KOOL — Typed — Dynamic

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

Abstract

This is the K dynamic semantics of the typed KOOL language. It is very similar to the semantics of the untyped KOOL, the difference being that we now check the typing policy dynamically. Since we have to now declare the types of variables and methods, we adopt a syntax for those which is close to Java. Like in the semantics of untyped SIMPLE, where we borrowed almost all the semantics of untyped SIMPLE, we are going to also borrow much of the semantics of dynamically typed SIMPLE here. We will highlight the differences between the dynamically typed and the untyped KOOL as we proceed with the semantics. In general, the type policy of the typed KOOL language is similar to that of Java. You may find it useful to also read the discussion in the preamble of the static semantics of typed KOOL before proceeding.

MODULE KOOL-TYPED-DYNAMIC-SYNTAX

Syntax

Like for the untyped KOOL language, the syntax of typed KOOL extends that of typed SIMPLE with object-oriented constructs. The syntax below was produced by copying and modifying/extending the syntax of dynamically typed SIMPLE. In fact, the only change we made to the existing syntax of dynamically typed SIMPLE was to change the strictness of the application construct like in untyped KOOL, from `strict` to `strict(2)` (because application is not strict in the first argument anymore due to dynamic method dispatch). The KOOL-specific syntactic extensions are identical to those in untyped KOOL.

SYNTAX *Id* ::= `Object`
| `Main`

Types

SYNTAX *Type* ::= `void`
| `int`
| `bool`
| `string`
| *Id*
| *Type*`[]`
| *Types* `->` *Type*
| (*Type*) [bracket]

SYNTAX *Types* ::= *List*{*Type*,","}

Declarations

SYNTAX *Param* ::= *Type Id*

SYNTAX *Params* ::= *List*{*Param*,","}

SYNTAX *Decl* ::= *Type Exps* `;` [avoid]
| *Type Id*(*Params*)*Block*
| `class` *Id Block*
| `class` *Id* `extends` *Id Block*

Expressions

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

Statements

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

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

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

Desugaring macros

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

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

RULE $\frac{T\ E1,E2,Es\ ;}{T\ E1\ ;\ T\ E2,Es\ ;}$ [macro]

RULE $\frac{T\ X = E\ ;}{T\ X\ ;\ X = E\ ;}$ [macro]

RULE $\frac{\texttt{class}\ C\ S}{\texttt{class}\ C\ \texttt{extends Object}\ S}$ [macro]

END MODULE

Semantics

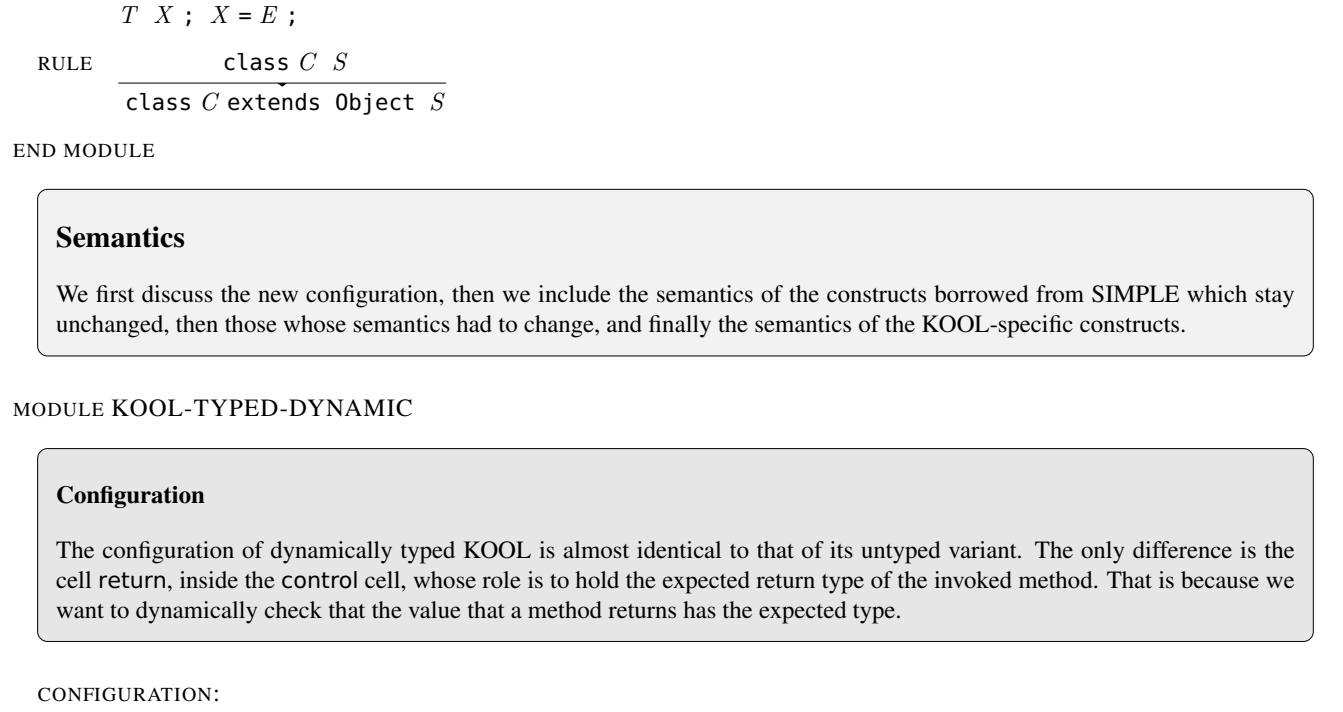
We first discuss the new configuration, then we include the semantics of the constructs borrowed from SIMPLE which stay unchanged, then those whose semantics had to change, and finally the semantics of the KOOL-specific constructs.

MODULE KOOL-TYPED-DYNAMIC

Configuration

The configuration of dynamically typed KOOL is almost identical to that of its untyped variant. The only difference is the cell return, inside the control cell, whose role is to hold the expected return type of the invoked method. That is because we want to dynamically check that the value that a method returns has the expected type.

CONFIGURATION:



Unchanged semantics from dynamically typed SIMPLE

The semantics below is taken over from dynamically typed SIMPLE unchanged. Like for untyped KOOL, the semantics of function/method declaration and invocation, and of program initialization needs to change. Moreover, due to subtyping, the semantics of several imported SIMPLE constructs can be made more general, such as that of the return statement, that of the assignment, and that of the exceptions. We removed all these from the imported semantics of SIMPLE below and gave their modified semantics right after, together with the extended semantics of thread spawning (which is identical to that of untyped KOOL).

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

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

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

SYNTAX *K* ::= $\perp_{Type}$

RULE ⟨ $\frac{T\ X\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{Env}{Env[L/X]}$ ⟩$_{env}$ ⟨⋯ $\frac{\cdot_{Map}}{L \mapsto \perp_T}$ ⋯⟩$_{store}$ ⟨ $\frac{L}{L +_{Int} 1}$ ⟩$_{nextLoc}$

RULE ⟨ $\frac{T\ X[N]\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{Env}{Env[L/X]}$ ⟩$_{env}$ ⟨⋯ $\frac{\cdot_{Map}}{L \mapsto \texttt{array}\,(T, L +_{Int} 1, N)\ (L +_{Int} 1) \ldots (L +_{Int} N) \mapsto \perp_T}$ ⋯⟩$_{store}$ ⟨ $\frac{L}{L +_{Int} 1 +_{Int} N}$ ⟩$_{nextLoc}$ requires $N \geq_{Int} 0$

CONTEXT — — [□] ;

SYNTAX *Id* ::= `$1`
| `$2`

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

RULE ⟨ $\frac{X}{V}$ ⋯⟩$_k$ ⟨⋯ $X \mapsto L$ ⋯⟩$_{env}$ ⟨⋯ $L \mapsto V$ ⋯⟩$_{store}$ [lookup]

CONTEXT `++` $\frac{\square}{\texttt{lvalue}\,(\square)}$

RULE ⟨ $\frac{\texttt{++ loc}\,(L)}{I +_{Int} 1}$ ⋯⟩$_k$ ⟨⋯ $L \mapsto \frac{I}{I +_{Int} 1}$ ⋯⟩$_{store}$ [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 $\frac{V1\ \texttt{!=}\ V2}{V1 \neq_K V2}$

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, anywhere]

RULE $\frac{\texttt{array}\,(-, L, M)[N]}{\texttt{lookup}\,(L +_{Int} N)}$ requires $N \geq_{Int} 0 \wedge_{Bool} N <_{Int} M$ [structural, anywhere]

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

SYNTAX *Val* ::= `nothing` (*Type*)

RULE ⟨ $\frac{\texttt{return ;}}{\texttt{return nothing}\,(T)\ ;}$ ⋯⟩$_k$ ⟨ T ⟩$_{return}$ [structural]

RULE ⟨ $\frac{\texttt{read}\,()}{I}$ ⋯⟩$_k$ ⟨ $\frac{I}{\cdot_{List}}$ ⋯⟩$_{in}$ [read]

CONTEXT $\frac{\square}{\texttt{lvalue}\,(\square)}$ = —

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

RULE ⟨ $\frac{\{S\}}{S \curvearrowright \texttt{env}\,(Env)}$ ⋯⟩$_k$ ⟨ Env ⟩$_{env}$ [structural]

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

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

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 ⟨ $\frac{\texttt{print}\,(V, Es)\ ;}{Es}$ ⋯⟩$_k$ ⟨⋯ $\frac{\cdot_{List}}{V}$ ⟩$_{out}$ requires $\texttt{typeOf}\,(V) =_K \texttt{int} \vee_{Bool} \texttt{typeOf}\,(V) =_K \texttt{string}$ [print]

RULE $\frac{\texttt{print}\,(\cdot_{Exps})\ ;}{\cdot_K}$ [structural]

RULE ⟨ ⟨ $\frac{\cdot_K}{}$ ⟩$_k$ ⟨ H ⟩$_{holds}$ ⟨ T ⟩$_{id}$ ⋯⟩$_{thread}$ $\frac{}{\cdot_{Bag}}$ ⟨ $\frac{Busy}{Busy -_{Set}\ \texttt{keys}\,(H)}$ ⟩$_{busy}$ ⟨⋯ $\frac{\cdot_{Set}}{T}$ ⋯⟩$_{terminated}$

RULE ⟨ $\frac{\texttt{join}\ T\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ T ⋯⟩$_{terminated}$

RULE ⟨ $\frac{\texttt{acquire}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ $\frac{\cdot_{Map}}{V \mapsto 0}$ ⋯⟩$_{holds}$ ⟨ $\frac{Busy}{Busy\ V}$ ⟩$_{busy}$ requires $\neg_{Bool}(V \in Busy)$ [acquire]

RULE ⟨ $\frac{\texttt{acquire}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ $V \mapsto \frac{N}{N +_{Int} 1}$ ⋯⟩$_{holds}$

RULE ⟨ $\frac{\texttt{release}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ $V \mapsto \frac{N}{N -_{Int} 1}$ ⋯⟩$_{holds}$ requires $N >_{Int} 0$

RULE ⟨ $\frac{\texttt{release}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ $\frac{V \mapsto 0}{\cdot_{Map}}$ ⋯⟩$_{holds}$ ⟨⋯ $\frac{V}{\cdot_{Set}}$ ⋯⟩$_{busy}$

RULE ⟨ $\frac{\texttt{rendezvous}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{\texttt{rendezvous}\ V\ ;}{\cdot_K}$ ⋯⟩$_k$ [rendezvous]

Unchanged auxiliary operations from dynamically typed SIMPLE.

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

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

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

SYNTAX *K* ::= `lookup` (*Int*)

RULE ⟨ $\frac{\texttt{lookup}\,(L)}{V}$ ⋯⟩$_k$ ⟨⋯ $L \mapsto V$ ⋯⟩$_{store}$ [lookup]

SYNTAX *K* ::= `env` (*Map*)

RULE ⟨ $\frac{\texttt{env}\,(Env)}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{-}{Env}$ ⟩$_{env}$ [structural]

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

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

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

RULE ⟨ $\texttt{lvalue}\,(\frac{X}{\texttt{loc}\,(L)})$ ⋯⟩$_k$ ⟨⋯ $X \mapsto L$ ⋯⟩$_{env}$ [structural]

CONTEXT `lvalue` (—[□])

CONTEXT `lvalue` (□[—])

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

SYNTAX *Type* ::= *Type* < *Vals* > [function]

RULE $\frac{T < -, Vs >}{T[] < Vs >}$

RULE $\frac{T < \cdot_{Vals} >}{T}$

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

RULE $\frac{N \ldots M \mapsto -}{\cdot_{Map}}$ requires $N >_{Int} M$

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

SYNTAX *Type* ::= `typeOf` (*K*) [function]

RULE $\frac{\texttt{typeOf}\,(-)}{\texttt{int}}$

RULE $\frac{\texttt{typeOf}\,(-)}{\texttt{bool}}$

RULE $\frac{\texttt{typeOf}\,(-)}{\texttt{string}}$

RULE $\frac{\texttt{typeOf}\,(\texttt{array}\,(T,-,-))}{(T[])}$

RULE $\frac{\texttt{typeOf}\,(\perp_T)}{T}$

RULE $\frac{\texttt{typeOf}\,(\texttt{nothing}\,(T))}{T}$

SYNTAX *Types* ::= `getTypes` (*Params*) [function]

RULE $\frac{\texttt{getTypes}\,(T\ -)}{T, \cdot_{Types}}$

RULE $\frac{\texttt{getTypes}\,(T\ -, P, Ps)}{T, \texttt{getTypes}\,(P, Ps)}$

RULE $\frac{\texttt{getTypes}\,(\cdot_{Params})}{\texttt{void}, \cdot_{Types}}$

Changes to the existing dynamically typed SIMPLE semantics

We extend/change the semantics of several SIMPLE constructs in order to take advantage of the richer KOOL semantic infrastructure and thus get more from the existing SIMPLE constructs.

Program initialization. Like in untyped KOOL.

SYNTAX *K* ::= `execute`

RULE ⟨ $\frac{\texttt{execute}}{\texttt{new Main}(\cdot_{Exps})\ ;}$ ⟩$_k$ ⟨ $\frac{}{\cdot_{Map}}$ ⟩$_{env}$ [structural]

Method application. The only change to untyped KOOL's values is that method closures are now typed (their first argument holds their type):

SYNTAX *Val* ::= `objectClosure` (*Bag*)
| `methodClosure` (*Type*, *Id*, *Int*, *Params*, *Stmt*)

The type held by a method closure will be the entire type of the method, not only its result type like the lambda-closure of typed SIMPLE. The reason for this change comes from the the need to perform more checks. Since values carry their own types, we also need to check for subtyping of closures with respect to superclass types and the declared type of the variable, a legitimate operation in a type-safe language; we have to be able to dynamically upcast them, and in order to do that elegantly it is convenient to store the "current type" of the method closure instead of just its result type. Note that this was unnecessary in the semantics of the dynamically typed SIMPLE language.

Method closure application needs to also set a new return type in the return cell, like in dynamically typed SIMPLE, in order for the values returned by its body to be checked against the expected return type of the method. To do this correctly, we also need to stack the current status of the return cell and then pop it when the returned value is obtained, so the calling environment can continue; so we group them together in the stack frame.

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

RULE ⟨ $\frac{\texttt{methodClosure}\,(- \texttt{->}\ T, Class, OL, Ps, S)(Vs) \curvearrowright K}{\texttt{mkDecls}\,(Ps, Vs)\ S\ \texttt{return ;}}$ ⟩$_k$ ⟨ $\frac{Env}{\cdot_{Map}}$ ⟩$_{env}$ ⟨⋯ $OL \mapsto \texttt{objectClosure}\,(\langle - \rangle_{crntClass}\ Obj)$ ⋯⟩$_{store}$ ⟨ C ⟨ $\frac{\cdot_{List}}{(Env, K, C\ \langle Obj' \rangle_{crntObj}\ \langle T' \rangle_{return})}$ ⋯⟩$_{fstack}$ ⟨ $\frac{T'}{T}$ ⟩$_{return}$ ⟨ $\frac{Obj'}{Obj}$ ⟨ $\frac{-}{Class}$ ⟩$_{crntClass}$ ⟩$_{crntObj}$ ⟩$_{control}$

At method return, we have to check that the type of the returned value is a subtype of the expected return type. Moreover, if that is the case, then we also upcast the returned value to one of the expected type. The computation item `unsafeCast(V,T)` changes the type V to T without any additional checks; however, it only does it when V is an object or a method, otherwise it returns V unchanged.

RULE ⟨ $\frac{\texttt{return}\ V\ ;\ \curvearrowright -}{\texttt{subtype}\,(\texttt{typeOf}\,(V), T) \curvearrowright \texttt{true?} \curvearrowright \texttt{unsafeCast}\,(V, T) \curvearrowright K}$ ⟩$_k$ ⟨ ⟨ $\frac{(Env, K, C)}{\cdot_{List}}$ ⋯⟩$_{fstack}$ ⟨ $\frac{T}{}$ ⟩$_{return}$ $\frac{-}{C}$ ⟩$_{control}$ ⟨ $\frac{-}{Env}$ ⟩$_{env}$

Assignment. Typed KOOL allows to assign subtype instance values to supertype lvalues. The semantics of assignment below is similar in spirit to dynamically typed SIMPLE's, but a check is performed that the assigned value's type is a subtype of the location's type. If that is the case, then the assigned value is returned as a result and stored, but it is upcast appropriately first, so the context will continue to see a value of the expected type of the location. Note that the type of a location is implicit in the type of its contents and it never changes during the execution of a program; its type is assigned when the location is allocated and initialized, and then only type-preserving values are allowed to be stored in each location.

RULE ⟨ $\frac{\texttt{loc}\,(L) = V}{\texttt{subtype}\,(\texttt{typeOf}\,(V), \texttt{typeOf}\,(V')) \curvearrowright \texttt{true?} \curvearrowright \texttt{unsafeCast}\,(V, \texttt{typeOf}\,(V'))}$ ⋯⟩$_k$ ⟨⋯ $L \mapsto \frac{V'}{\texttt{unsafeCast}\,(V, \texttt{typeOf}\,(V'))}$ ⋯⟩$_{store}$ [assignment]

Typed exceptions. Exceptions are propagated now until a catch that can handle them is encountered.

SYNTAX *K* ::= (*Param*, *Stmt*, *K*, *Map*, *Bag*)

SYNTAX *K* ::= `popx`

RULE ⟨ $\frac{\texttt{try}\ S1\ \texttt{catch}\,(P)S2 \curvearrowright K}{S1 \curvearrowright \texttt{popx}}$ ⟩$_k$ ⟨ ⟨ $\frac{\cdot_{List}}{(P, S2, K, Env, C)}$ ⋯⟩$_{xstack}$ C ⟩$_{control}$ ⟨ Env ⟩$_{env}$

RULE ⟨ $\frac{\texttt{popx}}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{-}{\cdot_{List}}$ ⋯⟩$_{xstack}$

RULE ⟨ $\frac{\texttt{throw}\ V\ ;\ \curvearrowright -}{\texttt{if}\,(\texttt{subtype}\,(\texttt{typeOf}\,(V), T))\{T\ X = V\ ;\ S2\}\ \texttt{else}\ \{\texttt{throw}\ V\ ;\}\ \curvearrowright K}$ ⟩$_k$ ⟨ ⟨ $\frac{(T\ X, S2, K, Env, C)}{\cdot_{List}}$ ⋯⟩$_{xstack}$ $\frac{-}{C}$ ⟩$_{control}$ ⟨ $\frac{-}{Env}$ ⟩$_{env}$

Spawn. Like in untyped KOOL.

RULE ⟨ ⟨ $\frac{\texttt{spawn}\ S}{T}$ ⋯⟩$_k$ ⟨ Env ⟩$_{env}$ ⟨ Obj ⟩$_{crntObj}$ ⋯⟩$_{thread}$ $\frac{\cdot_{Bag}}{\langle\langle S \rangle_k \langle Env \rangle_{env} \langle T \rangle_{id} \langle Obj \rangle_{crntObj} \cdots\rangle_{thread}}$ requires `fresh` (T)

Semantics of the new KOOL constructs

Class declaration. Like in untyped KOOL.

RULE ⟨ $\frac{\texttt{class}\ Class1\ \texttt{extends}\ Class2\ \{S\}}{\cdot_K}$ ⋯⟩$_k$ ⟨⋯ $\frac{\cdot_{Bag}}{\langle \langle Class1 \rangle_{className} \langle Class2 \rangle_{extends} \langle S \rangle_{declarations} \rangle_{class}}$ ⋯⟩$_{classes}$ [structural]

Method declaration. Methods are now typed and we need to store their types in their closures, so that their type contract can be checked at invocation time. The rule below is conceptually similar to that of untyped KOOL; the only difference is the addition of the types.

RULE ⟨ $\frac{T\ F(Ps)S}{\cdot_K}$ ⋯⟩$_k$ ⟨ C ⟩$_{crntClass}$ ⟨ OL ⟩$_{location}$ ⟨ $\frac{Env}{Env[L/F]}$ ⟩$_{env}$ ⟨⋯ $\frac{\cdot_{Map}}{L \mapsto \texttt{methodClosure}\,(\texttt{getTypes}\,(Ps)\ \texttt{->}\ T, C, OL, Ps, S)}$ ⋯⟩$_{store}$ ⟨ $\frac{L}{L +_{Int} 1}$ ⟩$_{nextLoc}$

New. The semantics of new in dynamically typed KOOL is also similar to that in untyped KOOL, the main difference being the management of the return types. Indeed, when a new object is created we also have to stack the current type in the return cell in order to be recovered after the creation of the new object. Only the first rule below needs to be changed; the others are identical to those in untyped KOOL.

SYNTAX *K* ::= (*Id*, *Bag*)

RULE ⟨ $\frac{\texttt{new}\ Class(Vs) \curvearrowright K}{\texttt{create}\,(Class) \curvearrowright \texttt{storeObj} \curvearrowright ((Class(Vs)\ ;\ \texttt{return this ;})}$ ⟩$_k$ ⟨ $\frac{Env}{\cdot_{Map}}$ ⟩$_{env}$ ⟨ $\frac{L}{L +_{Int} 1}$ ⟩$_{nextLoc}$ ⟨ C ⟨ $\frac{Obj}{\langle\langle \texttt{Object} \rangle_{crntClass} \langle (\texttt{Object}, \langle \cdot_{Map} \rangle_{env}) \rangle_{envStack} \langle L \rangle_{location}\rangle_{crntObj}}$ ⟩$_{crntObj}$ ⟨ $\frac{T}{Class}$ ⟩$_{return}$ ⟨ $\frac{\cdot_{List}}{(Env, K, C\ \langle T \rangle_{return}\ \langle Obj \rangle_{crntObj})}$ ⋯⟩$_{fstack}$ ⟩$_{control}$

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

RULE ⟨ $\frac{\texttt{create}\,(Class)}{\texttt{create}\,(Class1) \curvearrowright \texttt{setCrntClass}\,(Class) \curvearrowright S \curvearrowright \texttt{addEnvLayer}}$ ⋯⟩$_k$ ⟨ $Class$ ⟩$_{className}$ ⟨ $Class1$ ⟩$_{extends}$ ⟨ S ⟩$_{declarations}$ [structural]

RULE ⟨ $\frac{\texttt{create}\,(\texttt{Object})}{\cdot_K}$ ⋯⟩$_k$ [structural]

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

RULE ⟨ $\frac{\texttt{setCrntClass}\,(C)}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{-}{C}$ ⟩$_{crntClass}$ [structural]

SYNTAX *K* ::= `addEnvLayer`

RULE ⟨ $\frac{\texttt{addEnvLayer}}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{Env}{\cdot_{Map}}$ ⟩$_{env}$ ⟨ $Class$ ⟩$_{crntClass}$ ⟨ $\frac{\cdot_{List}}{(Class, \langle Env \rangle_{env})}$ ⋯⟩$_{envStack}$ [structural]

SYNTAX *K* ::= `storeObj`

RULE ⟨ $\frac{\texttt{storeObj}}{\cdot_K}$ ⋯⟩$_k$ ⟨ $\frac{Obj\ \langle L \rangle_{location}}{\cdot_{Bag}}$ ⟩$_{crntObj}$ ⟨⋯ $\frac{\cdot_{Map}}{L \mapsto \texttt{objectClosure}\,(Obj)}$ ⋯⟩$_{store}$

Self reference. Like in untyped KOOL.

RULE ⟨ $\frac{\texttt{this}}{\texttt{objectClosure}\,(Obj)}$ ⋯⟩$_k$ ⟨ Obj ⟩$_{crntObj}$ [structural]

Object member access. Like in untyped KOOL.

RULE ⟨ $\frac{X}{\texttt{this}\ .\ X}$ ⋯⟩$_k$ ⟨ Env ⟩$_{env}$ requires $\neg_{Bool}(X\ \texttt{in keys}\,(Env))$ [structural]

CONTEXT □ . — requires (□ $\neq_K$ `super`)

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

RULE ⟨ $\frac{\texttt{super}\ .\ X}{\texttt{lookupMember}\,(\langle EStack \rangle_{envStack}, X)}$ ⋯⟩$_k$ ⟨ $Class$ ⟩$_{crntClass}$ ⟨ $(Class, EnvC)\ EStack$ ⟩$_{envStack}$ [structural]

Method invocation. The method lookup is the same as in untyped KOOL.

RULE ⟨ $\frac{X}{V}(-)$ ⋯⟩$_k$ ⟨⋯ $X \mapsto L$ ⋯⟩$_{env}$ ⟨⋯ $L \mapsto V$ ⋯⟩$_{store}$ [lookup]

RULE ⟨ $\frac{X}{\texttt{this}\ .\ X}(-)$ ⋯⟩$_k$ ⟨ Env ⟩$_{env}$ requires $\neg_{Bool}(X\ \texttt{in keys}\,(Env))$ [structural]

CONTEXT □ . —(—) requires (□ $\neq_K$ `super`)

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

RULE ⟨ $\frac{\texttt{super}\ .\ X}{\texttt{lookupMember}\,(\langle EStack \rangle_{envStack}, X)}(-)$ ⋯⟩$_k$ ⟨ $Class$ ⟩$_{crntClass}$ ⟨ $(Class, -)\ EStack$ ⟩$_{envStack}$ [structural]

CONTEXT □(—) requires `getKLabel` □ $\neq_{KLabel}$ '_._ $\wedge_{Bool}$ `getKLabel` □ $\neq_{KLabel}$ 'lookup_

RULE ⟨ $\frac{\texttt{lookup}\,(L)}{V}(-)$ ⋯⟩$_k$ ⟨⋯ $L \mapsto V$ ⋯⟩$_{store}$ [lookup]

Instance of. Like in untyped KOOL.

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

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

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

Cast. Unlike in untyped KOOL, in typed KOOL we actually check that the object can indeed be cast to the claimed type.

RULE $\frac{(C)\ \texttt{objectClosure}\,(\langle - \rangle_{crntClass}\ Obj)}{\texttt{objectClosure}\,(Obj)\ \texttt{instanceOf}\ C \curvearrowright \texttt{true?} \curvearrowright \texttt{objectClosure}\,(\langle C \rangle_{crntClass}\ Obj)}$

KOOL-specific auxiliary declarations and operations

P@

Objects as lvalues. Like in untyped KOOL.

RULE ⟨ $\texttt{lvalue}\,(\frac{X}{\texttt{this}\ .\ X})$ ⋯⟩$_k$ ⟨ Env ⟩$_{env}$ requires $\neg_{Bool}(X\ \texttt{in keys}\,(Env))$ [structural]

CONTEXT `lvalue` (□ . —)

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

Lookup member. Like in untyped KOOL.

SYNTAX *K* ::= `lookupMember` (*Bag*, *Id*) [function]

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

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

typeOf for the additional values.

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

RULE $\frac{\texttt{typeOf}\,(\texttt{methodClosure}\,(T, -, -, -, -))}{T}$

Subtype checking. The subclass relation induces a subtyping relation.

SYNTAX *Exp* ::= `subtype` (*Types*, *Types*)

RULE $\frac{\texttt{subtype}\,(T, T)}{\texttt{true}}$ [structural]

RULE ⟨ $\frac{\texttt{subtype}\,(C1, C)}{\texttt{subtype}\,(C2, C)}$ ⋯⟩$_k$ ⟨ $C1$ ⟩$_{className}$ ⟨ $C2$ ⟩$_{extends}$ requires $C1 \neq_K C$ [structural]

RULE $\frac{\texttt{subtype}\,(\texttt{Object}, Class)}{\texttt{false}}$ requires $Class \neq_K$ `Object` [structural]

RULE $\frac{\texttt{subtype}\,(Ts \texttt{->} T2, Ts' \texttt{->} T2')}{\texttt{subtype}\,(((T2), Ts'), ((T2'), Ts))}$ [structural]

RULE $\frac{\texttt{subtype}\,((T, Ts), (T', Ts'))}{\texttt{subtype}\,(T, T')\ \&\&\ \texttt{subtype}\,(Ts, Ts')}$ requires $Ts \neq_K \cdot_{Types}$ [structural]

RULE $\frac{\texttt{subtype}\,(\cdot_{Types}, \cdot_{Types})}{\texttt{true}}$ [structural]

Unsafe Casting. Performs unsafe casting. One should only use it in combination with the subtype relation above.

SYNTAX *Val* ::= `unsafeCast` (*Val*, *Type*) [function]

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

RULE $\frac{\texttt{unsafeCast}\,(\texttt{methodClosure}\,(T', C, OL, Ps, S), T)}{\texttt{methodClosure}\,(T, C, OL, Ps, S)}$

RULE $\frac{\texttt{unsafeCast}\,(V, T)}{V}$ requires $\texttt{typeOf}\,(V) =_K T$

Generic guard. A generic computational guard: it allows the computation to continue only if a prefix guard evaluates to *true*.

SYNTAX *K* ::= `true?`

RULE $\frac{\texttt{true} \curvearrowright \texttt{true?}}{\cdot_K}$ [structural]

END MODULE