MODULE KOOL-TYPED-DYNAMIC-SYNTAX

SYNTAX *#Id* ::= `main`

KOOL addition: the "object" class

SYNTAX *#Id* ::= `object`
SYNTAX *Type* ::= `int`
| `bool`
| `string`
| `void`
| `array of` *Type*
| *Types* → *Type*

KOOL addition: types for objects

SYNTAX *Type* ::= `class` *#Id*
SYNTAX *Exps* ::= *List{Exp,","}*
SYNTAX *Types* ::= *List{Type,","}*
SYNTAX *Decl* ::= `var` *Exps* `;`

KOOL additions: function is renamed into method and classes are introduced.

SYNTAX *Decl* ::= `method` *#Id* (*Exps*) : *Type* *Stmt*
| `method` *#Id* (*Exps*) *Stmt*
| `class` *#Id* { *Stmts* }
| `class` *#Id* `extends` *#Id* { *Stmts* }
SYNTAX *Exp* ::= *#Int*
| *#Bool*
| *#String*
| *#Id*
| ++ *Exp*
| *Exp* + *Exp* [strict]
| *Exp* - *Exp* [strict]
| *Exp* $*$ *Exp* [strict]
| *Exp* / *Exp* [strict]
| *Exp* % *Exp* [strict]
| - *Exp* [strict]
| *Exp* < *Exp* [strict]
| *Exp* <= *Exp* [strict]
| *Exp* > *Exp* [strict]
| *Exp* >= *Exp* [strict]
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| *Exp* `and` *Exp* [strict]
| *Exp* `or` *Exp* [strict]
| `not` *Exp* [strict]
| *Exp* [*Exps*] [strict]
| `sizeOf(` *Exp* `)` [strict]
| *Exp* (*Exps*) [strict(2)]
| `read()`
| *Exp* = *Exp* [strict(2)]
| *Exp* : *Type*

Kool additions: new, this, super, object space, instanceOf, cast

SYNTAX *Exp* ::= `new` *#Id* (*Exps*) [strict(2)]
| `this`
| `super`
| *Exp* . *#Id*
| *Exp* `instanceOf` *#Id* [strict(1)]
| `cast` *Exp* `to` *#Id* [strict(1)]
SYNTAX *Stmt* ::= {}
| { *Stmts* }
| *Exp* ; [strict]
| `if` *Exp* `then` *Stmt* `else` *Stmt* [strict(1)]
| `if` *Exp* `then` *Stmt*
| `while` *Exp* `do` *Stmt*
| `for` *#Id* = *Exp* `to` *Exp* `do` *Stmt*
| `return` *Exp* ; [strict]
| `return;`
| `print(` *Exps* `);` [strict]
| `try` *Stmt* `catch(` *#Id* `)` *Stmt*
| `throw` *Exp* ; [strict]
| `spawn` *Stmt*
| `acquire` *Exp* ; [strict]
| `release` *Exp* ; [strict]
| `rendezvous` *Exp* ; [strict]
SYNTAX *Stmts* ::= *Decl*
| *Stmt*
| *Stmts* *Stmts* [seqstrict]

MACRO `if` E `then` S = `if` E `then` S `else {}`

MACRO `for` X = E_1 `to` E_2 `do` S = { `var` X : `int` = E_1 ; `while` X <= E_2 `do` { S X = X + 1 ; } }

MACRO `var` E_1 , E_2 , Es ; = `var` E_1 ; `var` E_2 , Es ;

MACRO `class` C { Ss } = `class` C `extends object` { Ss }

MACRO `method` X (Es) S = `method` X (Es): `void` S

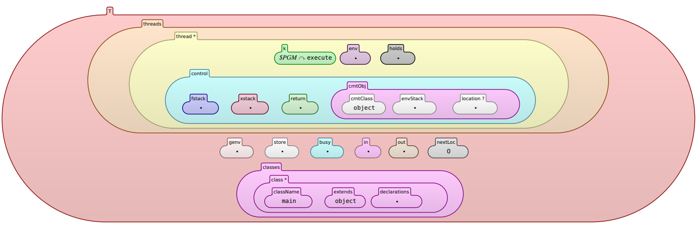
END MODULE

MODULE KOOL-TYPED-DYNAMIC
IMPORTS KOOL-TYPED-DYNAMIC-SYNTAX
SYNTAX *Vals* ::= *List{Val,","}*
SYNTAX *Val* ::= *#Int*
| *#Bool*
| *#String*
| `array(` *Type* , *#Nat* , *#Nat* `)`

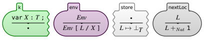
KOOL additions: lambda now takes current object as first argument.

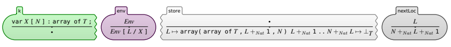
Also introducing the object value "obj".

SYNTAX *Val* ::= `lambda(` *K* , *Exps* , *Stmt* , *Type* `)`
| `obj(` *Bag* `)`
| `objRef(` *#Id* , *#Nat* `)`
SYNTAX *Exp* ::= *Val*
SYNTAX *KResult* ::= *Val*
CONFIGURATION:


T
threads
thread *
k
$PGM ↷ execute
env
•
holds
•
control
fstack
•
xstack
•
return
•
crntObj
crntClass
object
envStack
•
location ?
•
genv
•
store
•
busy
•
in
•
out
•
nextLoc
0
classes
class *
className
main
extends
object
declarations
•

SYNTAX K ::= $\bot_{Type}$

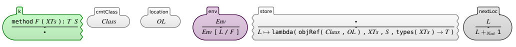
RULE

k
var X : T ;
•
env
Env
Env [L / X]
store
•
L ↦ ⊥T
nextLoc
L
L +Nat 1

RULE

k
var X [N] : array of T ;
•
env
Env
Env [L / X]
store
•
L ↦ array(array of T , L +Nat 1 , N) L +Nat 1 . . N +Nat L ↦ ⊥T
nextLoc
L
N +Nat L +Nat 1

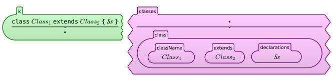
CONTEXT: `var` – [☐] : – ;

CONTEXT: `var` – [☐] : – ;

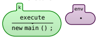
SYNTAX *#Id* ::= `$1`
| `$2`
RULE `var` X [N_1 , N_2 , Vs] : `array of array of` T ; ⇒ `var` X [N_1] : `array of array of` T ; { `var $1 : array of array of` T = X ; `for $2 = 0 to` N_1 `- 1 do` { `var` X [N_2 , Vs] : `array of` T ; `$1 [ $2 ] =` X ; } } [structural]

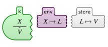
RULE

k
method F (XTs) : T S
•
crntClass
Class
location
OL
env
Env
Env [L / F]
store
•
L ↦ lambda(objRef(Class , OL) , XTs , S , types(XTs) → T)
nextLoc
L
L +Nat 1

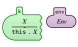
Class declaration

RULE

k
class Class1 extends Class2 { Ss }
•
classes
•
class
className
Class1
extends
Class2
declarations
Ss

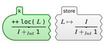
SYNTAX K ::= `execute`

RULE

k
execute
new main () ;
env
•

RULE
 [transition]
k
X
V
env
X ↦ L
store
L ↦ V

RULE
 when $\neg_{Bool}$ X `in keys` Env
k
X
this . X
env
Env

CONTEXT: ++ $\dfrac{☐}{\texttt{l-value(}\ ☐\ \texttt{)}}$

RULE
 [transition]
k
++ loc(L)
I +Int 1
store
L ↦
I
I +Int 1

RULE I_1 + I_2 ⇒ $I_1 +_{Int} I_2$

RULE Str_1 + Str_2 ⇒ $Str_1 +_{String} Str_2$

RULE I_1 - I_2 ⇒ $I_1 -_{Int} I_2$

RULE I_1 $*$ I_2 ⇒ $I_1 *_{Int} I_2$

RULE I_1 / I_2 ⇒ $I_1 \div_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE I_1 % I_2 ⇒ I_1 %$_{Int}$ I_2 when $I_2 \neq_{Bool} 0$

RULE - I ⇒ $-_{Int} I$

RULE I_1 < I_2 ⇒ $I_1 <_{Int} I_2$

RULE I_1 <= I_2 ⇒ $I_1 \leq_{Int} I_2$

RULE I_1 > I_2 ⇒ $I_1 >_{Int} I_2$

RULE I_1 >= I_2 ⇒ $I_1 \geq_{Int} I_2$

RULE V_1 == V_2 ⇒ $V_1 =_{Bool} V_2$

RULE V_1 != V_2 ⇒ $V_1 \neq_{Bool} V_2$

RULE B_1 `and` B_2 ⇒ $B_1 \wedge_{Bool} B_2$

RULE B_1 `or` B_2 ⇒ $B_1 \vee_{Bool} B_2$

RULE `not` B ⇒ $\neg_{Bool} B$

Check array bounds, as part of the dynamic typing policy.

RULE V [N_1 , N_2 , Vs] ⇒ V [N_1] [N_2 , Vs] [structural]

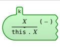
RULE `array(` – , L , M `)` [N] ⇒ `lookup(` $N +_{Int} L$ `)` when $N <_{Nat} M \wedge_{Bool} N \geq_{Nat} 0$ [structural]

RULE `sizeOf( array(` – , – , N `) )` ⇒ N

Method call

SKOOL has dynamic method dispatch, like most OO languages. There are several cases to treat, though.

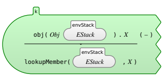
Case 1: method is called directly, not through an object or super. Then "this" is assumed

RULE


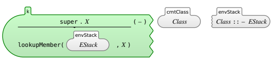
Case 2: method is called through an object or super. There are two subcases:

Case 2.1: called through a proper object (not super), so dynamic dispatch. Then evaluate the object espression and lookup method from top of its environment.

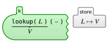
CONTEXT: ☐ . – (–) when ☐ $\neq_{Bool}$ `super`

RULE

k
obj(Obj
envStack
EStack
) . X
(–)
lookupMember(
envStack
EStack
, X)

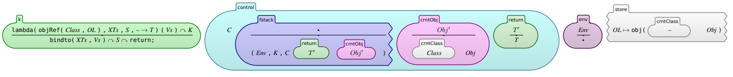
Case 2.2: called through super, so static dispatch from parent class

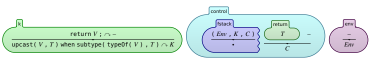
RULE

k
super . X
(–)
lookupMember(
envStack
EStack
, X)
crntClass
Class
envStack
Class : : – EStack

Eventually, each of the rules above produces a lookup(L) instead of the method. When that happens, just lookup the function value at location L.

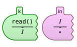
RULE

k
lookup(L) (–)
V
store
L ↦ V

SYNTAX *ListItem* ::= (*Map* , *K* , *Bag*)

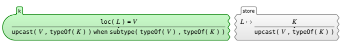
RULE

k
lambda(objRef(Class , OL) , XTs , S , – → T) (Vs) ↷ K
bindto(XTs , Vs) ↷ S ↷ return;
control
C
fstack
•
(Env , K , C
return
T′
crntObj
Obj′
)
crntObj
Obj′
crntClass
Class
Obj
return
T′
T
env
Env
•
store
OL ↦ obj(
crntClass
–
Obj)

RULE

k
return V ; ↷ –
upcast(V , T) when subtype(typeOf(V) , T) ↷ K
control
fstack
(Env , K , C)
•
return
T
–
C
env
–
Env

SYNTAX *Val* ::= `nothing`
RULE `return;` ⇒ `return nothing ;` [structural]

RULE
 [transition]

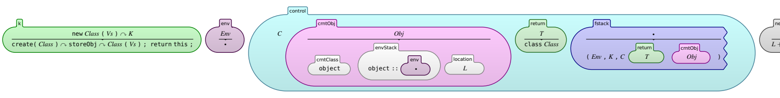
CONTEXT: $\dfrac{☐}{\texttt{l-value(}\ ☐\ \texttt{)}}$ = –

RULE
 [transition]
k
loc(L) = V
upcast(V , typeOf(K)) when subtype(typeOf(V) , typeOf(K))
store
L ↦
K
upcast(V , typeOf(K))

New

The semantics of new consists of two actions: memory allocation for the new object and the execution of the corresponding constructor.

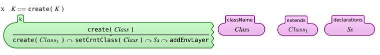
The current environment and object are stored on the stack and recovered after new, because new destroys them.

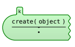
RULE

k
new Class (Vs) ↷ K
create(Class) ↷ storeObj ↷ Class (Vs) ; return this ;
env
Env
•
control
C
crntObj
Obj
crntClass
object
envStack
object : :
env
•
location
L
return
T
class Class
fstack
•
(Env , K , C
return
T
crntObj
Obj
)
nextLoc
L
L +Nat 1

Create a new object (memory allocation)

A memory object representation is a layered structure: for each class on the path from the instance class to the root of the hierarchy there is a layer including the memory allocated for the members (both fields and methods) of that class. Even though "create" is an auxilliary operation, it is used only here.

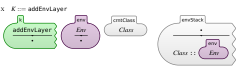
SYNTAX K ::= `create(` K `)`

RULE

k
create(Class)
create(Class1) ↷ setCrntClass(Class) ↷ Ss ↷ addEnvLayer
className
Class
extends
Class1
declarations
Ss

RULE

k
create(object)
•

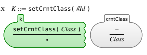
Add a new tagged environment layer to the current object.

SYNTAX K ::= `addEnvLayer`

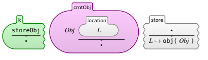
RULE

k
addEnvLayer
•
env
Env
•
crntClass
Class
envStack
•
Class : :
env
Env

Set the current class of the current object. This is necessary to be done at each layer, because the current object is enclosed as part of the function values (see the semantics of method declarations above).

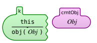
SYNTAX K ::= `setCrntClass(` *#Id* `)`

RULE

k
setCrntClass(Class)
•
crntClass
–
Class

SYNTAX K ::= `storeObj`

RULE

k
storeObj
•
crntObj
Obj
location
L
•
store
•
L ↦ obj(Obj)

Self reference via "this"

RULE

k
this
obj(Obj)
crntObj
Obj

Self reference via "this"

RULE $\overset{\text{k}}{\dfrac{\texttt{this}}{\texttt{obj(}\ Obj\ \texttt{)}}}\quad \overset{\text{crntObj}}{Obj}$

Object member access

There are two cases, like for method dispatch:

Case 1: member access through a proper object (not super); Then evaluate the object expression and lookup member starting with current class (static scoping).

CONTEXT: $\square\ .\ \_$ when $\square \neq_{Bool}$ `super`

RULE $\overset{\text{k}}{\dfrac{\texttt{obj(}\ \overset{\text{crntClass}}{Class}\ \ \overset{\text{envStack}}{Class :: EnvC\ \ EStack}\ \texttt{)}\ .\ X}{\texttt{lookupMember(}\ \overset{\text{envStack}}{Class :: EnvC\ \ EStack}\ ,\ X\ \texttt{)}}}$

Case 2: member access through super

RULE $\overset{\text{k}}{\dfrac{\texttt{super}}{\texttt{obj(}\ Obj\ \ \overset{\text{crntClass}}{Class_2}\ \texttt{)}}\ .\ X}\quad \overset{\text{crntObj}}{Obj\ \ \overset{\text{crntClass}}{Class_1}}\quad \overset{\text{className}}{Class_1}\quad \overset{\text{extends}}{Class_2}$

RULE $\texttt{obj(}\ \_\ \ \overset{\text{envStack}}{Class :: \_}\ \texttt{)}\ \texttt{instanceOf}\ Class \Rightarrow \texttt{true}$

RULE $\texttt{obj(}\ \_\ \ \overset{\text{envStack}}{\dfrac{Class_1 :: \_}{\bullet}}\ \texttt{)}\ \texttt{instanceOf}\ Class_2$ when $Class_1 \neq_{Bool} Class_2$

RULE $\texttt{obj(}\ \_\ \ \overset{\text{envStack}}{\bullet}\ \texttt{)}\ \texttt{instanceOf}\ Class \Rightarrow \texttt{false}$

RULE $\overset{\text{k}}{\dfrac{\texttt{cast obj(}\ \overset{\text{crntClass}}{\_}\ \ Obj\ \texttt{) to}\ Class}{\texttt{obj(}\ \overset{\text{crntClass}}{Class}\ \ Obj\ \texttt{) when obj(}\ Obj\ \texttt{) instanceOf}\ Class}}$

RULE $\texttt{\{\}} \Rightarrow \bullet$ [structural]

RULE $\overset{\text{k}}{\dfrac{\texttt{\{}\ Ss\ \texttt{\}}}{Ss \curvearrowright \texttt{env(}\ Env\ \texttt{)}}}\quad \overset{\text{env}}{Env}$ [structural]

RULE $S_1\ \ S_2 \Rightarrow S_1 \curvearrowright S_2$

RULE $V\ \texttt{;} \Rightarrow \bullet$

RULE $\texttt{if true then}\ S\ \texttt{else}\ \_ \Rightarrow S$

RULE $\texttt{if false then}\ \_\ \texttt{else}\ S \Rightarrow S$

RULE $\overset{\text{k}}{\dfrac{\texttt{while}\ E\ \texttt{do}\ S}{\texttt{if}\ E\ \texttt{then \{}\ S\ \ \texttt{while}\ E\ \texttt{do}\ S\ \texttt{\} else \{\}}}}$ [structural]

We only allow printing integers and strings

RULE $\overset{\text{k}}{\texttt{print(}\ \dfrac{V\ ,\ Vs}{Vs}\ \texttt{);}}\quad \overset{\text{out}}{\dfrac{\bullet}{V}}$ when $\texttt{typeOf(}\ V\ \texttt{)} =_{Bool} \texttt{int} \vee_{Bool} \texttt{typeOf(}\ V\ \texttt{)} =_{Bool} \texttt{string}$ [transition]

RULE $\texttt{print();} \Rightarrow \bullet$ [structural]

SYNTAX $ListItem ::= (\ Exp\ ,\ Stmt\ ,\ K\ ,\ Map\ ,\ Bag\ )$
SYNTAX $K ::= \texttt{popx}$

RULE $\overset{\text{k}}{\dfrac{\texttt{try}\ S_1\ \texttt{catch(}\ X\ \texttt{)}\ S_2 \curvearrowright K}{S_1 \curvearrowright \texttt{popx}}}\quad \overset{\text{control}}{\overset{\text{xstack}}{\dfrac{\bullet}{(\ X\ ,\ S_2\ ,\ K\ ,\ Env\ ,\ C\ )}}\ \ C}\quad \overset{\text{env}}{Env}$

RULE $\overset{\text{k}}{\dfrac{\texttt{popx}}{\bullet}}\quad \overset{\text{xstack}}{\dfrac{\_}{\bullet}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{throw}\ V\ \texttt{;} \curvearrowright \_}{\texttt{\{ var}\ X : \texttt{int} = V\ \texttt{;}\ \ S_2\ \texttt{\}} \curvearrowright K}}\quad \overset{\text{control}}{\overset{\text{xstack}}{\dfrac{(\ X\ ,\ S_2\ ,\ K\ ,\ Env\ ,\ C\ )}{\bullet}}\ \ \dfrac{\_}{C}}\quad \overset{\text{env}}{\dfrac{\_}{Env}}$

RULE $\overset{\text{thread}}{\overset{\text{k}}{\dfrac{\texttt{spawn}\ S}{\bullet}}\ \ \overset{\text{env}}{Env}}\quad \dfrac{\bullet}{\overset{\text{thread}}{\overset{\text{k}}{S}\ \ \overset{\text{env}}{Env}}}$

RULE $\dfrac{\overset{\text{thread}}{\overset{\text{k}}{\bullet}\ \ \overset{\text{holds}}{H}}}{\bullet}\quad \overset{\text{busy}}{\dfrac{Busy}{Busy\ -_{Set}\ \texttt{keys}\ H}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{acquire}\ V\ \texttt{;}}{\bullet}}\quad \overset{\text{holds}}{\dfrac{\bullet}{V \mapsto 0}}\quad \overset{\text{busy}}{Busy\ \ \dfrac{\bullet}{V}}$ when $\neg_{Bool}\ V\ \texttt{in}\ Busy$ [transition]

RULE $\overset{\text{k}}{\dfrac{\texttt{acquire}\ V\ \texttt{;}}{\bullet}}\quad \overset{\text{holds}}{V \mapsto \dfrac{N}{N +_{Nat} 1}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{release}\ V\ \texttt{;}}{\bullet}}\quad \overset{\text{holds}}{V \mapsto \dfrac{N}{N -_{Int} 1}}$ when $N\ >_{Nat} 0$

RULE $\overset{\text{k}}{\dfrac{\texttt{release}\ V\ \texttt{;}}{\bullet}}\quad \overset{\text{holds}}{\dfrac{V \mapsto 0}{\bullet}}\quad \overset{\text{busy}}{\dfrac{V}{\bullet}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{rendezvous}\ V\ \texttt{;}}{\bullet}}\quad \overset{\text{k}}{\dfrac{\texttt{rendezvous}\ V\ \texttt{;}}{\bullet}}$ [transition]

Auxiliary operations

`l-value`

SYNTAX $Exp ::= \texttt{l-value(}\ K\ \texttt{)}$
SYNTAX $Val ::= \texttt{loc(}\ \#Nat\ \texttt{)}$

RULE $\overset{\text{k}}{\dfrac{\texttt{l-value(}\ X\ \texttt{)}}{\texttt{loc(}\ L\ \texttt{)}}}\quad \overset{\text{env}}{X \mapsto L}$

KOOL addiiton: Implicit member access

RULE $\overset{\text{k}}{\texttt{l-value(}\ \dfrac{X}{\texttt{this}\ .\ X}\ \texttt{)}}\quad \overset{\text{env}}{Env}$ when $\neg_{Bool}\ X\ \texttt{in keys}\ Env$

CONTEXT: $\texttt{l-value(}\ \_\ [\ \square\ ]\ \texttt{)}$

CONTEXT: $\texttt{l-value(}\ \square\ [\ \_\ ]\ \texttt{)}$

Explicit member access First evaluate the object, then lookup its corresponding member staticaly. lookupMember is defined below; it evaluates to a lookup(L), also defined below, where L points to the actual member value. Note that object values are envStack cells, and that lookupMember takes such cells as first argument; these cells are also defined below.

CONTEXT: $\texttt{l-value(}\ \square\ .\ \_\ \texttt{)}$

RULE $\overset{\text{k}}{\dfrac{\texttt{l-value(obj(}\ \overset{\text{crntClass}}{Class}\ \ \overset{\text{envStack}}{Class :: EnvC\ \ EStack}\ \texttt{)}\ .\ X\ \texttt{)}}{\texttt{lookupMember(}\ \overset{\text{envStack}}{Class :: EnvC\ \ EStack}\ ,\ X\ \texttt{)}}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{l-value(lookup(}\ L\ \texttt{))}}{\texttt{loc(}\ L\ \texttt{)}}}$

`lookup`

SYNTAX $K ::= \texttt{lookup(}\ \#Nat\ \texttt{)}$

RULE $\overset{\text{k}}{\dfrac{\texttt{lookup(}\ L\ \texttt{)}}{V}}\quad \overset{\text{store}}{L \mapsto V}$ [transition]

`bindto` also checks the well-formedness of the function parameters

SYNTAX $K ::= \texttt{bindto(}\ Exps\ ,\ Vals\ \texttt{)}$

RULE $\overset{\text{k}}{\dfrac{\bullet}{X = V\ \texttt{;}} \curvearrowright \texttt{bindto(}\ \dfrac{X : T\ ,\ XTs}{XTs}\ ,\ \dfrac{V\ ,\ Vs}{Vs}\ \texttt{)}}\quad \overset{\text{env}}{\dfrac{Env}{Env\ [\ L\ /\ X\ ]}}\quad \overset{\text{store}}{\dfrac{\bullet}{L \mapsto \bot_T}}\quad \overset{\text{nextLoc}}{\dfrac{L}{L +_{Nat} 1}}$

RULE $\overset{\text{k}}{\dfrac{\texttt{bindto(}\ \bullet\ ,\ \bullet\ \texttt{)}}{\bullet}}$

`env`

SYNTAX $K ::= \texttt{env(}\ Map\ \texttt{)}$

RULE $\overset{\text{k}}{\dfrac{\texttt{env(}\ Env\ \texttt{)}}{\bullet}}\quad \overset{\text{env}}{\dfrac{\_}{Env}}$ [structural]

RULE $\dfrac{\texttt{env(}\ \_\ \texttt{)}}{\bullet} \curvearrowright \texttt{env(}\ \_\ \texttt{)}$ [structural]

`typeOf`

SYNTAX $Type ::= \texttt{typeOf(}\ K\ \texttt{)}$
RULE $\texttt{typeOf(}\ I\ \texttt{)} \Rightarrow \texttt{int}$ [structural]

RULE $\texttt{typeOf(}\ B\ \texttt{)} \Rightarrow \texttt{bool}$ [structural]

RULE $\texttt{typeOf(}\ \_\ \texttt{)} \Rightarrow \texttt{string}$ [structural]

RULE $\texttt{typeOf(array(}\ T\ ,\ \_\ ,\ \_\ \texttt{))} \Rightarrow T$ [structural]

RULE $\texttt{typeOf(lambda(}\ \_\ ,\ \_\ ,\ \_\ ,\ T\ \texttt{))} \Rightarrow T$ [structural]

RULE $\texttt{typeOf(}\ \bot_T\ \texttt{)} \Rightarrow T$ [structural]

RULE $\texttt{typeOf(nothing)} \Rightarrow \texttt{void}$ [structural]

RULE $\texttt{typeOf(obj(}\ \_\ \ \overset{\text{envStack}}{Class :: \_}\ \texttt{))} \Rightarrow \texttt{class}\ Class$ [structural]

SYNTAX $Types ::= \texttt{types(}\ Exps\ \texttt{)}$
RULE $\texttt{types()} \Rightarrow \bullet$ [structural]

RULE $\texttt{types(}\ X : T\ ,\ XTs\ \texttt{)} \Rightarrow T\ ,\ \texttt{types(}\ XTs\ \texttt{)}$ [structural]

Sequences of locations

SYNTAX $List\{K\} ::= \#Nat\ ..\ \#Nat$
RULE $N_1\ ..\ N_2 \Rightarrow \bullet$ when $N_1\ >_{Nat} N_2$ [structural]

RULE $N_1\ ..\ N_2 \Rightarrow N_1\ ,\ N_1 +_{Nat} 1\ ..\ N_2$ when $N_1\ \leq_{Nat} N_2$ [structural]

Conditional Pass

Allows the process to continue only if the expression evaluates to true. Useful for checking side conditions to typing rules.

SYNTAX $K ::= K\ \texttt{when}\ K$ [strict(2)]
RULE $K\ \texttt{when true} \Rightarrow K$

Subtyping constraints

The subclass relation introduces a subtyping relation.

SYNTAX $K ::= \texttt{subtype(}\ Types\ ,\ Types\ \texttt{)}$
RULE $\texttt{subtype(,)} \Rightarrow \texttt{true}$

RULE $\texttt{subtype(}\ T_1\ ,\ T_2\ ,\ Ts\ ,\ T'_1\ ,\ T'_2\ ,\ Ts'\ \texttt{)} \Rightarrow \texttt{subtype(}\ T_2\ ,\ Ts\ ,\ T'_2\ ,\ Ts'\ \texttt{) when subtype(}\ T_1\ ,\ T'_1\ \texttt{)}$

RULE $\texttt{subtype(}\ T\ ,\ T\ \texttt{)} \Rightarrow \texttt{true}$

RULE $\overset{\text{k}}{\dfrac{\texttt{subtype(class}\ Class_1\ \texttt{, class}\ Class\ \texttt{)}}{\texttt{subtype(class}\ Class_2\ \texttt{, class}\ Class\ \texttt{)}}}\quad \overset{\text{class}}{\overset{\text{className}}{Class_1}\ \ \overset{\text{extends}}{Class_2}}$ when $Class_1 \neq_{Bool} Class$

RULE $\overset{\text{k}}{\dfrac{\texttt{subtype(class}\ Class_1\ \texttt{, class}\ Class\ \texttt{)}}{\texttt{subtype(class}\ Class_2\ \texttt{, class}\ Class\ \texttt{)}}}\quad \overset{\text{className}}{Class_1}\quad \overset{\text{extends}}{Class_2}$ when $Class_1 \neq_{Bool} Class$

UpCasting

SYNTAX $Val ::= \texttt{upcast(}\ Val\ ,\ Type\ \texttt{)}$

RULE $\texttt{upcast(obj(}\ \overset{\text{crntClass}}{\_}\ \ Obj\ \texttt{) , class}\ Class\ \texttt{)} \Rightarrow \texttt{obj(}\ \overset{\text{crntClass}}{Class}\ \ Obj\ \texttt{)}$ [structural]

RULE $\texttt{upcast(}\ V\ ,\ T\ \texttt{)} \Rightarrow V$ when $\texttt{typeOf(}\ V\ \texttt{)} =_{Bool} T$ [structural]

SYNTAX $K ::= \#Id :: BagItem$
$\quad\mid \texttt{lookupMember(}\ BagItem\ ,\ \#Id\ \texttt{)}$

RULE $\texttt{lookupMember(}\ \overset{\text{envStack}}{\_ :: \overset{\text{env}}{X \mapsto L}}\ ,\ X\ \texttt{)} \Rightarrow \texttt{lookup(}\ L\ \texttt{)}$

RULE $\texttt{lookupMember(}\ \overset{\text{envStack}}{\dfrac{\_ :: \overset{\text{env}}{Env}}{\bullet}}\ ,\ X\ \texttt{)}$ when $\neg_{Bool}\ X\ \texttt{in keys}\ Env$ [structural]

END MODULE