KOOL-TYPED-STATIC

KOOL — Typed — Static Grigore Roşu and Traian Florin Şerbănuţă (`{grosu,tserban2}@illinois.edu`)

Abstract

This is the $\mathbb{K}$ static semantics of the typed KOOL language. It extends the static semantics of typed SIMPLE with static semantics for the object-oriented constructs. Also, the static semantics of some of the existing SIMPLE constructs need to change, in order to become more generous with regards to the set of accepted programs, mostly due to subtyping. For example, the assignment construct "`x = e`" required that both the variable `x` and the expression `e` had the same type in SIMPLE. In KOOL, the type of `e` can be a subtype of the type of `x`. Specifically, we define the following typing policy for KOOL, everything else not mentioned below borrowing its semantics from SIMPLE:

- Each class `C` yields a homonymous type, which can be explicitly used in programs to type variables and methods, possibly in combination with other types.
- Since now we have user-defined types, we check that each type used in a KOOL program is well-formed, that is, it is constructed only from primitive and class types corresponding to declared classes.
- Class members and their types form a *class type environment*. Each class will have such a type environment. Each member in a class is allowed to be declared only once. Since in KOOL we allow methods to be assigned to fields, we make no distinction between field and method members; in other words, we reject programs declaring both a field and a method with the same name.
- If an identifier is not found in the local type environment, it will be searched for in the current class type environment. If not there, then it will be searched for in its superclass' type environment. And so on and so forth. If not found until the `Object` class is reached, a typing error is reported.
- The assignment allows variables to be assigned values of more concrete types. The result type of the assignment expression construct will be the (more abstract) type of the assigned variable, and not the (more concrete) type of the expression, like in Java.
- Exceptions are changed (from SIMPLE) to allow throwing and catching only objects, like in Java. Also, unlike in SIMPLE, we do not check whether the type of the thrown exception matches the type of the caught variable, because exceptions can be caught by other `try/catch` blocks, even by ones in other methods. To avoid having to annotate each method with what exceptions it can throw, we prefer to not check the type safety of exceptions (although this is an excellent homework!). We only check that the `try` block type-checks and that the `catch` block type-checks after we bind the caught variable to its claimed type.
- Class declarations are not allowed to have any cycles in their extends relation. Such cycles would lead to non-termination of `new`, as it actually does in the dynamic semantics of KOOL where no such circularity checks are performed.
- Methods overriding other methods should be in the right subtyping relationship with the overridden methods: co-variant in the codomain and contra-variant in the domain.

MODULE KOOL-TYPED-STATIC-SYNTAX

Syntax

The syntax of statically typed KOOL is identical to that of dynamically typed KOOL, they both taking as input the same programs. What differs is the $\mathbb{K}$ strictness attributes. Like in statically typed SIMPLE, almost all language constructs are strict now, since we want each to type its arguments almost all the time. Like in the other two KOOL definitions, we prefer to copy and then modify/extend the syntax of statically typed SIMPLE.

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

Types

SYNTAX *Type* ::= `void`
| `int`
| `bool`
| `string`
| *Id* [notInRules, klabel('class)]
| `class` (*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* [strict(1)]
| *Exp*[*Exps*] [strict, klabel('_:Exp[_:Exps])]
| *Exp*(*Exps*) [strict]
| `-` *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]
| *Exp* `||` *Exp* [strict]
| `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]
| `if` (*Exp*)*Block*
| `while` (*Exp*)*Block* [strict]
| `for` (*Stmts* *Exp* `;` *Exp*)*Block*
| `return` *Exp* `;` [strict]
| `return ;`
| `print` (*Exps*) `;` [strict]
| `try` *Block* `catch` (*Param*)*Block* [strict(1)]
| `throw` *Exp* `;` [strict]
| `join` *Exp* `;` [strict]
| `acquire` *Exp* `;` [strict]
| `release` *Exp* `;` [strict]
| `rendezvous` *Exp* `;` [strict]

SYNTAX *Stmts* ::= *Stmt*
| *Stmts* *Stmts* [seqstrict]

Desugaring macros

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

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

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

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

RULE $\dfrac{\texttt{class}\ C\ \ S}{\texttt{class}\ C\ \texttt{extends theObject}\ \ S}$

END MODULE

Static semantics

We first discuss the configuration, then give the static semantics taken over unchanged from SIMPLE, then discuss the static semantics of SIMPLE syntactic constructs that needs to change, and in the end we discuss the static semantics and additional checks specifically related to the KOOL proper syntax.

MODULE KOOL-TYPED-STATIC

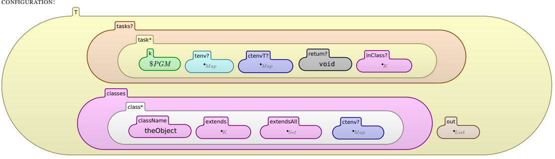
Configuration

The configuration of our type system consists of a `tasks` cell with the same meaning like in statically typed SIMPLE, of an `out` cell streamed to the standard output that will be used to display typing error messages, and of a cell `classes` holding data about each class in a separate `class` cell. The `task` cells now have two additional optional subcells, namely `ctenvT` and `inClass`. The former holds a temporary class type environment; its contents will be transferred into the `ctenv` cell of the corresponding class as soon as all the fields and methods in the task are processed. In fact, there will be three types of tasks in the subsequent semantics, each determined by the subset of cells that it holds:

1. *Main task*, holding only a `k` cell holding the original program as a set of classes. The role of this task is to process each class, generating a class task (see next) for each.
2. *Class task*, holding `k`, `ctenvT`, and `inClass` subcells. The role of this task type is to process a class' contents, generating a class type environment in the `ctenvT` cell and a method task (see next) for each method in the class. To avoid interference with object member lookup rules below, it is important to add the class type environment to a class atomically; this is the reason for which we use `ctenvT` temporary cells within class tasks (instead of adding each member incrementally to the class' type environment).
3. *Method task*, holding `k`, `tenv` and `return` cells. These tasks are similar to SIMPLE's function tasks, so we do not discuss them here any further.

Each `class` cell hods its name (in the `className` cell) and the name of the class it extends (in the `extends` cell), as well as its type environment (in the `ctenv` cell) and the set of all its superclasses (in the `extendsAll` cell). The later is useful for example for checking whether there are cycles in the class extends relation.

CONFIGURATION:



Unchanged semantics from statically typed SIMPLE

The syntax and rules below are borrowed unchanged from statically typed SIMPLE, so we do not discuss them much here.

SYNTAX *Exp* ::= *Type*

SYNTAX *BlockOrStmtType* ::= `block`
| `stmt`

SYNTAX *Type* ::= *BlockOrStmtType*

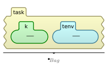
SYNTAX *Block* ::= *BlockOrStmtType*

SYNTAX *KResult* ::= *Type*

CONTEXT — —[□] ;

RULE $\dfrac{T\ \ E[\,\texttt{int}\,\texttt{,}\ Ts]\ \texttt{;}}{T[]\ \ E[Ts]\ \texttt{;}}$

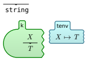
RULE $\dfrac{T\ \ E[\bullet_{Types}]\ \texttt{;}}{T\ \ E\ \texttt{;}}$

RULE


RULE $\dfrac{—}{\texttt{int}}$

RULE $\dfrac{—}{\texttt{bool}}$

RULE $\dfrac{—}{\texttt{string}}$

RULE


CONTEXT `++` $\dfrac{□}{\texttt{ltype}\ (□)}$

RULE $\dfrac{\texttt{++ int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int + int}}{\texttt{int}}$

RULE $\dfrac{\texttt{string + string}}{\texttt{string}}$

RULE $\dfrac{\texttt{int - int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int}\ *\ \texttt{int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int / int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int \% int}}{\texttt{int}}$

RULE $\dfrac{\texttt{- int}}{\texttt{int}}$

RULE $\dfrac{\texttt{int < int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int <= int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int > int}}{\texttt{bool}}$

RULE $\dfrac{\texttt{int >= int}}{\texttt{bool}}$

RULE $\dfrac{T\ \texttt{==}\ T}{\texttt{bool}}$

RULE $\dfrac{T\ \texttt{!=}\ T}{\texttt{bool}}$

RULE $\dfrac{\texttt{bool \&\& bool}}{\texttt{bool}}$

RULE $\dfrac{\texttt{bool || bool}}{\texttt{bool}}$

RULE $\dfrac{\texttt{! bool}}{\texttt{bool}}$

RULE $\dfrac{(T[])[\,\texttt{int}\,\texttt{,}\ Ts]}{T[Ts]}$

RULE $\dfrac{T[\bullet_{Types}]}{T}$

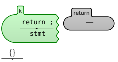
RULE $\dfrac{\texttt{sizeOf}\ (T[])}{\texttt{int}}$

RULE $\dfrac{\texttt{read}\ ()}{\texttt{int}}$

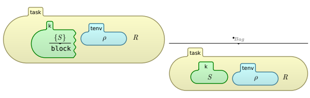
RULE $\texttt{print}\ (\underbrace{T\,\texttt{,}\ Ts}_{Ts})\ \texttt{;}$ requires $T =_K \texttt{int} \vee_{Bool} T =_K \texttt{string}$

RULE $\dfrac{\texttt{print}\ (\bullet_{Types})\ \texttt{;}}{\texttt{stmt}}$

CONTEXT $\dfrac{□}{\texttt{ltype}\ (□)}$ `=` —

RULE


RULE $\dfrac{\{\}}{\texttt{block}}$

RULE


RULE $\dfrac{—\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{if (bool) block else block}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{while (bool) block}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{join int ;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{acquire}\ \text{—}\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{release}\ \text{—}\ \texttt{;}}{\texttt{stmt}}$

RULE $\dfrac{\texttt{rendezvous}\ \text{—}\ \texttt{;}}{\texttt{stmt}}$

SYNTAX $\mathit{Stmt} ::= \mathit{BlockOrStmtType}$

RULE $\dfrac{\text{—}\ \text{—}}{\texttt{stmt}}$

Unchanged auxiliary operations from dynamically typed SIMPLE.

SYNTAX $\mathit{Decl} ::= \texttt{mkDecls}\ (\mathit{Params})$ [function]

RULE $\dfrac{\texttt{mkDecls}\ (T\ \ X\ \texttt{,}\ Ps)}{T\ \ X\ \texttt{;}\ \ \texttt{mkDecls}\ (Ps)}$

RULE $\dfrac{\texttt{mkDecls}\ (\bullet_{Params})}{\{\}}$

SYNTAX $\mathit{LValue} ::= \mathit{Id}$

RULE $\dfrac{isLValue(\text{—}[\text{—}])}{\texttt{true}}$

SYNTAX $\mathit{Exp} ::= \texttt{ltype}\ (\mathit{Exp})$

CONTEXT $\texttt{ltype}\ (\square)$ requires $isLValue(\square)$

SYNTAX $\mathit{Types} ::= \texttt{getTypes}\ (\mathit{Params})$ [function]

RULE $\dfrac{\texttt{getTypes}\ (T\ \ \text{—})}{T\ \texttt{,}\ \bullet_{Types}}$

RULE $\dfrac{\texttt{getTypes}\ (T\ \ \text{—}\ \texttt{,}\ P\ \texttt{,}\ Ps)}{T\ \texttt{,}\ \texttt{getTypes}\ (P\ \texttt{,}\ Ps)}$

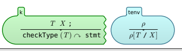
RULE $\dfrac{\texttt{getTypes}\ (\bullet_{Params})}{\texttt{void}\ \texttt{,}\ \bullet_{Types}}$

Changes to the existing statically typed SIMPLE semantics

Below we give the new static semantics for language constructs that come from SIMPLE, but whose SIMPLE static semantics was too restrictive or too permissive and thus had to change.

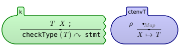
Local variable declration. Since we can define new types in KOOL (corresponding to classes), the variable declaration needs to now check that the claimed types exist. The operation `checkType`, defined at the end of this module, checks whether the argument type is correct (it actually works with lists of types as well).

RULE



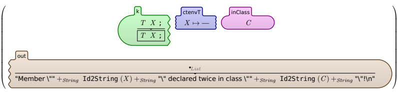
Class member declaration. In class tasks, variable declarations mean class member declarations. Since we reduce method declarations to variable declarations (see below), a variable declaration in a class task can mean either a field or a method declaration. Unlike local variable declarations, which can shadow previous homonymous local or member declarations, member declarations are regarded as a set, so we disallow multiple declarations for the same member (one could improve upon this, like in Java, by treating members with different types or number of arguments as different, etc., but we do not do it here). We also issue an error message if one attempts to redeclare the same class member. The framed variable declaration in the second rule below should be read "stuck". In fact, it is nothing but a unary operation called `stuck`, which takes a $\mathbb{K}$-term as argument and does nothing with it; this `stuck` operation is displayed as a frame in this PDF document because of its latex attribute (see the ASCII .k file, at the end of this module).

RULE



requires $\neg_{Bool}(X\ \texttt{in}\ \texttt{keys}\ (\rho))$

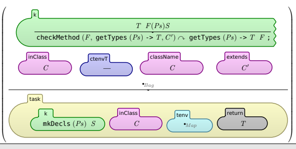
RULE


k
T X ;
T X ;
ctenvT
X ↦ —
inClass
C
out
•List
"Member \"" +String Id2String (X) +String "\" declared twice in class \"" +String Id2String (C) +String "\"!\n"

[structural]

Method declaration. A method declaration requires two conceptual checks to be performed: first, that the method's type is consistent with the type of the homonymous method that it overrides, if any; and second, that its body types correctly. At the same time, it should also be added to the type environment of its class. The first conceptual task is performed using the `checkMethod` operation defined below, and the second by generating a corresponding method task. To add it to the class type environment, we take advantage of the fact that KOOL is higher order and reduce the problem to a field declaration problem, which we have already defined. The role of the `ctenvT` cell in the rule below is to structurally ensure that the method declaration takes place in a class task (we do not want to allow methods to be declared, for example, inside other methods).

RULE



[structural]

Assignment. A more concrete value is allowed to be assigned to a more abstract variable. The operation `checkSubtype` is defined at the end of the module and it also works with pairs of lists of types.

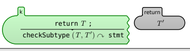
RULE $\dfrac{T\ \texttt{=}\ T'}{\texttt{checkSubtype}\ (T', T) \curvearrowright T}$

Method invocation and return. Methods can be applied on values of more concrete types than their arguments:

RULE $\dfrac{(Ts\ \texttt{->}\ T)(Ts')}{\texttt{checkSubtype}\ (Ts', Ts) \curvearrowright T}$

Similarly, we allow values of more concrete types to be returned by methods:

RULE



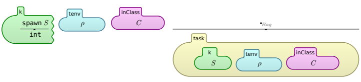
Exceptions. Exceptions can throw and catch values of any types. Since unlike in Java KOOL's methods do not declare the exception types that they can throw, we cannot test the full type safety of exceptions. Instead, we only check that the `try` and the `catch` statements type correctly.

RULE $\dfrac{\texttt{try block catch}\ (T\ \ X)S}{\{T\ \ X\ \texttt{;}\ \ S\}}$ [structural]

RULE $\dfrac{\texttt{throw}\ T\ \texttt{;}}{\texttt{stmt}}$

Spawn. The spawned cell needs to also be passed the parent's class.

RULE

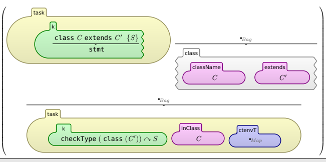


Semantics of the new KOOL constructs

Class declaration. We process each class in the main task, adding the corresponding data into its `class` cell and also adding a class task for it. We also perform some well-formedness checks on the class hierarchy.

Initiate class processing We create a class cell and a class task for each task. Also, we start the class task with a check that the class it extends is declared (this delays the task until that class is processed using another instance of this rule).

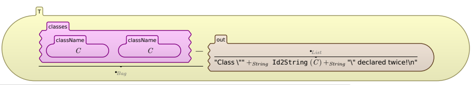
RULE



[structural]

Check for unique class names

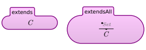
RULE


T
classes
className
C
className
C
•Bag
—
out
•List
"Class \"" +String Id2String (C) +String "\" declared twice!\n"

[structural]

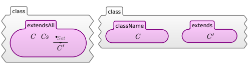
Check for cycles in class hierarchy We check for cycles in the class hierarchy by transitively closing the class extends relation using the `extendsAll` cells, and checking that a class will never appear in its own `extendsAll` cell. The first rule below initiates the transitive closure of the superclass relation, the second transitively closes it, and the third checks for cycles.

RULE



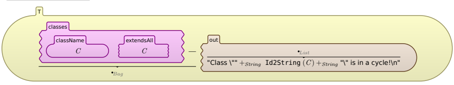
[structural]

RULE



requires $\neg_{Bool}(C'\ \texttt{in}\ C\ \ Cs)$ [structural]

RULE


T
classes
className
C
extendsAll
C
•Bag
—
out
•List
"Class \"" +String Id2String (C) +String "\" is in a cycle!\n"

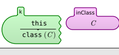
[structural]

New. To type `new` we only need to check that the class constructor can be called with arguments of the given types, so we initiate a call to the constructor method in the corresponding class. If that succeeds, meaning that it types to `stmt`, then we discard the `stmt` type and produce instead the corresponding class type of the new object. The auxiliary `discard` operation is defined also at the end of this module.

RULE $\dfrac{\texttt{new}\ C(Ts)}{\texttt{class}\ (C)\ \texttt{.}\ C(Ts) \curvearrowright \texttt{discard} \curvearrowright \texttt{class}\ (C)}$

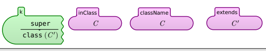
Self reference. The typing rule for `this` is straightforward: reduce to the current class type.

RULE



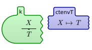
Super. Similarly, `super` types to the parent class type. Note that for typing concerns, super can be considered as an object (recall that this was not the case in the dynamic semantics).

RULE

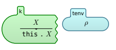


Object member access. There are several cases to consider here. First, if we are in a class task, we should lookup the member into the temporary class type environemnt in cell `ctenvT`. That is because we want to allow initialized field declarations in classes, such as "`int x=10;`". This is desugared to a declaration of `x`, which is added to `ctenvT` during the class task processing, followed by an assignment of `x` to 10. In order for the assignment to type check, we need to know that `x` has been declared with type `int`; this information can only be found in the `ctenvT` cell. Second, we should redirect non-local variable lookups in method tasks to corresponding member accesses (the local variables are handled by the rule borrowed from SIMPLE). This is what the second rule below does. Third, we should allow object member accesses as lvalues, which is done by the third rule below. These last two rules therefore ensure that each necessary object member access is explicitly allowed for evaluation. Recall from the annotated syntax module above that the member access operation is strict in the object. That means that the object is expected to evaluate to a class type. The next two rules below define the actual member lookup operation, moving the search to the superclass when the member is not found in the current class. Note that this works because we create the class type environments atomically; thus, a class either has its complete type environment available, in which case these rules can safely apply, or its cell `ctenv` is not yet available, in which case these rules have to wait. Finally, the sixth rule below reports an error when the `Object` class is reached.

RULE



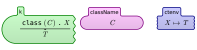
RULE



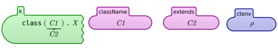
requires $\neg_{Bool}(X\ \texttt{in}\ \texttt{keys}\ (\rho))$

RULE $\dfrac{isLValue(\text{—}\ \texttt{.}\ \text{—})}{\texttt{true}}$

RULE

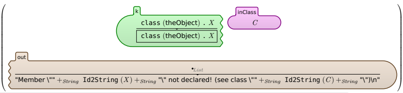


RULE



requires $\neg_{Bool}(X\ \texttt{in}\ \texttt{keys}\ (\rho))$ [structural]

RULE


k
class (theObject) . X
class (theObject) . X
inClass
C
out
•List
"Member \"" +String Id2String (X) +String "\" not declared! (see class \"" +String Id2String (C) +String "\")\n"

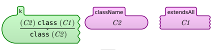
[structural]

Instance of and casting. As it is hard to check statically whether casting is always safe, the programmer is simply trusted from a typing perspective. We only do some basic upcasting and downcasting checks, to reject casts which will absolutely fail. However, dynamic semantics or implementations of the language need to insert runtime checks for downcasting to be safe.

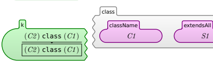
RULE $\dfrac{\texttt{class}\ (C1)\ \texttt{instanceOf}\ C2}{\texttt{bool}}$

RULE $\dfrac{(C)\ \texttt{class}\ (C)}{\texttt{class}\ (C)}$

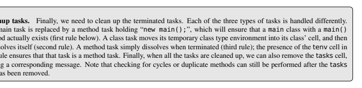
RULE



RULE



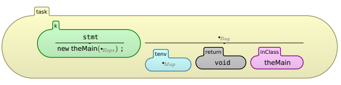
RULE


k
(C2) class (C1)
(C2) class (C1)
class
className
C1
extendsAll
S1
class
className
C2
extendsAll
S2
out
•List
"Classes \"" +String Id2String (C1) +String "\" and \"" +String Id2String (C2) +String "\" are incompatible!\n"

requires $\neg_{Bool}(C1\ \texttt{in}\ S2) \wedge_{Bool} \neg_{Bool}(C2\ \texttt{in}\ S1)$

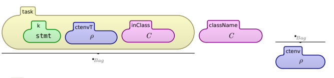
Cleanup tasks. Finally, we need to clean up the terminated tasks. Each of the three types of tasks is handled differently. The main task is replaced by a method task holding "`new main();`", which will ensure that a `main` class with a `main()` method actually exists (first rule below). A class task moves its temporary class type environment into its class' cell, and then it dissolves itself (second rule). A method task simply dissolves when terminated (third rule); the presence of the `tenv` cell in that rule ensures that that task is a method task. Finally, when all the tasks are cleaned up, we can also remove the `tasks` cell, issuing a corresponding message. Note that checking for cycles or duplicate methods can still be performed after the `tasks` cell has been removed.

RULE



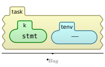
[structural]

RULE

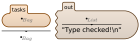


[structural]

RULE



[structural]

RULE 

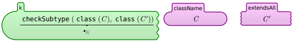
[structural]

KOOL-specific auxiliary declarations and operations

Subtype checking. The subclass relation introduces a subtyping relation.

SYNTAX $\mathit{KItem} ::= \mathtt{checkSubtype}\,(\mathit{Types}, \mathit{Types})$

RULE $\dfrac{\mathtt{checkSubtype}\,(T, T)}{\bullet_K}$ [structural]

RULE 

[structural]

RULE $\dfrac{\mathtt{checkSubtype}\,(Ts1 \;\mathtt{->}\; T2, Ts1' \;\mathtt{->}\; T2')}{\mathtt{checkSubtype}\,(((T2), Ts1'), ((T2'), Ts1))}$ [structural]

RULE $\dfrac{\mathtt{checkSubtype}\,((T, Ts), (T', Ts'))}{\mathtt{checkSubtype}\,(T, T') \curvearrowright \mathtt{checkSubtype}\,(Ts, Ts')}$ requires $Ts \neq_K \bullet_{Types}$ [structural]

RULE $\dfrac{\mathtt{checkSubtype}\,(\bullet_{Types}, \bullet_{Types})}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkSubtype}\,(\bullet_{Types}, \mathtt{void})}{\bullet_K}$ [structural]

Checking well-formedness of types. Since now any *Id* can be used as the type of a class, we need to check that the types used in the program actually exists

SYNTAX $\mathit{KItem} ::= \mathtt{checkType}\,(\mathit{Types})$

RULE $\dfrac{\mathtt{checkType}\,(T, Ts)}{\mathtt{checkType}\,(T) \curvearrowright \mathtt{checkType}\,(Ts)}$ requires $Ts \neq_K \bullet_{Types}$ [structural]

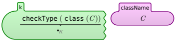
RULE $\dfrac{\mathtt{checkType}\,(\bullet_{Types})}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkType}\,(\,\mathtt{int})}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkType}\,(\,\mathtt{bool})}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkType}\,(\,\mathtt{string})}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkType}\,(\,\mathtt{void})}{\bullet_K}$ [structural]

RULE 

[structural]

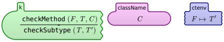
RULE $\dfrac{\mathtt{checkType}\,(\,\mathtt{class}\,(\mathsf{theObject}))}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{checkType}\,(Ts \;\mathtt{->}\; T)}{\mathtt{checkType}\,(T, Ts)}$ [structural]

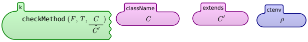
RULE $\dfrac{\mathtt{checkType}\,(T[\,])}{\mathtt{checkType}\,(T)}$ [structural]

Checking correct overiding of methods. The `checkMethod` operation below searches to see whether the current method overrides some other method in some superclass. If yes, then it issues an additional check that the new method's type is more concrete than the overridden method's. The types T and T' below can only be function types. See the definition of `checkSubtype` on function types at the end of this module (it is co-variant in the codomain and contra-variant in the domain).

SYNTAX $\mathit{KItem} ::= \mathtt{checkMethod}\,(\mathit{Id}, \mathit{Type}, \mathit{Id})$

RULE 

[structural]

RULE 

requires $\neg_{Bool}(F \;\mathtt{in}\; \mathtt{keys}\,(\rho))$ [structural]

RULE $\dfrac{\mathtt{checkMethod}\,(\text{—}, \text{—}, \mathsf{theObject})}{\bullet_K}$ [structural]

Generic operations which could be part of the $\mathbb{K}$ framework.

SYNTAX $\mathit{KItem} ::= \boxed{K}$

SYNTAX $\mathit{KItem} ::= \mathtt{discard}$

RULE $\dfrac{\text{—} \curvearrowright \mathtt{discard}}{\bullet_K}$ [structural]

RULE $\dfrac{\mathtt{Id2String}\,(\mathsf{theMain})}{\text{"Main"}}$

RULE $\dfrac{\mathtt{Id2String}\,(\mathsf{theObject})}{\text{"Object"}}$

END MODULE