MODULE KOOL-UNTYPED-SYNTAX

SYNTAX $\#Id$ ::= main

> KOOL addition: the "object" class.

SYNTAX $\#Id$ ::= object
SYNTAX Ids ::= $List\{\#Id, ","\}$
SYNTAX $Exps$ ::= $List\{Exp, ","\}$
SYNTAX $Decl$ ::= var $Exps$;

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

```
SYNTAX Decl ::= method #Id ( Ids ) 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]
```

> KOOL addition: unlike in SIMPLE, application is only strict(2)

```
SYNTAX Exp ::= Exp ( Exps ) [strict(2)]
             | read()
             | Exp = Exp [strict(2)]
             | 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
```

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

MACRO for X = E_1 to E_2 do S = { var X = 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 var X = E ; = var X ; X = E ;

> KOOL addition: all classes extend the "object" class.

MACRO class $Class$ { Ss } = class $Class$ extends object { Ss }

END MODULE

MODULE KOOL-UNTYPED

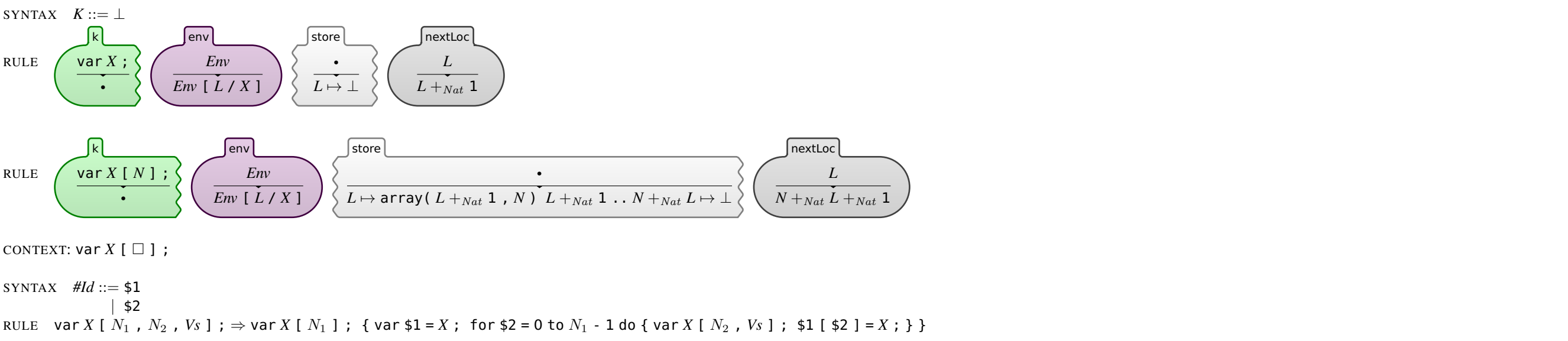
IMPORTS KOOL-UNTYPED-SYNTAX

SYNTAX $Vals$ ::= $List\{Val, ","\}$
SYNTAX Val ::= $\#Int$ | $\#Bool$ | $\#String$ | array($\#Nat$, $\#Nat$)

> KOOL additions: lambda now takes current object as first argument.
>
> Also introducing the object value "obj".

SYNTAX Val ::= lambda(K , Ids , $Stmt$) | obj(Bag) | objRef($\#Id$, $\#Nat$)
SYNTAX Exp ::= Val
SYNTAX $KResult$ ::= Val

CONFIGURATION:



SYNTAX K ::= ⊥

RULE k: var X ; ⇒ • | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto \bot$ | nextLoc: L ⇒ $L +_{Nat}$ 1

RULE k: var X [N] ; ⇒ • | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto$ array($L +_{Nat}$ 1 , N) $L +_{Nat}$ 1 .. $N +_{Nat} L \mapsto \bot$ | nextLoc: L ⇒ $N +_{Nat} L +_{Nat}$ 1

CONTEXT: var X [□] ;

SYNTAX $\#Id$ ::= $1 | $2

RULE var X [N_1 , N_2 , Vs] ; ⇒ var X [N_1] ; { var $1 = X ; for $2 = 0 to N_1 - 1 do { var X [N_2 , Vs] ; $1 [$2] = X ; } } [structural]

> **Method declaration**
>
> Like in SIMPLE, method names are added to the environment and bound to their code. This way, we can easily allow objects to pass and change their methods (see p23).

RULE k: method F (Xs) S ⇒ • | crntClass: $Class$ | location: OL | env: Env ⇒ Env [L / F] | store: • ⇒ $L \mapsto$ lambda(objRef($Class$, OL) , Xs , S) | nextLoc: L ⇒ $L +_{Nat}$ 1

> **Class declaration**

RULE k: class $Class_1$ extends $Class_2$ { Ss } ⇒ • | classes: • ⇒ class(className: $Class_1$ | extends: $Class_2$ | declarations: Ss)

SYNTAX K ::= execute

RULE k: execute ⇒ new main () ; | env: •

RULE k: X ⇒ V | env: $X \mapsto L$ | store: $L \mapsto V$ [transition]

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

CONTEXT: ++ □ ⇒ l-value(□)

RULE k: ++ loc(L) ⇒ $I +_{Int}$ 1 | store: $L \mapsto$ I ⇒ $I +_{Int}$ 1 [transition]

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 ⇒ $\neg_{Bool} B$

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

RULE array(L , –) [N] ⇒ lookup($N +_{Int} L$) [structural]

SYNTAX K ::= lookup($\#Nat$)

RULE k: lookup(L) ⇒ V | store: $L \mapsto V$ [transition]

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 k: X ⇒ this . X (–)

> 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: $Class$ :: $EnvC$ $EStack$

> 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: $L \mapsto V$

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

RULE k: lambda(objRef($Class$, OL) , Xs , S) (Vs) ↷ K ⇒ bindto(Xs , Vs) ↷ S ↷ return; | control: C fstack: • ⇒ (Env , K , C crntObj: Obj') | crntObj: Obj' ⇒ crntClass: $Class$ Obj | env: Env ⇒ • | store: $OL \mapsto$ obj(crntClass: – Obj)

RULE k: return V ; ↷ – ⇒ V ↷ K | control: fstack: (Env , K , C) ⇒ • ; – ⇒ C | env: – ⇒ Env

SYNTAX Val ::= nothing

RULE return; ⇒ return nothing ; [structural]

SYNTAX K ::= bindto(Ids , $Vals$)

RULE k: bindto(X , Xs ⇒ Xs , V , Vs ⇒ Vs) | env: Env ⇒ Env [L / X] | store: • ⇒ $L \mapsto V$ | nextLoc: L ⇒ $L +_{Nat}$ 1

RULE k: bindto(• , •) ⇒ •

RULE k: read() ⇒ I | in: I ⇒ • [transition]

CONTEXT: □ ⇒ l-value(□) = –

RULE k: loc(L) = V ⇒ V | store: $L \mapsto$ – ⇒ V [transition]

> **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 | fstack: • ⇒ (Env , K , C 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($Class_1$) ↷ setCrntClass($Class$) ↷ Ss ↷ addEnvLayer | className: $Class$ | extends: $Class_1$ | 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: • ⇒ $L \mapsto$ obj(Obj)

> Self reference via "this"

RULE k: this ⇒ obj(Obj) | 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: □ . – when □ $\neq_{Bool}$ super

RULE k: obj(crntClass: $Class$ envStack: $Class$ :: $EnvC$ $EStack$) . X ⇒ lookupMember(envStack: $Class$ :: $EnvC$ $EStack$, X)

> Case 2: member access through super

RULE k: super . X ⇒ obj(Obj crntClass: $Class_2$) . X | crntObj: Obj crntClass: $Class_1$ | className: $Class_1$ | extends: $Class_2$

> instanceOf

RULE obj(– envStack: $Class$ :: –) instanceOf $Class$ ⇒ true

RULE obj(– envStack: $Class_1$ :: – ⇒ •) instanceOf $Class_2$ when $Class_1 \neq_{Bool} Class_2$

RULE obj(– envStack: •) instanceOf $Class$ ⇒ false

> Unsafe casting

RULE k: cast obj(crntClass: – Obj) to $Class$ ⇒ obj(crntClass: $Class$ Obj)

RULE {} ⇒ • [structural]

RULE k: { Ss } ⇒ Ss ↷ env(Env) | env: Env [structural]

SYNTAX K ::= env(Map)

RULE k: env(Env) ⇒ • | env: – ⇒ Env [structural]

RULE env(–) ⇒ • ↷ env(–) [structural]

RULE S_1 S_2 ⇒ S_1 ↷ S_2

RULE V ; ⇒ •

RULE if true then S else – ⇒ S

RULE if false then – else S ⇒ S

RULE k: while E do S ⇒ if E then { S while E do S } else {} [structural]

RULE k: print(V , Vs ⇒ Vs); | out: • ⇒ V [transition]

RULE print(); ⇒ • [structural]

SYNTAX $ListItem$ ::= ($\#Id$, $Stmt$, K , Map , Bag)
SYNTAX K ::= popx

RULE k: try S_1 catch(X) S_2 ↷ K ⇒ S_1 ↷ popx | control: xstack: • ⇒ (X , S_2 , K , Env , C) C | env: Env

RULE k: popx ⇒ • | xstack: – ⇒ •

RULE k: throw V ; ↷ – ⇒ { var X = V ; S_2 } ↷ K | control: xstack: (X , S_2 , K , Env , C) ⇒ • ; – ⇒ C | env: – ⇒ Env

RULE thread(k: spawn S ⇒ • | env: Env) • ⇒ thread(k: S | env: Env)

RULE thread(k: • | holds: H) ⇒ • | busy: $Busy$ ⇒ $Busy$ $-_{Set}$ keys H

RULE k: acquire V ; ⇒ • | holds: • ⇒ $V \mapsto$ 0 | busy: $Busy$ • ⇒ V when $\neg_{Bool}$ V in $Busy$ [transition]

RULE k: acquire V ; ⇒ • | holds: $V \mapsto$ N ⇒ $N +_{Nat}$ 1

RULE k: release V ; ⇒ • | holds: $V \mapsto$ N ⇒ $N -_{Int}$ 1 when $N >_{Nat}$ 0

RULE k: release V ; ⇒ • | holds: $V \mapsto$ 0 ⇒ • | busy: V ⇒ •

RULE k: rendezvous V ; ⇒ • | k: rendezvous V ; ⇒ • [transition]

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

RULE k: l-value(X) ⇒ loc(L) | env: $X \mapsto L$

> KOOL addiiton: Implicit member access

RULE k: l-value(X ⇒ this . X) | env: Env when $\neg_{Bool}$ X in keys Env

CONTEXT: l-value(– [□])

CONTEXT: l-value(□ [–])

> **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: l-value(□ . –)

RULE k: l-value(obj(crntClass: $Class$ envStack: $Class$ :: $EnvC$ $EStack$) . X) ⇒ lookupMember(envStack: $Class$ :: $EnvC$ $EStack$, X)

RULE k: l-value(lookup(L)) ⇒ loc(L)

SYNTAX K ::= $\#Id$ :: $BagItem$ | lookupMember($BagItem$, $\#Id$)

RULE lookupMember(envStack: – :: env: $X \mapsto L$, X) ⇒ lookup(L)

RULE lookupMember(envStack: – :: env: Env ⇒ • , X) when $\neg_{Bool}$ X in keys Env [structural]

SYNTAX $List\{K\}$ ::= $\#Nat$ .. $\#Nat$

RULE N_1 .. N_2 ⇒ • when $N_1 >_{Nat} N_2$ [structural]

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

END MODULE