends`” I':Identifier “`{`”
F:Field-Declaration*
C:Constructor-Declaration
M:Method-Declaration* “`}`” ⟧ =
recursively bind I to the class of (
the type-bindings of F, the method-bindings of M,
the constructor of C, the class bound to I') .

- the type-bindings of _ :: Field-Declaration* $\rightarrow$ type-bindings .

(2) the type-bindings of $\langle\ \rangle$ =
the empty-map .

(3) the type-bindings of $[\![$ “`private`” T:Type I:Identifier “;” $]\!]$ =
the map of I to the type denoted by T .

(4) the type-bindings of $\langle$ F_1:Field-Declaration F_2:Field-Declaration$^+$ $\rangle$ =
the disjoint-union of (the type-bindings of F_1, the type-bindings of F_2) .

- the constructor of _ :: Constructor-Declaration $\rightarrow$
yielder [of a constructor] [using current bindings] .

(5) the constructor of $[\![$ “`public`” I:Identifier “(” F:Formal-Parameters “)” “{”
“`super`” “(” A:Arguments “)” “;”
B:Block-Statements “}” $]\!]$ =
the closure of the abstraction of
furthermore
bind “`this`” to the given object#1 and
produce the field-variable-bindings of the given object#1
hence
furthermore
give the rest of the given (object, value*) then
respectively formally bind F
hence
give the given object#1 and
respectively evaluate A
then enact the application of
the constructor of the superclass of the class bound to I
to the given (object, value*)
and then
execute B
trap a return then complete .

- the method-bindings of _ :: Method-Declaration* $\rightarrow$
yielder [of method-bindings] [using current bindings] .

(6) the method-bindings of $\langle\ \rangle$
the empty-map .

(7) the method-bindings of $[\![$ “`public`” T:(“`void`” | Type) I:Identifier
“(” F:Formal-Parameters “)” “{”
B:Block-Statements “}” $]\!]$ =
the map of I to the closure of the abstraction of
furthermore
bind “`this`” to the given object#1 and
produce the field-variable-bindings of the given object#1
hence
furthermore
give the rest of the given (object, value*) then
respectively formally bind F
hence
execute B
trap a return then give the returned-value of it .

(8) the method-bindings of ⟨ M_1:Method-Declaration M_2:Method-Declaration$^+$ ⟩
the disjoint-union of (the method-bindings of M_1,
the method-bindings of M_2) .

- respectively formally bind _ :: Formal-Parameters →
action [binding | storing] [using the given value* | current storage] .

(9) respectively formally bind ⟨ ⟩ =
complete .

(10) respectively formally bind F:Formal-Parameter =
formally bind F .

(11) respectively formally bind ⟨ F_1:Formal-Parameter "," F_2:Formal-Parameters ⟩ =
| give the first of the given value$^+$ then
| formally bind F_1
and
| give the rest of the given value$^+$ then
| respectively formally bind F_2 .

- formally bind _ :: Formal-Parameter →
action [binding | storing] [using the given value | current storage] .

(12) formally bind ⟦ T:Type I:Identifier ⟧ =
allocate a variable initialised to the given value then
bind I to it .

- elaborate _ :: Variable-Declaration* →
action [binding | storing] [using current storage] .

(13) elaborate ⟨ ⟩ =
complete .

(14) elaborate ⟦ T:Type I:Identifier ";" ⟧ =
allocate a variable then
bind I to it .

(15) elaborate ⟨ V_1:Variable-Declaration V_2:Variable-Declaration ⟩ =
elaborate V_1 before elaborate V_2 .

- the type denoted by _ :: Type → type .

(16) the type denoted by "`boolean`" =
the boolean-type .

(17) the type denoted by "`int`" =
the integer-type .

(18) the type denoted by I:Identifier =
the reference-type .

Chapter 3

JOOS/Semantic Entities

needs: [**Mosses 1992**]/(**Data Notation, Action Notation**) .

3.1 Data

needs: **Values** , **Variables** , **Types** , **Classes** , **Objects** , **Escapes** .

- datum = value | variable | type | class | method | constructor |
  type-bindings | variable-bindings | method-bindings | token |
  reason-for-escape (*disjoint*) .
- token = (letter, (letter | digit)*) .
- bindable = class | object | variable .
- storable = value .

3.2 Values

needs: **Objects** .

introduces: value .

- value = truth-value | integer | reference (*disjoint*) .

3.3 Variables

needs: **Values** .

introduces: variable , allocate a variable initialised to _ .

- variable = cell .

- allocate a variable initialised to _ :: yielder [of a value] →
  action [storing | giving a variable] [using current storage] .

(1) allocate a variable initialised to v:yielder [of a value] =
| allocate a variable and give v
then
| store the given value#2 in the given variable#1 and
| give the given variable#1 .

3.4 Types

needs: **Values** , **Objects** .

introduces: type , boolean-type , integer-type , reference-type ,
default-value _ .

- type = boolean-type **|** integer-type **|** reference-type (*individual*) .

- default-value _ :: type → value (*total*) .

(1) default-value (boolean-type) = false .

(2) default-value (integer-type) = 0 .

(3) default-value (reference-type) = null .

3.5 Classes

needs: **Data** , **Types** .

introduces: class ,
type-bindings , method-bindings , method , constructor ,
class of _ ,
field-type-bindings _ , method-bindings _ ,
constructor _ , superclass _ , superclasses _ ,
method _ of _ .

- class = class of (type-bindings, method-bindings, constructor, class$^?$) .
- type-bindings = map[token to type] .
- method-bindings = map[token to method] .
- method = abstraction [giving a value$^?$ **|** storing **|** diverging **|** escaping]
[using the given (object, value*) **|** current storage] .
- constructor = abstraction [storing **|** diverging **|** escaping]
[using the given (object, value*) **|** current storage] .

- class of _ :: (type-bindings, method-bindings, constructor, class$^?$) →
class (*total*, *injective*) .
- field-type-bindings _ :: class → type-bindings (*total*) .
- method-bindings _ :: class → method-bindings (*total*) .
- constructor _ :: class → constructor (*total*) .

- superclass _ :: class → class$^?$ (*total*) .
- superclasses _ :: class → set[class] (*total*) .

(1) c = class of (t:type-bindings, m:method-bindings, k:constructor) ⇒

  (1) field-type-bindings c = t ;
  (2) method-bindings c = m ;
  (3) constructor c = k ;
  (4) superclass c = () ;
  (5) superclasses c = set of c .

(2) c = class of (t:type-bindings, m:method-bindings, k:constructor, c':class) ⇒

  (1) field-type-bindings c = disjoint-union (t, field-type-bindings c') ;
  (2) method-bindings c = overlay (m, method-bindings c') ;
  (3) constructor c = k ;
  (4) superclass c = c' ;
  (5) superclasses c = union (set of c, superclasses c') .

- method _ of _ :: token, class → method (*partial*) .

(3) method t:token of c:class = method-bindings c at t .

3.6 Objects

needs: **Data** , **Variables** , **Types** , **Classes** .

introduces: reference , null , object , variable-bindings ,
object of _ , class _, field-variable-bindings _ ,
_ is _ , _ is an instance of _ ,
allocate an object of _ .

privately introduces: identity , identity _ , instantiate _ .

- reference = null | object (*disjoint*) .
- null : reference .
- object = object of (class, variable-bindings, identity) .
- variable-bindings = map[token to variable] .
- identity = cell .

- object of _ :: (class, variable-bindings, identity) → object (*total*, *injective*) .
- class _ :: object → class (*total*) .
- field-variable-bindings _ :: object → variable-bindings (*total*) .
- identity _ :: object → identity (*total*) .

(1) o = object of (c:class, v:variable-bindings, i:identity) $\Rightarrow$

- (1) class o = c ;
- (2) field-variable-bindings o = v ;
- (3) identity o = i .

- _ is _ :: reference, reference $\to$ truth-value (*total*, *commutative*) .

(2) null is null = true .

(3) null is o:object = false .

(4) o_1:object is o_2:object = identity o_1 is identity o_2 .

- _ is an instance of _ :: reference, class $\to$ truth-value (*total*) .

(5) null is an instance of c:class = false .

(6) o:object is an instance of c:class = c is in superclasses (class o) .

- allocate an object of _ :: yielder [of a class] $\to$
  action [storing | giving an object] [using current storage] .

(7) allocate an object of c:yielder [of a class] =
    | give c and
    | instantiate the field-type-bindings of c and
    | allocate an identity
    then give the object of (the given class#1,
        the given variable-bindings#2, the given identity#3) .

- instantiate _ :: yielder [of type-bindings] $\to$
  action [storing | giving variable-bindings] [using current storage] .

(8) instantiate t:yielder [of type-bindings] =
    | check (t is the empty-map) and then
    | give the empty-map
    or
    | | give t and
    | | choose a token [in the mapped-set of t]
    | then
    | | instantiate (the given type-bindings#1
    | | omitting the set of the given token#2) and
    | | give the given token#2 and
    | | allocate a variable initialised to the default-value of the type yielded by
    | | (the given type-bindings#1 at the given token#2)
    | then give the disjoint-union of (the given variable-bindings#1,
    | the map of the given token#2 to the given variable#3) .

3.7 Escapes

needs: **Values** .

introduces: return , return of _ , returned-value _ ,
exception , null-reference-exception ,
reason-for-escape ,
_ trap _ then _ .

- return = return of value$^?$.

- return of _ :: value$^?$ $\rightarrow$ return $(total,\ injective)$.
- returned-value _ :: return $\rightarrow$ value$^?$ $(total)$.

(1) returned-value (return of v:value$^?$) = v .

- exception = null-reference-exception **|** $\square$ $(individual)$.

- reason-for-escape = return **|** exception $(disjoint)$.

- _ trap _ then _ :: action [escaping], reason-for-escape, action $\rightarrow$ action .

(2) A_1:action trap ($R\leq$ reason-for-escape) then A_2:action =
    A_1 trap
    | | check (there is a given R) and then A_2
    | or
    | | check not (there is a given R) and then escape .