# KOOL — Untyped

Grigore Roşu and Traian Florin Şerbănuţă ({grosu,tserban2}@illinois.edu)
University of Illinois at Urbana-Champaign

## Abstract

This is the K semantic definition of the untyped KOOL language. KOOL is aimed at being a pedagogical and research language that captures the essence of the object-oriented programming paradigm. Its untyped variant discussed here is simpler than the typed one, ignoring several intricate aspects of types in the presence of objects. A program consists of a set of class declarations. Each class can extend at most one other class (KOOL is single-inheritance). A class can declare a set of fields and a set of methods, all public and called the class' *members*. Specifically, KOOL includes the following features:

- Class declarations, where a class may or may not explicitly extend another class. In case a class does not explicitly extend another class, then it is assumed that it extends the default top-most and empty (i.e., no members) class called `Object`. Each class is required to declare precisely one homonymous method, called its *constructor*. Each valid program should contain one class named `Main`, whose constructor, `Main()`, takes no arguments. The execution of a program consists of creating an object instance of class `Main` and invoking the constructor `Main()` on it, that is, of executing "`new Main();`".
- All features of SIMPLE (see `examples/simple/untyped`), i.e., multidimensional arrays, function (here called "method") abstractions with call-by-value parameter passing style and static scoping, blocks with locals, input/output, parametric exceptions, and concurrency via dynamic thread creation/termination and synchronization. The only change in the syntax of SIMPLE when imported in KOOL is the function declaration keyword, `function`, which is changed into `method`. The exact same desugaring macros from SIMPLE are also included in KOOL. We can think of KOOL's classes as embedding SIMPLE programs (extended with OO constructs, as discussed next).
- Object creation using the "`new C(e1,...,en)`" expression construct. An object instance of class `C` is first created and then the constructor `C(e1,...,en)` is implicitly called on that object. KOOL only allows (and requires) one constructor per class. The class constructor can be called either implicitly during a new object creation for the class, or explicitly. The superclass constructor is *not* implicitly invoked when a class constructor is invoked; if you want to invoke the superclass constructor from a subclass constructor then you have to do it explicitly.
- An expression construct `this`, which evaluates to the current object.
- An expression construct `super`, which is used (only) in combination with member lookup (see next) to refer to a superclass field or method.
- A member lookup expression construct `e.x`, where `e` is an expression (either an expression expected to evaluate to an object or the `super` construct) and `x` is a class member name, that is, a field or a method name.
- Expression constructs "`e instanceOf C`" and "`(C) e`", where `e` is an expression expected to evaluate to an object and `C` a class name. The former tells whether the class of `e` is a subclass of `C`, that is, whether `e` can be used as an instance of `C`, and the latter changes the class of `e` to `C`. These operations always succeed: the former returns a Boolean value, while the latter changes the current class of `e` to `C` regardless of whether it is safe to do so or not. The typed version of KOOL will check the safety of casting by ensuring that the instance class of the object is a subclass of `C`. In untyped KOOL we do not want to perform this check because we want to allow the programmer maximum of flexibility: if one always accesses only available members, then the program can execute successfully despite the potentially unsafe cast.

There are some specific aspects of KOOL that need to be discussed.

First, KOOL is higher-order, allowing function abstractions to be treated like any other values in the language. For example, if `m` is a method of object `e` then `e.m` evaluates to the corresponding function abstraction. The function abstraction is in fact a closure, because in addition to the method parameters and body it also encapsulates the object value (i.e., the environment of the object together with its current class—see below) that `e` evaluates to. This way, function abstractions can be invoked anywhere and have the capability to change the state of their object. For example, if `m` is a method of object `e` which increments a field `c` of `e` when invoked, and if `getm` is another method of `e` which simply returns `m` when invoked, then the double application `(e.getm())()` has the same effect as `e.m()`, that is, increments the counter `c` of `e`. Note that the higher-order nature of KOOL was not originally planned; it came as a natural consequence of evaluating methods to closures and we decided to keep it. If you do not like it then do not use it.

Second, since all the fields and methods are public in KOOL, and since they can be redeclared in subclasses, it is not immediately clear how to lookup the member `x` when we write `e.x` and `e` is different from `super`. We distinguish two cases, depending on whether `e.x` occurs in a method invocation context (i.e., `e.x(...)`) or in a field context. KOOL has dynamic method dispatch, so if `e.x` is invoked as a method then `x` will be searched for starting with the instance class of the object value to which `e` evaluates. If `e.x` occurs in a non-method-invocation context then `x` will be treated as a field although it may hold a method closure due to the higher-order nature of KOOL) and thus will be searched starting with the current class of the object value of `e` (which, because of `this` and casting, may be different from its instance class). In order to achieve the above, each object value will consist of a pair holding the current class of the object and an environment stack with one layer for each class in the object's instance class hierarchy.

Third, although KOOL is dynamic method dispatch, its capabilities described above are powerful enough to allow us to mimic static method dispatch. For example, suppose that you want to invoke method `m()` statically. Then all you need to do is to declare a local variable and bind it to `m`, for example "`var staticm = m;`", and then call `staticm()`. This works because `staticm` is first bound to the method closure that `m` evaluates to, and then looked up as any local variable when invoked. We only enable the dynamic method dispatch when we have an object member on an application position, e.g., `m()`.

In what follows, we limit our comments to the new, KOOL-specific aspects of the language. We refer the reader to the untyped SIMPLE language for documentation on the remaining features, because those were all borrowed from SIMPLE.

MODULE KOOL-UNTYPED-SYNTAX

## Syntax

The syntax of KOOL extends that of SIMPLE with object-oriented constructs. We removed from the K annotated syntax of SIMPLE two constructs, namely the one for function declarations (because we want to call them "methods" now) and the one for function application (because application is not strict in the first argument anymore—needs to initiate dynamic method dispatch). The additional syntax includes:

- First, we need a new dedicated identifier, `Object`, for the default top-most class.
- Second, we rename the `function` keyword of SIMPLE into `method`.
- Third, we add syntax for class declarations together with a macro making classes which extend nothing to extend `Object`.
- Fourth, we change the strictness attribute of application into `strict(2)`.
- Finally, we add syntax and corresponding strictness for the KOOL object-oriented constructs.

SYNTAX *Id* ::= *Token*{"theObject"} | *Token*{"theMain"}

SYNTAX *Decl* ::= var *Exps* ;
| method *Id*(*Ids*) *Block*
| class *Id* *Block*
| class *Id* extends *Id* *Block*

SYNTAX *Exp* ::= *Int*
| *Bool*
| *String*
| *Id*
| this
| super
| (*Exp*) [bracket]
| ++ *Exp*
| *Exp* instanceOf *Id* [strict(1)]
| (*Id*) *Exp* [strict(2)]
| new *Id*(*Exps*) [strict(2)]
| *Exp* . *Id*
| *Exp*[*Exps*] [strict, klabel(`_[_]`)]
| *Exp*(*Exps*) [strict(2)]
| - *Exp* [strict]
| sizeOf (*Exp*) [strict]
| read ()
| *Exp* * *Exp* [strict]
| *Exp* / *Exp* [strict]
| *Exp* % *Exp* [strict]
| *Exp* + *Exp* [strict]
| *Exp* - *Exp* [strict]
| *Exp* < *Exp* [strict]
| *Exp* <= *Exp* [strict]
| *Exp* > *Exp* [strict]
| *Exp* >= *Exp* [strict]
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| ! *Exp* [strict]
| *Exp* && *Exp* [strict(1)]
| *Exp* || *Exp* [strict(1)]
| spawn *Block*
| *Exp* = *Exp* [strict(2)]

SYNTAX *Ids* ::= *List*{*Id*,","}

SYNTAX *Exps* ::= *List*{*Exp*,","} [strict]

SYNTAX *Block* ::= {}
| {*Stmts*}

SYNTAX *Stmt* ::= *Decl*
| *Block*
| *Exp* ; [strict]
| if (*Exp*)*Block* else *Block* [avoid, strict(1)]
| if (*Exp*)*Block*
| while (*Exp*)*Block*
| for (*Stmts* *Exp* ; *Exp*)*Block*
| return *Exp* ; [strict]
| return ;
| print (*Exps*) ; [strict]
| try *Block* catch (*Id*)*Block*
| throw *Exp* ; [strict]
| join *Exp* ; [strict]
| acquire *Exp* ; [strict]
| release *Exp* ; [strict]
| rendezvous *Exp* ; [strict]

SYNTAX *Stmts* ::= *Stmt*
| *Stmts* *Stmts*

Old desugaring rules, from SIMPLE

$$\text{RULE}\quad \frac{\texttt{if }(E)S}{\texttt{if }(E)S\ \texttt{else }\{\}}$$ [macro]

$$\text{RULE}\quad \frac{\texttt{for}(Start\ Cond\ ;\ Step)\{S\}}{\{Start\ \texttt{while}\,(Cond)\{S\ Step\ ;\}\}}$$ [macro]

$$\text{RULE}\quad \frac{\texttt{var } E1\ ,\ E2\ ,\ Es\ ;}{\texttt{var } E1\ ;\ \texttt{var } E2\ ,\ Es\ ;}$$ [macro]

$$\text{RULE}\quad \frac{\texttt{var } X = E\ ;}{\texttt{var } X\ ;\ X = E\ ;}$$ [macro]

New desugaring rule

$$\text{RULE}\quad \frac{\texttt{class } C\ S}{\texttt{class } C \texttt{ extends theObject } S}$$

END MODULE

## Semantics

We first discuss the new configuration of KOOL, which extends that of SIMPLE. Then we include the semantics of the constructs borrowed from SIMPLE unchanged; we refrain from discussing those, because they were already discussed in the K definition of SIMPLE. Then we discuss changes in SIMPLE's semantics needed for the more general setting of KOOL (for example for thread spawning, assignment, etc.). Finally, we discuss in detail the semantics of the additional KOOL constructs.

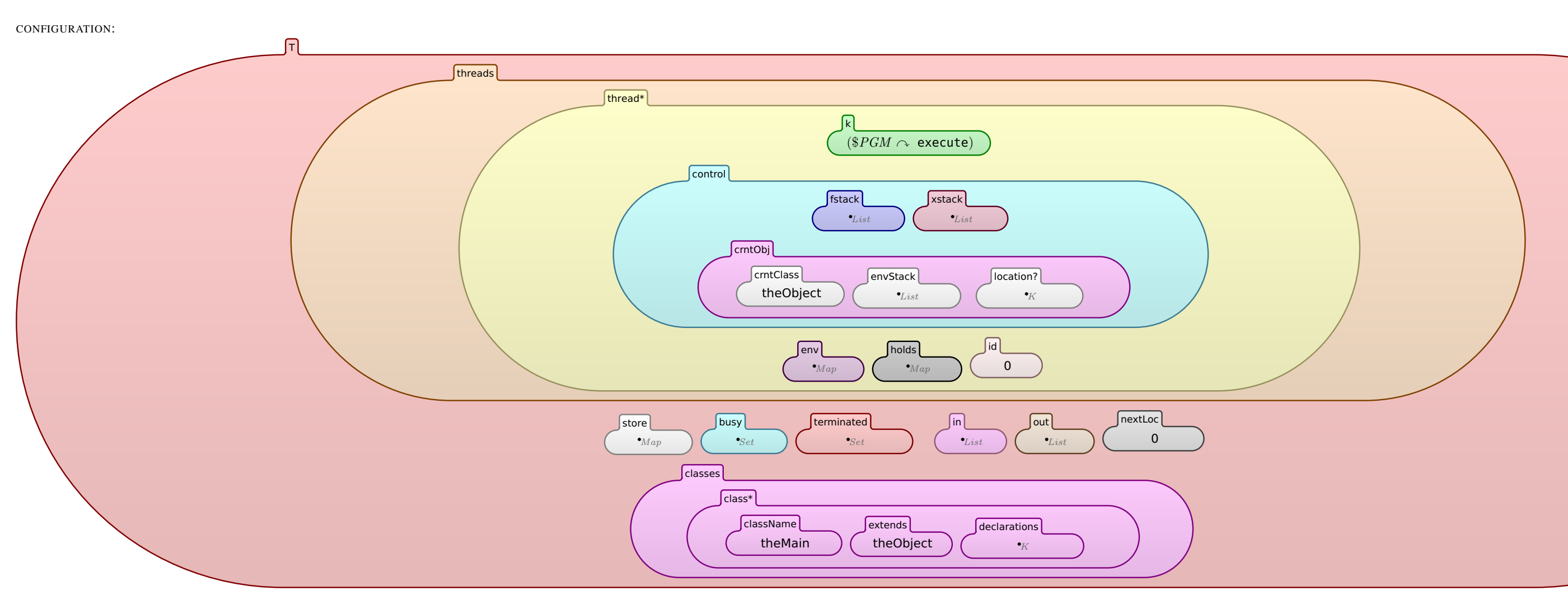
MODULE KOOL-UNTYPED

### Configuration

KOOL removes one cell and adds two nested cells to the configuration of SIMPLE. The cell which is removed is the one holding the global environment, because a KOOL program consists of a set of classes only, with no global declarations. In fact, since informally speaking each KOOL class now includes a SIMPLE program, it is safe to say that the global variables in SIMPLE became class fields in KOOL. Let us now discuss the new cells that are added to the configuration of SIMPLE.

- The cell crntObj holds data pertaining to the current object, that is, the object environment in which the code in cell k executes: crntClass holds the current class (which can change as methods of the current object are invoked); envStack holds the stack of environments as a list, each layer corresponding to one class in the objects' instance class hierarchy; location, which is optional, holds the location in the store where the current object is or has to be located (this is useful both for method closures and for semantics of object creation).
- The cell classes holds all the declared classes, each class being held in its own class cell which contains a name (className), a parent (extends), and the actual member declarations (declarations).

CONFIGURATION:



### Unchanged Semantics from untyped SIMPLE

The semantics below is taken over from SIMPLE unchanged. The semantics of function declaration and invocation, including the use of the special lambda abstraction value, needs to change in order to account for the fact that methods are now invoked into their object's environment. The semantics of function return actually stays unchanged. Also, the semantics of program initialization is different: now we have to create an instance of the Main class which also calls the constructor Main(), while in SIMPLE we only had to invoke the function Main(). Finally, the semantics of thread spawning needs to change, too: the parent thread needs to also share its object environment with the spawned thread (in addition to its local environment, like in SIMPLE). This is needed in order for be able to spawn method invocations under dynamic method dispatch; for example, "`spawn { run(); }`" will need to look up the method `run()` in the newly created thread, operation which will most likely fail unless the child thread sees the object environment of the parent thread. Note that the `spawn` statement of KOOL is more permissive than the threads of Java. In fact, the latter can be implemented in terms of our `spawn`—see the program `threads.kool` for a sketch.

Below is a subset of the values of SIMPLE, which are also values of KOOL. We will add other values later in the semantics, such as object and method closures.

SYNTAX *Val* ::= *Int*
| *Bool*
| *String*
| array (*Int*, *Int*)

SYNTAX *Vals* ::= *List*{*Val*,","}

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

The semantics below are taken verbatim from the untyped SIMPLE definition.

SYNTAX *KItem* ::= ⊥

RULE $\langle \frac{\texttt{var } X\ ;}{\cdot} \cdots\rangle_k\ \langle \frac{Env}{Env[L/X]}\rangle_{env}\ \langle \cdots \frac{\cdot}{L \mapsto \bot} \cdots\rangle_{store}\ \langle \frac{L}{L +_{Int} 1}\rangle_{nextLoc}$

CONTEXT var −[□] ;

RULE $\langle \frac{\texttt{var } X[N]\ ;}{\cdot} \cdots\rangle_k\ \langle \frac{Env}{Env[L/X]}\rangle_{env}\ \langle \cdots \frac{\cdot}{L \mapsto \texttt{array}(L +_{Int} 1, N)\ (L +_{Int} 1) \cdots (L +_{Int} N) \mapsto \bot} \cdots\rangle_{store}\ \langle \frac{L}{L +_{Int} 1 +_{Int} N}\rangle_{nextLoc}$ requires $N \geq_{Int} 0$

SYNTAX *Id* ::= *Token*{"$1"} | *Token*{"$2"}

RULE $\frac{\texttt{var } X[N1, N2, Vs]\ ;}{\texttt{var } X[N1]\ ;\ \{\ \texttt{var \$1 = X}\ ;\ \texttt{for}(\ \texttt{var \$2 = 0}\ ;\ \texttt{\$2 <= N1 - 1}\ ;\ \texttt{++\$2})\{\ \texttt{var } X[N2, Vs]\ ;\ \texttt{\$1[\$2] = } X\ ;\ \}\ \}}$ [structural]

RULE $\langle \frac{X}{V} \cdots\rangle_k\ \langle \cdots X \mapsto L \cdots\rangle_{env}\ \langle \cdots L \mapsto V \cdots\rangle_{store}$ [lookup]

CONTEXT ++ □ lvalue(□)

RULE $\langle \frac{\texttt{++ loc}(L)}{I +_{Int} 1} \cdots\rangle_k\ \langle \cdots L \mapsto \frac{I}{I +_{Int} 1} \cdots\rangle_{store}$ [increment]

RULE $\frac{I1 + I2}{I1 +_{Int} I2}$

RULE $\frac{Str1 + Str2}{Str1 +_{String} Str2}$

RULE $\frac{I1 - I2}{I1 -_{Int} I2}$

RULE $\frac{I1 * I2}{I1 *_{Int} I2}$

RULE $\frac{I1\ /\ I2}{I1 \div_{Int} I2}$ requires $I2 \neq_{K} 0$

RULE $\frac{I1\ \%\ I2}{I1 \%_{Int} I2}$ requires $I2 \neq_{K} 0$

RULE $\frac{-\ I}{0 -_{Int} I}$

RULE $\frac{I1 < I2}{I1 <_{Int} I2}$

RULE $\frac{I1 <= I2}{I1 \leq_{Int} I2}$

RULE $\frac{I1 > I2}{I1 >_{Int} I2}$

RULE $\frac{I1 >= I2}{I1 \geq_{Int} I2}$

RULE $\frac{V1 == V2}{V1 =_{K} V2}$

RULE $\frac{V1\ !=\ V2}{V1 \neq_{K} V2}$

RULE $\frac{!\ T}{\neg_{Bool}(T)}$

RULE $\frac{\texttt{true}\ \&\&\ E}{E}$

RULE $\frac{\texttt{false}\ \&\&\ -}{\texttt{false}}$

RULE $\frac{\texttt{true}\ ||\ -}{\texttt{true}}$

RULE $\frac{\texttt{false}\ ||\ E}{E}$

RULE $\frac{V[N1, N2, Vs]}{V[N1][N2, Vs]}$ [structural]

RULE $\frac{\texttt{array}(L, -)[N]}{\texttt{lookup}(L +_{Int} N)}$ [structural]

RULE $\frac{\texttt{lvalue}(V[N1, N2, Vs])}{\texttt{lvalue}(V[N1][N2, Vs])}$ [structural]

RULE $\texttt{lvalue}(\frac{\texttt{lookup}(L)\ [\ N\ ]}{\ }) \cdots$ — $\frac{\texttt{lvalue}(\texttt{array}(L, -)[N])}{\texttt{lookup}(L +_{Int} N)}$ [structural]

RULE $\frac{\texttt{sizeOf}(\texttt{array}(-, N))}{N}$

The semantics of function application needs to change into dynamic method dispatch invocation, which is defined shortly. However, interestingly, the semantics of return stays unchanged.

RULE $\langle \frac{\texttt{return}(V)\ ;}{V} \curvearrowright \frac{-}{K}\rangle_k\ \langle \frac{(Env, K, C)}{\cdot} \cdots\rangle_{fstack}\ \langle \frac{-}{C}\rangle_{crntObj}\ \langle \frac{-}{Env}\rangle_{env}$

SYNTAX *Val* ::= nothing

RULE $\frac{\texttt{return}\ ;}{\texttt{return nothing}\ ;}$ [macro]

RULE $\langle \frac{\texttt{read}()}{I} \cdots\rangle_k\ \langle \frac{I}{\cdot} \cdots\rangle_{in}$ [read]

CONTEXT $\frac{□}{\texttt{lvalue}(□)} = -$

RULE $\langle \frac{\texttt{loc}(L) = V}{V} \cdots\rangle_k\ \langle \cdots L \mapsto \frac{-}{V} \cdots\rangle_{store}$ [assignment]

RULE $\frac{\{\}}{\cdot}$ [structural]

RULE $\langle \frac{\{S\}}{S \curvearrowright \texttt{env}(Env)} \cdots\rangle_k\ \langle Env \rangle_{env}$ [structural]

RULE $\frac{S1\ S2}{S1 \curvearrowright S2}$ [structural]

RULE $\frac{-\ ;}{\cdot}$

RULE $\frac{\texttt{if}\ (\texttt{true})S\ \texttt{else}\ -}{S}$

RULE $\frac{\texttt{if}\ (\texttt{false})\ -\ \texttt{else}\ S}{S}$

RULE $\frac{\texttt{while}\ (E)S}{\texttt{if}\ (E)\{S\ \texttt{while}\ (E)S\}}$ [structural]

RULE $\langle \frac{\texttt{print}\ (V, Es)\ ;}{Es} \cdots\rangle_k\ \langle \cdots \frac{\cdot}{V}\rangle_{out}$ [print]

RULE $\frac{\texttt{print}\ (\cdot_{Vals})\ ;}{\cdot}$ [structural]

SYNTAX *KItem* ::= (*Id*, *Stmt*, *K*, *Map*, *Bag*)

SYNTAX *KItem* ::= popx

RULE $\langle \frac{\texttt{try } S1\ \texttt{catch}\,(X)\{S2\}}{S1 \curvearrowright \texttt{popx}} \curvearrowright K\rangle_k\ \langle \frac{\cdot}{(X, S2, K, Env, C)} \cdots\rangle_{xstack}\ C\ \langle Env \rangle_{env}$

RULE $\langle \frac{\texttt{popx}}{\cdot} \cdots\rangle_k\ \langle \frac{-}{\cdot} \cdots\rangle_{xstack}$

RULE $\langle \frac{\texttt{throw } V\ ;}{\{\ \texttt{var } X = V\ ;\ S2\ \}} \curvearrowright \frac{-}{K}\rangle_k\ \langle \frac{(X, S2, K, Env, C)}{\cdot} \cdots\rangle_{xstack}\ \frac{-}{C}\ \langle \frac{-}{Env}\rangle_{env}$

Thread spawning needs a new semantics, because we want the child thread to also share the object environment with its parent. The new semantics of thread spawning will be defined shortly. However, interestingly, the other concurrency constructs keep their semantics from SIMPLE unchanged.

RULE $\langle \langle \frac{\cdot}{\cdot} \rangle_k\ \langle H \rangle_{holds}\ \langle T \rangle_{id} \cdots\rangle_{thread}$ $\frac{\cdot}{\cdot}$ $\langle \frac{Busy}{Busy -_{Set}\ \texttt{keys}(H)}\rangle_{busy}\ \langle \cdots \frac{\cdot}{T} \cdots\rangle_{terminated}$

RULE $\langle \frac{\texttt{join } T\ ;}{\cdot} \cdots\rangle_k\ \langle \cdots T \cdots\rangle_{terminated}$

RULE $\langle \frac{\texttt{acquire } V\ ;}{\cdot} \cdots\rangle_k\ \langle \cdots \frac{\cdot}{V \mapsto 0} \cdots\rangle_{holds}\ \langle \frac{Busy}{Busy\ V}\rangle_{busy}$ requires $\neg_{Bool}(V \texttt{ in } Busy)$ [acquire]

RULE $\langle \frac{\texttt{acquire } V\ ;}{\cdot} \cdots\rangle_k\ \langle \cdots V \mapsto \frac{N}{N +_{Int} 1} \cdots\rangle_{holds}$

RULE $\langle \frac{\texttt{release } V\ ;}{\cdot} \cdots\rangle_k\ \langle \cdots V \mapsto \frac{N}{N -_{Int} 1} \cdots\rangle_{holds}$ requires $N >_{Int} 0$

RULE $\langle \frac{\texttt{release } V\ ;}{\cdot} \cdots\rangle_k\ \langle \cdots \frac{V \mapsto 0}{\cdot} \cdots\rangle_{holds}\ \langle \cdots \frac{V}{\cdot} \cdots\rangle_{busy}$

RULE $\langle \frac{\texttt{rendezvous } V\ ;}{\cdot} \cdots\rangle_k\ \langle \frac{\texttt{rendezvous } V\ ;}{\cdot} \cdots\rangle_k$ [rendezvous]

### Unchanged auxiliary operations from untyped SIMPLE

SYNTAX *Decl* ::= mkDecls (*Ids*, *Vals*) [function]

RULE $\frac{\texttt{mkDecls}((X, Xs), (V, Vs))}{\texttt{var } X = V\ ;\ \texttt{mkDecls}(Xs, Vs)}$

RULE $\frac{\texttt{mkDecls}(\cdot_{Ids}, \cdot_{Vals})}{\{\}}$

SYNTAX *KItem* ::= lookup (*Int*)

RULE $\langle \frac{\texttt{lookup}(L)}{V} \cdots\rangle_k\ \langle \cdots L \mapsto V \cdots\rangle_{store}$ [lookup]

SYNTAX *KItem* ::= env (*Map*)

RULE $\langle \frac{\texttt{env}(Env)}{\cdot} \cdots\rangle_k\ \langle \frac{-}{Env}\rangle_{env}$ [structural]

RULE $\frac{\texttt{env}(-) \curvearrowright \texttt{env}(-)}{\cdot}$ [structural]

SYNTAX *Exp* ::= lvalue (*K*)

SYNTAX *Val* ::= loc (*Int*)

RULE $\langle \texttt{lvalue}(\frac{X}{\texttt{loc}(L)}) \cdots\rangle_k\ \langle \cdots X \mapsto L \cdots\rangle_{env}$ [structural]

CONTEXT lvalue (−[□])

CONTEXT lvalue (□[−])

RULE $\texttt{lvalue}(\frac{\texttt{lookup}(L)}{\texttt{loc}(L)})$ [structural]

SYNTAX *Map* ::= *Int* ... *Int* ↦ *K* [function]

RULE $\frac{N\ ...\ M \mapsto -}{\cdot}$ requires $N >_{Int} M$

RULE $\frac{N\ ...\ M \mapsto K}{N \mapsto K\ (N +_{Int} 1)\ ...\ M \mapsto K}$ requires $N \leq_{Int} M$

### Changes to the existing untyped SIMPLE semantics

When we extend a language, sometimes we need to do more than just add new language constructs and semantics for them. Sometimes we want to also extend the semantics of existing language constructs, in order to get more from them.

**Program initialization.** In SIMPLE, once all the global declarations were processed, the function `main()` was invoked. In KOOL, the global declarations are classes, and their specific semantics is given shortly; essentially, they are pre-processed one by one and added into the class cell structure in the configuration. Once all the classes are processed, the computation item `execute`, which was placed right after the program in the initial configuration, and which can only be reached after all the classes are processed, we must call the constructor of the Main class, we can simply create an instance of the class `Main`; in SIMPLE, the program was initialized by calling the method `main()`. In KOOL, the program is initialized by creating an object instance of class `Main`. This will also implicitly call the method `Main()` (the Main class constructor). The emptiness of the env cell below is just a sanity check, to make sure that the user has not declared anything but classes at the top level of the program.

SYNTAX *KItem* ::= execute

RULE $\langle \frac{\texttt{execute}}{\texttt{new theMain}(\cdot_{Exps})\ ;} \cdots\rangle_k\ \langle \cdot_{Map} \rangle_{env}$

The semantics of new (defined below) requires the execution of all the class' declarations (and also of its superclasses').

**Object and method closures.** Before we can define the semantics of method application (previously called function application in SIMPLE), we need to add two more values to the language, namely object and method closures:

SYNTAX *Val* ::= objectClosure (*Bag*)
| methodClosure (*Id*, *Int*, *Ids*, *Stmt*)

An object value consists of an `objectClosure`-wrapped bag containing the current class of the object and the environment stack of the object. The current class of an object will always be one of the classes mapped to an environment in the environment stack of the object. A method closure encapsulates the method's parameters and code (last two arguments), as well as the object context in which the method code should execute. This object context includes the current class of the object (the first argument of `methodClosure`) and the current object environment stack (located in the object stored at the location specified as the second argument of `methodClosure`).

**Method application.** KOOL has a complex mechanism to invoke methods, because it allows both dynamic method dispatch and methods as first-class-citizen values (the latter making it a higher-order language). The invocation mechanism will be defined later. What is sufficient to know for now is that the two arguments of the application construct eventually reduce to values, the first being a method closure and the latter to a list of values. The semantics of the method closure application is then as expected: the local environment and control are stacked, then we switch to method closure's class and object environment and execute the method body. The `mkDecls` construct is the one from the unchanged SIMPLE semantics above.

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

RULE $\langle \frac{\texttt{methodClosure}(Class, OL, Xs, S)(Vs) \curvearrowright K}{\texttt{mkDecls}(Xs, Vs)\ S\ \texttt{return}\ ;}\rangle_k\ \langle \frac{Env}{\cdot_{Map}}\rangle_{env}\ \langle \cdots OL \mapsto \texttt{objectClosure}(\langle - \rangle_{crntClass}\ Obj) \cdots\rangle_{store}$
$\langle C\ \langle \frac{\cdot}{(Env, K, C\ \langle Obj' \rangle_{crntObj})} \cdots\rangle_{fstack}\ \langle \frac{Obj'}{\langle Class \rangle_{crntClass}\ Obj} \rangle_{crntObj}\rangle_{control}$

**Spawn.** We want to extend the semantics of `spawn` to also share the current object environment with the child thread, in addition to the current environment. This extension will allow us to also use method invocations in the spawned statements, which will be thus looked up as expected, using dynamic method dispatch. This lookup operation would fail if the child thread did not have access to its parent's object environment.

RULE $\langle \langle \frac{\texttt{spawn } S}{T} \cdots\rangle_k\ \langle Env \rangle_{env}\ \langle Obj \rangle_{crntObj} \cdots\rangle_{thread}$ $\frac{\cdot}{\langle \langle S \rangle_k\ \langle Env \rangle_{env}\ \langle T \rangle_{id}\ \langle Obj \rangle_{crntObj} \cdots\rangle_{thread}}$ requires fresh (T)

### Semantics of the new KOOL constructs

**Class declaration.** Initially, the classes forming the program are moved into their corresponding cells:

RULE $\langle \frac{\texttt{class } Class1 \texttt{ extends } Class2\ \{S\}}{\cdot} \cdots\rangle_k\ \langle \cdots \frac{\cdot}{\langle \langle Class1 \rangle_{className}\ \langle Class2 \rangle_{extends}\ \langle S \rangle_{declarations}\rangle_{class}} \cdots\rangle_{classes}$ [structural]

**Method declaration.** Like in SIMPLE, method names are added to the environment and bound to their code. However, unlike in SIMPLE where each function was executed in the same environment, namely the program global environment, a method in KOOL needs to be executed into its object's environment. Thus, methods evaluate to closures, which encapsulate their object's context (i.e., the current class and environment stack of the object) in addition to method's parameters and body. This approach to bind method names to method closures in the environment will also allow objects to pass their methods to other objects, to dynamically change their methods by assigning them other method closures, and even to allow all these to be done from other objects. This gives the KOOL programmer a lot of power; one should use this power wisely, though, because programs can become easily hard to understand and reason about if one overuses these features.

RULE $\langle \frac{\texttt{method } F(Xs)S}{\cdot} \cdots\rangle_k\ \langle Class \rangle_{crntClass}\ \langle OL \rangle_{location}\ \langle \frac{Env}{Env[L/F]}\rangle_{env}\ \langle \cdots \frac{\cdot}{L \mapsto \texttt{methodClosure}(Class, OL, Xs, S)} \cdots\rangle_{store}\ \langle \frac{L}{L +_{Int} 1}\rangle_{nextLoc}$

**New.** The semantics of `new` consists of two actions: memory allocation for the new object and execution of the corresponding constructor. Then the created object is returned as the result of the `new` operation; the value returned by the constructor, if any, is discarded. The current environment and object are stored onto the stack and recovered after `new` (according to the semantics of `return` borrowed from SIMPLE, when the statement "`return this;`" in the rule below is reached and evaluated), because the object creation part of `new` will destroy them. The rule below also initializes the object creation process by emptying the local environment and the current object, and allocating a location in the store where the created object will be eventually stored (this is what the "`storeObj`" task after the object creation task in the rule below will do—its rule is defined shortly). The location where the object will be stored is also made available in the `crntObj` cell, so that method closures can refer to it (see rule above).

SYNTAX *KItem* ::= (*Id*, *BagItem*)

RULE $\langle \frac{\texttt{new } Class(Vs) \curvearrowright K}{\texttt{create}(Class) \curvearrowright \texttt{storeObj} \curvearrowright Class(Vs)\ ;\ \texttt{return this}\ ;}\rangle_k\ \langle \frac{Env}{\cdot_{Map}}\rangle_{env}\ \langle \frac{L}{L +_{Int} 1}\rangle_{nextLoc}$
$\langle C\ \langle \frac{Obj}{\langle \texttt{theObject} \rangle_{crntClass}\ \langle (\texttt{theObject}, \langle \cdot_{Map} \rangle_{env}) \rangle_{envStack}\ \langle L \rangle_{location}}\rangle_{crntObj}\ \langle \frac{\cdot}{(Env, K, C\ \langle Obj \rangle_{crntObj})} \cdots\rangle_{fstack}\rangle_{control}$

The creation of a new object (the memory allocation part only) is a recursive process, requiring to first create an object for the superclass. 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.

SYNTAX *KItem* ::= create (*Id*)

RULE $\langle \frac{\texttt{create}(Class)}{\texttt{create}(Class1) \curvearrowright \texttt{setCrntClass}(Class) \curvearrowright S \curvearrowright \texttt{addEnvLayer}} \cdots\rangle_k\ \langle Class \rangle_{className}\ \langle Class1 \rangle_{extends}\ \langle S \rangle_{declarations}$ [structural]

RULE $\langle \frac{\texttt{create}(\texttt{theObject})}{\cdot} \cdots\rangle_k$ [structural]

The next operation sets the current class of the current object. This is necessary to be done at each layer, because the current class of the object is enclosed as part of the method closures (see the semantics of method declarations above).

SYNTAX *KItem* ::= setCrntClass (*Id*)

RULE $\langle \frac{\texttt{setCrntClass}(C)}{\cdot} \cdots\rangle_k\ \langle \frac{-}{C}\rangle_{crntClass}$ [structural]

The next operation adds a new tagged environment layer to the current object and gets ready for the next layer by clearing the environment (note that `create` expects the environment to be empty).

SYNTAX *KItem* ::= addEnvLayer

RULE $\langle \frac{\texttt{addEnvLayer}}{\cdot} \cdots\rangle_k\ \langle \frac{Env}{\cdot_{Map}}\rangle_{env}\ \langle Class \rangle_{crntClass}\ \langle \frac{\cdot}{(Class, Env)} \cdots\rangle_{envStack}$ [structural]

The following operation stores the created object at the location reserved by `new`. Note that the location reserved by `new` was temporarily stored in the `crntObj` cell precisely for this purpose. Now that the newly created object is stored at its location and that all method closures are aware of it, the location is unnecessary and thus we delete it from the `crntObj` cell.

SYNTAX *KItem* ::= storeObj

RULE $\langle \frac{\texttt{storeObj}}{\cdot} \cdots\rangle_k\ \langle Obj\ \langle \frac{L}{\cdot} \rangle_{location}\rangle_{crntObj}\ \langle \cdots \frac{\cdot}{L \mapsto \texttt{objectClosure}(Obj)} \cdots\rangle_{store}$ [structural]

**Self reference.** The semantics of `this` is straightforward: evaluate to the current object.

RULE $\langle \frac{\texttt{this}}{\texttt{objectClosure}(Obj)} \cdots\rangle_k\ \langle Obj \rangle_{crntObj}$ [structural]

**Object member access.** We can access an object member (field or method) either explicitly, using the construct `e.x`, or implicitly, using only the member name `x` directly. The borrowed semantics of SIMPLE will already lookup a sole name in the local environment. The first rule below reduces implicit member access to explicit access when the name cannot be found in the local environment. There are two cases to analyze for explicit object member access, depending upon whether the object is a proper object or it is just a redirection to the parent class via the construct `super`. In the first case, we evaluate the object expression and lookup the member starting with the current class (static scoping). Note the use of the conditional evaluation context. In the second case, we just lookup the member starting with the superclass of the current class. In both cases, the `lookupMember` task eventually yields a `lookup(L)` task for some appropriate location `L`, which will be further solved with the corresponding rule borrowed from SIMPLE. Note that the current object is not altered by `super`, so future method invocations see the entire object, as needed for dynamic method dispatch.

RULE $\langle \frac{X}{\texttt{this}\ .\ X} \cdots\rangle_k\ \langle Env \rangle_{env}$ requires $\neg_{Bool}(X \texttt{ in keys}(Env))$ [structural]

CONTEXT □ . − requires □ ≠_K super

RULE $\frac{\texttt{objectClosure}(\langle Class \rangle_{crntClass}\ \langle (Class, EnvC)\ EStack \rangle_{envStack})\ .\ X}{\texttt{lookupMember}(\langle (Class, EnvC)\ EStack \rangle_{envStack}, X)}$ [structural]

RULE $\texttt{objectClosure}(\langle \frac{Class}{\ } \rangle_{crntClass}\ \langle \frac{(Class', -)}{\cdot_{List}} \cdots\rangle_{envStack})\ .\ X$ requires $Class \neq_{K} Class'$ [structural]

RULE $\langle \frac{\texttt{super}\ .\ X}{\texttt{lookupMember}(\langle EStack \rangle_{envStack}, X)} \cdots\rangle_k\ \langle Class \rangle_{crntClass}\ \langle (Class, -)\ EStack \rangle_{envStack}$ [structural]

RULE $\langle \texttt{super}\ .\ X \cdots\rangle_k\ \langle Class \rangle_{crntClass}\ \langle \frac{(Class', -)}{\cdot_{List}} \cdots\rangle_{envStack}$ requires $Class \neq_{K} Class'$ [structural]

**Method invocation.** Unlike in SIMPLE, in KOOL application was declared strict only in its second argument. That is because we want to ensure dynamic method dispatch when the first argument is a method access. As a consequence, we need to consider all the cases of interest for the first argument and to explicitly say what to do in each case. For instance, for method access in a proper object (i.e., not super), we want the same behavior for the first argument as if it was not in a method invocation position. When it is a member access (the third rule below), we look it up starting with the instance class of the corresponding object. This ensures dynamic dispatch for methods; it actually dynamically dispatches field accesses, too, which is correct in KOOL, because one can assign method closures to fields and the field appeared in a method invocation context. The last context declaration below says that method applications or array accesses are also allowed as first argument to applications; that is because methods are allowed to return methods and arrays are allowed to hold methods in KOOL, since it is higher-order. If that is the case, then we want to evaluate the method call or the array access.

RULE $\langle \frac{X}{V}(-) \cdots\rangle_k\ \langle \cdots X \mapsto L \cdots\rangle_{env}\ \langle \cdots L \mapsto V \cdots\rangle_{store}$ [lookup]

RULE $\langle \frac{X}{\texttt{this}\ .\ X}(-) \cdots\rangle_k\ \langle Env \rangle_{env}$ requires $\neg_{Bool}(X \texttt{ in keys}(Env))$ [structural]

CONTEXT □ . −(−) requires □ ≠_K super

RULE $\frac{\texttt{objectClosure}(Obj\ \langle EStack \rangle_{envStack})\ .\ X(-)}{\texttt{lookupMember}(\langle EStack \rangle_{envStack}, X)}$ [structural]

RULE $\langle \frac{\texttt{super}\ .\ X}{\texttt{lookupMember}(\langle EStack \rangle_{envStack}, X)}(-) \cdots\rangle_k\ \langle Class \rangle_{crntClass}\ \langle (Class, -)\ EStack \rangle_{envStack}$ [structural]

RULE $\langle (\texttt{super}\ .\ X)(-) \cdots\rangle_k\ \langle Class \rangle_{crntClass}\ \langle \frac{(Class', -)}{\cdot_{List}} \cdots\rangle_{envStack}$ requires $Class \neq_{K} Class'$ [structural]

CONTEXT □(−) requires getKLabel(□) ==_KLabel '_(_) ∨_Bool getKLabel(□) ==_KLabel '_[_] ∨_Bool … Exp1:Exps

Eventually, each of the rules above produces a `lookup(L)` task as a replacement for the method. When that happens, we just lookup the value at location `L`:

RULE $\langle \frac{\texttt{lookup}(L)}{V}(-) \cdots\rangle_k\ \langle \cdots L \mapsto V \cdots\rangle_{store}$ [lookup]

The value `V` looked up above is expected to be a method closure, in which case the semantics of method application given above will apply. Otherwise, the execution will get stuck.

**Instance Of.** It searches the object environment for a layer corresponding to the desired class. It returns `true` iff it can find the class, otherwise it returns `false`; it only gets stuck when its first argument does not evaluate to an object.

RULE $\frac{\texttt{objectClosure}(-\ \langle (C, -) \cdots\rangle_{envStack})\ \texttt{instanceOf}\ C}{\texttt{true}}$

RULE $\texttt{objectClosure}(-\ \langle \frac{(C, -)}{\cdot_{List}} \cdots\rangle_{envStack})\ \texttt{instanceOf}\ C'$ requires $C \neq_{K} C'$ [structural]

RULE $\frac{\texttt{objectClosure}(-\ \langle \cdot_{List} \rangle_{envStack})\ \texttt{instanceOf}\ -}{\texttt{false}}$

**Cast.** In untyped KOOL we prefer to not check the validity of casting. In other words, any cast is allowed on any object, simply changing the current class of the object to the desired class. The execution will get stuck later if one attempts to access a field which is not available. Moreover, the execution may complete successfully even in the presence of invalid casts, provided that each accessed member during the current execution is, or happens to be, available.

RULE $\frac{(C)\ \texttt{objectClosure}(\langle - \rangle_{crntClass}\ Obj)}{\texttt{objectClosure}(\langle C \rangle_{crntClass}\ Obj)}$

### KOOL-specific auxiliary declarations and operations

Here we define all the auxiliary constructs used in the above KOOL-specific semantics (those used in the SIMPLE fragment have already been defined in a corresponding section above).

**Objects as lvalues.** The current machinery borrowed with the semantics of SIMPLE allows us to enrich the set of lvalues, this way allowing new means to assign values to locations. In KOOL, we want object member names to be lvalues, so that we can assign values to them using the already existing machinery. The first rule below ensures that the object is always explicit, the evaluation context enforces the object to be evaluated, and finally the second rule initiates the lookup for the member's location based on the current class of the object.

RULE $\langle \texttt{lvalue}(\frac{X}{\texttt{this}\ .\ X}) \cdots\rangle_k\ \langle Env \rangle_{env}$ requires $\neg_{Bool}(X \texttt{ in keys}(Env))$ [structural]

CONTEXT lvalue (□ . −)

RULE $\frac{\texttt{lvalue}(\texttt{objectClosure}(\langle Class \rangle_{crntClass}\ \langle (Class, EnvC)\ EStack \rangle_{envStack})\ .\ X)}{\texttt{lookupMember}(\langle (Class, EnvC)\ EStack \rangle_{envStack}, X)}$ [structural]

RULE $\texttt{lvalue}(\texttt{objectClosure}(\langle Class \rangle_{crntClass}\ \langle \frac{(Class', -)}{\cdot_{List}} \cdots\rangle_{envStack})\ .\ X)$ requires $Class \neq_{K} Class'$ [structural]

**Lookup member.** It searches for the given member in the given environment stack, starting with the most concrete class and going up in the hierarchy.

SYNTAX *KItem* ::= lookupMember (*BagItem*, *Id*) [function]

RULE $\frac{\texttt{lookupMember}(\langle (-, X \mapsto L\ -) \cdots\rangle_{envStack}, X)}{\texttt{lookup}(L)}$ [structural]

RULE $\texttt{lookupMember}(\langle \frac{(-, Env)}{\cdot_{List}} \cdots\rangle_{envStack}, X)$ requires $\neg_{Bool}(X \texttt{ in keys}(Env))$ [structural]

END MODULE