MODULE KOOL-TYPED-STATIC-SYNTAX

SYNTAX *#Id* ::= `main`

> KOOL addition: the "object" class

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

> KOOL addition: types for objects

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

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

SYNTAX *Decl* ::= `method` *#Id* (*Exps*) : *Type* *Stmt*
| `class` *#Id* { *Stmts* }
| `class` *#Id* `extends` *#Id* { *Stmts* }
SYNTAX *Exp* ::= *#Int*
| *#Bool*
| *#String*
| *#Id*
| ++ *Exp*
| *Exp* + *Exp* [strict]
| *Exp* - *Exp* [strict]
| *Exp* * *Exp* [strict]
| *Exp* / *Exp* [strict]
| *Exp* % *Exp* [strict]
| - *Exp* [strict]
| *Exp* < *Exp* [strict]
| *Exp* <= *Exp* [strict]
| *Exp* > *Exp* [strict]
| *Exp* >= *Exp* [strict]
| *Exp* == *Exp* [strict]
| *Exp* != *Exp* [strict]
| *Exp* `and` *Exp* [strict]
| *Exp* `or` *Exp* [strict]
| `not` *Exp* [strict]
| *Exp* [*Exps*] [strict]
| `sizeOf(` *Exp* `)` [strict]
| *Exp* (*Exps*) [strict]
| `read()`
| *Exp* = *Exp* [strict(2)]
| *Exp* : *Type*
| `new` *#Id* (*Exps*) [strict(2)]
| `this`
| `super`
| *Exp* . *#Id*
| *Exp* `instanceOf` *#Id* [strict(1)]
| `cast` *Exp* `to` *#Id* [strict(1)]
SYNTAX *Stmt* ::= {}
| { *Stmts* }
| *Exp* ; [strict]
| `if` *Exp* `then` *Stmt* `else` *Stmt* [strict]
| `if` *Exp* `then` *Stmt*
| `while` *Exp* `do` *Stmt* [strict]
| `for` *#Id* = *Exp* `to` *Exp* `do` *Stmt*
| `return` *Exp* ; [strict]
| `return;`
| `print(` *Exps* `);` [strict]
| `try` *Stmt* `catch(` *#Id* `)` *Stmt* [strict(1)]
| `throw` *Exp* ; [strict]
| `spawn` *Stmt* [strict]
| `acquire` *Exp* ; [strict]
| `release` *Exp* ; [strict]
| `rendezvous` *Exp* ; [strict]
SYNTAX *Stmts* ::= *Decl*
| *Stmt*
| *Stmts* *Stmts* [seqstrict]

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

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

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

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

END MODULE

MODULE KOOL-TYPED-STATIC

IMPORTS KOOL-TYPED-STATIC-SYNTAX

SYNTAX *Exp* ::= *Type*
SYNTAX *Stmt* ::= *Type*
SYNTAX *KResult* ::= *Type*

CONFIGURATION:

T: [classes: $PGM] [class +: [className: object] [extends ?: object] [ctenv ?: •] [task *: [k: •] [tenv: •] [return ?: •] [pass: classPass]]]

SYNTAX *Pass* ::= `classPass`
| `methodPass`
SYNTAX *K* ::= *Pass*
SYNTAX *Type* ::= `stmt`

RULE [classes: `class` C `extends` C' { Ss } ⇒ • ...] [• ⇒ class: [className: C] [extends: C'] [task: [k: Ss] [pass: classPass] [tenv: •]]]

RULE [classes: •] ⇒ •

RULE `var` X : T ; ⇒ `bindto(` X , T `)`

CONTEXT: `var` – [☐] : – ;

RULE `var` X [Ts] : T ; ⇒ `checkDepth(` Ts , T `)` ↷ `bindto(` X , T `)`

RULE [k: `method` M (XTs) : T S ⇒ `bindto(` M , `types(` XTs `)` → T `)` ...] [pass: classPass] [• ⇒ task: [k: `var` XTs ; S] [tenv: •] [return: T] [pass: methodPass]]

RULE [task ⇒ •: [k: •] [tenv: ρ] [pass: classPass] ...] [• ⇒ ctenv: ρ]

RULE [k: I ⇒ `int` ...]

RULE [k: B ⇒ `bool` ...]

RULE [k: Str ⇒ `string` ...]

RULE [k: X ⇒ T ...] [tenv: ... X ↦ T ...] [pass: methodPass]

RULE [k: X ⇒ `this` . X ...] [tenv: ρ] [pass: methodPass]   when $\neg_{Bool}$ X `in keys` ρ

CONTEXT: ++ (☐ ⇒ `l-type(` ☐ `)`)

RULE ++ `int` ⇒ `int`

RULE `int` + `int` ⇒ `int`

RULE `string` + `string` ⇒ `string`

RULE `int` - `int` ⇒ `int`

RULE `int` * `int` ⇒ `int`

RULE `int` / `int` ⇒ `int`

RULE `int` % `int` ⇒ `int`

RULE - `int` ⇒ `int`

RULE `int` < `int` ⇒ `bool`

RULE `int` <= `int` ⇒ `bool`

RULE `int` > `int` ⇒ `bool`

RULE `int` >= `int` ⇒ `bool`

RULE T == T ⇒ `bool`

RULE T != T ⇒ `bool`

RULE `bool and bool` ⇒ `bool`

RULE `bool or bool` ⇒ `bool`

RULE `not bool` ⇒ `bool`

RULE `array of` T [`int` , Ts] ⇒ T [Ts]

RULE T [] ⇒ T

RULE `sizeOf( array of` T `)` ⇒ `int`

RULE [k: Ts → T (Ts') ⇒ T `when subtype(` Ts' , Ts `)` ...] [pass: methodPass]

RULE `read()` ⇒ `int`

CONTEXT: (☐ ⇒ `l-type(` ☐ `)`) = –

RULE T = T' ⇒ T `when subtype(` T' , T `)`

> **"new" Operator**
>
> To type "new" we only need to check that the class constructor can be called with arguments of the given type.

RULE `new` C (Ts) ⇒ `class` C . C (Ts)

> "this" types to the current class

RULE [k: `this` ⇒ `class` C ...] [className: C]

> "super" types to the parent class. Note that for typing concerns super can be considered as an object

RULE [k: `super` ⇒ `class` C' ...] [className: C] [extends: C']

> **Object member access**
>
> Case 1: member declared in current class:

RULE [k: `class` C . X ⇒ T ...] [className: C] [ctenv: ... X ↦ T ...]

RULE [k: `class` C . X ⇒ T ...] [class: ... [className: C] [ctenv: ... X ↦ T ...] ...]

> Case 2: member not declared in current class; check parent class. This currently assumes there are no cycles in the class hierarchy.

RULE [k: `class` (C_1 ⇒ C_2) . X ...] [className: C_1] [extends: C_2] [ctenv: ρ]   when $\neg_{Bool}$ X `in keys` ρ [structural]

RULE [k: `class` (C_1 ⇒ C_2) . X ...] [class: ... [className: C_1] [extends: C_2] [ctenv: ρ] ...]   when $\neg_{Bool}$ X `in keys` ρ [structural]

RULE `class` C_1 `instanceOf` C_2 ⇒ `bool`

RULE `cast class` C_1 `to` C_2 ⇒ `class` C_2

RULE {} ⇒ `stmt`

RULE [k: { Ss } ⇒ Ss ↷ `tenv(` ρ `)` ...] [tenv: ρ]

RULE T ; ⇒ `stmt`

RULE `if bool then stmt else stmt` ⇒ `stmt`

RULE `while bool do stmt` ⇒ `stmt`

RULE [k: `return` T ; ⇒ `stmt when subtype(` T , T' `)` ...] [return: T']

RULE `return;` ⇒ `stmt`

RULE `print(` (`int` , Ts ⇒ Ts) `);`

RULE `print(` (`string` , Ts ⇒ Ts) `);`

RULE `print();` ⇒ `stmt`

RULE `try stmt catch(` X `)` S ⇒ { `var` X : `int` ; S }   [structural]

RULE `throw int ;` ⇒ `stmt`

RULE `spawn stmt` ⇒ `stmt`

RULE `acquire` T ; ⇒ `stmt`

RULE `release` T ; ⇒ `stmt`

RULE `rendezvous` T ; ⇒ `stmt`

RULE `stmt stmt` ⇒ `stmt`

> **Clean-Up and Auxiliary Operations**

RULE [task: [k: `stmt`] [pass: methodPass] ...] ⇒ •

RULE [class: [className: C_1] [extends: C_2] [ctenv: –]] ⇒ •

SYNTAX *Exp* ::= *Type* `when` *Exp* [strict(2)]
RULE T `when true` ⇒ T

SYNTAX *Exp* ::= `subtype(` *Types* , *Types* `)`
RULE `subtype(,)` ⇒ `true`

RULE `subtype(` T , Ts , T' , Ts' `)` ⇒ `subtype(` T , T' `) and subtype(` Ts , Ts' `)`

RULE `subtype(` T , T `)` ⇒ `true`

RULE [k: `subtype( class` C_1 , `class` C `)` ⇒ `subtype( class` C_2 , `class` C `)` ...] [class: ... [className: C_1] [extends: C_2] ...]

RULE [k: `subtype( class` C_1 , `class` C `)` ⇒ `subtype( class` C_2 , `class` C `)` ...] [className: C_1] [extends: C_2]

SYNTAX *Exp* ::= `l-type(` *Exp* `)`
RULE `l-type(` X `)` ⇒ X [structural]

CONTEXT: `l-type(` – [☐] `)`

CONTEXT: `l-type(` ☐ [–] `)`

RULE `l-type(` T `)` ⇒ T [structural]

SYNTAX *K* ::= `checkDepth(` *Types* , *Type* `)`
RULE `checkDepth(` (`int` , Ts ⇒ Ts) , (`array of` T ⇒ T) `)`   [structural]

RULE `checkDepth(` • , – `)` ⇒ •   [structural]

SYNTAX *K* ::= `bindto(` *#Id* , *Type* `)`
RULE [k: `bindto(` X , T `)` ⇒ `existsType(` T `)` ...] [tenv: ρ (• ⇒ X ↦ T)] [pass: classPass]   when $\neg_{Bool}$ X `in keys` ρ

RULE [k: `bindto(` X , T `)` ⇒ `existsType(` T `)` ↷ `stmt` ...] [tenv: ρ ⇒ ρ [T / X]] [pass: methodPass]

SYNTAX *K* ::= `tenv(` *Map* `)`
RULE [k: `stmt` ↷ (`tenv(` ρ `)` ⇒ •) ...] [tenv: – ⇒ ρ]   [structural]

SYNTAX *Types* ::= `types(` *Exps* `)`
RULE `types()` ⇒ •   [structural]

RULE `types(` X : T , XTs `)` ⇒ T , `types(` XTs `)`   [structural]

SYNTAX *K* ::= `existsType(` *Type* `)`
RULE `existsType( int )` ⇒ `true`

RULE `existsType( bool )` ⇒ `true`

RULE `existsType( string )` ⇒ `true`

RULE `existsType( void )` ⇒ `true`

RULE [k: `existsType( class` C `)` ⇒ `true` ...] [class: ... [className: C] ...]

RULE [k: `existsType( class` C `)` ⇒ `true` ...] [className: C]

END MODULE