KOOL-TYPED-DYNAMIC

MODULE KOOL-TYPED-DYNAMIC-SYNTAX

SYNTAX *Id* ::= `main`

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

SYNTAX *Types* ::= $List\{Type, “,”\}$ [strict]

SYNTAX *Decl* ::= `var` *Exps* `;`

SYNTAX *Exp* ::= *Int*
| *Bool*
| *String*
| *Id*
| `this`
| `super`
| (*Exp*)
| `++` *Exp*
| *Exp* `instanceOf` *Id* [strict(1)]
| `cast` *Exp* `to` *Id* [strict(1)]
| `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, non-assoc]
| *Exp* `<=` *Exp* [strict, non-assoc]
| *Exp* `>` *Exp* [strict, non-assoc]
| *Exp* `>=` *Exp* [strict, non-assoc]
| *Exp* `==` *Exp* [strict, non-assoc]
| *Exp* `!=` *Exp* [strict, non-assoc]
| `not` *Exp* [strict]
| *Exp* `and` *Exp* [strict]
| *Exp* `or` *Exp* [strict]
| *Exp* `=` *Exp* [strict(2)]
| *Exp* `:` *Type*

SYNTAX *Exps* ::= $List\{Exp, “,”\}$

SYNTAX *Stmt* ::= {}
| {*Stmts*}
| *Exp* `;` [strict]
| `if` *Exp* `then` *Stmt* `else` *Stmt* [avoid, strict(1)]
| `if` *Exp* `then` *Stmt*
| `while` *Exp* `do` *Stmt*
| `for` *Id* `=` *Exp* `to` *Exp* `do` *Stmt*
| `return` *Exp* `;` [strict]
| `return;`
| `print` (*Exps*) `;` [strict]
| `throw` *Exp* `;` [strict]
| `spawn` *Stmt*
| `acquire` *Exp* `;` [strict]
| `release` *Exp* `;` [strict]
| `rendezvous` *Exp* `;` [strict]

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

RULE `if` E `then` S ⇒ `if` E `then` S `else` {}

RULE `for` X = $E1$ `to` $E2$ `do` S ⇒ {`var` '_:_(X,`int`) = $E1$, $\bullet_{Exps}$; `while` X <= $E2$ `do` {S X = X + 1 ;}}

RULE `var` $E1$, $E2$, Es ; ⇒ `var` $E1$, $\bullet_{Exps}$; `var` $E2$, Es ;

RULE `var` '_:_(X, T) = E , $\bullet_{Exps}$; ⇒ `var` '_:_(X, T) , $\bullet_{Exps}$; X = E ;

SYNTAX *Id* ::= `object`

SYNTAX *Type* ::= `class` *Id*

SYNTAX *Decl* ::= `method` *Id*(*Exps*) `:` *Type* *Stmt*
| `method` *Id*(*Exps*)*Stmt*

RULE `method` X(Es)S ⇒ `method` X(Es) : `void` S

SYNTAX *Decl* ::= `class` *Id*{*Stmts*}
| `class` *Id* `extends` *Id*{*Stmts*}

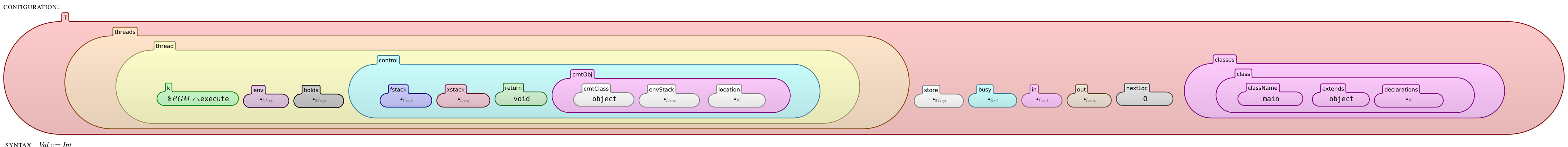
RULE `class` C{Ss} ⇒ `class` C `extends` `object`{Ss}

SYNTAX *Stmt* ::= `try` *Stmt* `catch` (*Exp*)*Stmt*

END MODULE

MODULE KOOL-TYPED-DYNAMIC

CONFIGURATION:



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

SYNTAX *Vals* ::= $List\{Val, “,”\}$

SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

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

RULE k: `var` '_:_(X, T), $\bullet_{Exps}$; ⇒ $\bullet_K$ | env: Env ⇒ $Env[L / X]$ | store: $\bullet_{Map}$ ⇒ $L \mapsto \bot_T$ | nextLoc: L ⇒ $L +_{Int} 1$

RULE k: `var` '_:_($X[N, \bullet_{Vals}]$, T`[]`), $\bullet_{Exps}$; ⇒ $\bullet_K$ | env: Env ⇒ $Env[L / X]$ | store: $\bullet_{Map}$ ⇒ $L \mapsto$ `arrayRef` (T`[]`, $L +_{Int} 1$, N) $L +_{Int} 1 .. L +_{Int} N \mapsto \bot_T$ | nextLoc: L ⇒ $L +_{Int} 1 +_{Int} N$

CONTEXT `var` '_:_(X[□], T), $\bullet_{Exps}$;

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

RULE `var` '_:_($X[N1, N2, Vs]$, T`[][]`), $\bullet_{Exps}$; ⇒ `var` $X[N1, \bullet_{Exps}]$: T`[][]`, $\bullet_{Exps}$; {`var` `$1` : T`[][]` = X, $\bullet_{Exps}$; `for` `$2` = 0 `to` $N1$ - 1 `do` {`var` $X[N2, Vs]$: T`[]`, $\bullet_{Exps}$; `$1`[`$2`, $\bullet_{Exps}$] = X ;}}

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

CONTEXT `++` (□ ⇒ `lvalue` (□))

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

RULE $I1$ + $I2$ ⇒ $I1 +_{Int} I2$

RULE $Str1$ + $Str2$ ⇒ $Str1 +_{String} Str2$

RULE $I1$ - $I2$ ⇒ $I1 -_{Int} I2$

RULE $I1$ * $I2$ ⇒ $I1 *_{Int} I2$

RULE $I1$ / $I2$ ⇒ $I1 \div_{Int} I2$ when $I2$ =/=Int 0

RULE $I1$ % $I2$ ⇒ $I1 \%_{Int} I2$ when $I2$ =/=Int 0

RULE - I ⇒ $0 -_{Int} I$

RULE $I1$ < $I2$ ⇒ $I1 <_{Int} I2$

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

RULE $I1$ > $I2$ ⇒ $I1 >_{Int} I2$

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

RULE $V1$ == $V2$ ⇒ $V1$ ==K $V2$

RULE $V1$!= $V2$ ⇒ $V1$ =/=K $V2$

RULE $B1$ `and` $B2$ ⇒ $B1 \wedge_{Bool} B2$

RULE $B1$ `or` $B2$ ⇒ $B1 \vee_{Bool} B2$

RULE `not` B ⇒ $\neg_{Bool} B$

RULE $V[N1, N2, Vs]$ ⇒ $V[N1, \bullet_{Exps}][N2, Vs]$

RULE `arrayRef` (—, L, M)[N, $\bullet_{Exps}$] ⇒ `lookup` ($L +_{Int} N$)   when $N \geq_{Int} 0 \wedge_{Bool} N <_{Int} M$

RULE `sizeOf` (`arrayRef` (—, —, N)) ⇒ N

SYNTAX *Val* ::= `nothing`

RULE `return;` ⇒ `return nothing ;`

RULE k: `read` () ⇒ I | in: I ⇒ $\bullet_{List}$

CONTEXT (□ ⇒ `lvalue` (□)) = —

RULE {} ⇒ $\bullet_K$

RULE k: {Ss} ⇒ Ss ↷`env` (Env) | env: Env

RULE $Ss1$ $Ss2$ ⇒ $Ss1$ ↷ $Ss2$

RULE V ; ⇒ $\bullet_K$

RULE `if true then` S `else` — ⇒ S

RULE `if false then` — `else` S ⇒ S

RULE k: `while` E `do` S ⇒ `if` E `then` {S `while` E `do` S} `else` {}

RULE k: `print` (V , Vs) ; ⇒ `print` (Vs) ; | out: $\bullet_{List}$ ⇒ V when `typeOf` (V) ==K `int` $\vee_{Bool}$ `typeOf` (V) ==K `string`

RULE `print` ($\bullet_{Vals}$) ; ⇒ $\bullet_K$

RULE thread: (k: $\bullet_K$ | holds: H) ⇒ $\bullet_{Bag}$ | busy: $Busy$ ⇒ $Busy$ `-Set keys` H

RULE k: `acquire` V ; ⇒ $\bullet_K$ | holds: $\bullet_{Map}$ ⇒ $V \mapsto 0$ | busy: $Busy$ ($\bullet_{Set}$ ⇒ V)   when $\neg_{Bool} V$ `in` $Busy$

RULE k: `acquire` V ; ⇒ $\bullet_K$ | holds: $V \mapsto$ (N ⇒ $N +_{Int} 1$)

RULE k: `release` V ; ⇒ $\bullet_K$ | holds: $V \mapsto$ (N ⇒ $N -_{Int} 1$)   when $N >_{Int} 0$

RULE k: `release` V ; ⇒ $\bullet_K$ | holds: $V \mapsto 0$ ⇒ $\bullet_{Map}$ | busy: V ⇒ $\bullet_{Set}$

RULE k: `rendezvous` V ; ⇒ $\bullet_K$ | k: `rendezvous` V ; ⇒ $\bullet_K$

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

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

RULE k: `lvalue` (X) ⇒ `loc` (L) | env: $X \mapsto L$

CONTEXT `lvalue` (—[□])

CONTEXT `lvalue` (□[—])

RULE `lvalue` (`lookup` (L)) ⇒ `loc` (L)

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

RULE k: `lookup` (L) ⇒ V | store: $L \mapsto V$

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

RULE k: `env` (Env) ⇒ $\bullet_K$ | env: — ⇒ Env

RULE `env` (—) ↷`env` (—) ⇒ $\bullet_K$

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

RULE `typeOf` (I) ⇒ `int`

RULE `typeOf` (B) ⇒ `bool`

RULE `typeOf` (—) ⇒ `string`

RULE `typeOf` (`arrayRef` (T, —, —)) ⇒ T

RULE `typeOf` ($\bot_T$) ⇒ T

RULE `typeOf` (`nothing`) ⇒ `void`

SYNTAX *Types* ::= `types` (*Exps*) [function, klabel(types)]

RULE `types` ($\bullet_{Exps}$) ⇒ `void`, $\bullet_{Types}$

RULE `types` ('_:_(X, T), $\bullet_{Exps}$) ⇒ T , $\bullet_{Types}$

RULE `types` ('_:_(X, T), E, Es) ⇒ T , `types` (E , Es)

SYNTAX *List{K}* ::= *Int* `..` *Int*

RULE $N1$ `..` $N2$ ⇒ $\bullet_{ListK}$ when $N1 >_{Int} N2$

RULE $N1$ `..` $N2$ ⇒ $N1, N1 +_{Int} 1$ `..` $N2$ when $N1 \leq_{Int} N2$

RULE thread: (k: `spawn` S ⇒ $\bullet_K$ | env: Env | crntObj: Obj) ; $\bullet_{Bag}$ ⇒ thread: (k: S | env: Env | crntObj: Obj)

RULE k: `loc` (L) = V ⇒ `subtype` (`typeOf` (V), $\bullet_{Types}$, `typeOf` (K), $\bullet_{Types}$) ↷`true?`↷`unsafeCast` (V, `typeOf` (K)) | store: $L \mapsto$ (K ⇒ `unsafeCast` (V, `typeOf` (K)))

SYNTAX *ListItem* ::= (*Exp*, *Stmt*, *K*, *Map*, *Bag*)

SYNTAX *K* ::= `popx`

RULE k: `try` $S1$ `catch` ('_:_(X, T))$S2$ ↷ K ⇒ $S1$ ↷`popx` | control: (xstack: $\bullet_{List}$ ⇒ ('_:_(X, T), $S2$, K, Env, C)) C | env: Env

RULE k: `popx` ⇒ $\bullet_K$ | xstack: — ⇒ $\bullet_{List}$

RULE k: `throw` V ;↷ — ⇒ `if subtype` (`typeOf` (V), $\bullet_{Types}$, T, $\bullet_{Types}$) `then` {`var` '_:_(X, T) = V, $\bullet_{Exps}$; $S2$} `else throw` V ; ↷ K | control: (xstack: ('_:_(X, T), $S2$, K, Env, C) ⇒ $\bullet_{List}$) (— ⇒ C) | env: — ⇒ Env

SYNTAX *K* ::= `bindto` (*Exps*, *Vals*)

RULE k: ($\bullet_K$ ⇒ X = V ;) ↷`bindto` ('_:_(X, T), XTs ⇒ XTs, V, Vs ⇒ Vs) | env: Env ⇒ $Env[L / X]$ | store: $\bullet_{Map}$ ⇒ $L \mapsto \bot_T$ | nextLoc: L ⇒ $L +_{Int} 1$

RULE k: `bindto` ($\bullet_{Exps}$, $\bullet_{Vals}$) ⇒ $\bullet_K$

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

RULE k: `class` $Class1$ `extends` $Class2${Ss} ⇒ $\bullet_K$ | classes: $\bullet_{Bag}$ ⇒ class: (className: $Class1$ | extends: $Class2$ | declarations: Ss)

SYNTAX *K* ::= `execute`

RULE k: `execute` ⇒ `new main`($\bullet_{Exps}$) ; | env: $\bullet_{Map}$

RULE k: `method` F(XTs) : T S ⇒ $\bullet_K$ | crntClass: $Class$ | location: OL | env: Env ⇒ $Env[L / F]$ | store: $\bullet_{Map}$ ⇒ $L \mapsto$`methodClosure` ($Class$, OL, XTs, S, (`types` (XTs) `->` T)) | nextLoc: L ⇒ $L +_{Int} 1$

RULE k: 'new_`(\_`)($Class$, Vs) ↷ K ⇒ `create` ($Class$) ↷`storeObj`↷ '_`(\_`)($Class$, Vs) ; `return this` ; | env: Env ⇒ $\bullet_{Map}$ | control: C crntObj: (Obj ⇒ crntClass: `object` | envStack: `object ::` env: $\bullet_{Map}$ | location: L) | return: T ⇒ `class` $Class$ | fstack: …

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

RULE k: `create` ($Class$) ⇒ `create` ($Class1$) ↷`setCrntClass` ($Class$) ↷ Ss ↷`addEnvLayer` | className: $Class$ | extends: $Class1$ | declarations: Ss

RULE k: `create` (`object`) ⇒ $\bullet_K$

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

RULE k: `setCrntClass` ($Class$) ⇒ $\bullet_K$ | crntClass: — ⇒ $Class$

SYNTAX *K* ::= `addEnvLayer`

RULE k: `addEnvLayer` ⇒ $\bullet_K$ | env: Env ⇒ $\bullet_{Map}$ | crntClass: $Class$ | envStack: $\bullet_{List}$ ⇒ $Class$ `::` env: Env

SYNTAX *K* ::= `storeObj`

RULE k: `storeObj` ⇒ $\bullet_K$ | crntObj: Obj (location: L ⇒ $\bullet_{Bag}$) | store: $\bullet_{Map}$ ⇒ $L \mapsto$`objectClosure` (Obj)

RULE k: `this` ⇒ `objectClosure` (Obj) | crntObj: Obj

RULE k: X ⇒ '_._(`this`, X) | env: Env when $\neg_{Bool} X$ `in keys` Env

CONTEXT '_._(□, —)   when □ =/=K `super`

RULE k: '_._(`objectClosure` (crntClass: $Class$ | envStack: $Class$ `::` $EnvC$ $EStack$), X) ⇒ `lookupMember` (envStack: $Class$ `::` $EnvC$ $EStack$, X)

RULE k: '_._(`super`, X) ⇒ `lookupMember` (envStack: $EStack$, X) | crntClass: $Class$ | envStack: $Class$ `::` $EnvC$ $EStack$

RULE k: '_`(\_`)(X ⇒ V, —) | env: $X \mapsto L$ | store: $L \mapsto V$

RULE k: '_`(\_`)(X ⇒ '_._(`this`, X), —) | env: Env when $\neg_{Bool} X$ `in keys` Env

CONTEXT '_`(\_`)('_._(□, —), —)   when □ =/=K `super`

RULE k: '_`(\_`)('_._(`objectClosure` (Obj envStack: $EStack$), X) ⇒ `lookupMember` (envStack: $EStack$, X), —)

RULE k: '_`(\_`)('_._(`super`, X) ⇒ `lookupMember` (envStack: $EStack$, X), —) | crntClass: $Class$ | envStack: $Class$ `::` — $EStack$

CONTEXT '_`(\_`)(□, —)   when `getKLabel` □ ==KLabel '_`(\_`) $\vee_{Bool}$ `getKLabel` □ ==KLabel '_`[\_`]

RULE k: '_`(\_`)(`lookup` (L) ⇒ V, —) | store: $L \mapsto V$

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

RULE k: '_`(\_`)(`methodClosure` ($Class$, OL, XTs, S, (— `->` T)), Vs) ↷ K ⇒ `bindto` (XTs, Vs) ↷ S ↷`return;` | control: C fstack: ($\bullet_{List}$ ⇒ (Env, K, C crntObj: Obj' return: T')) | crntObj: Obj' ⇒ crntClass: $Class$ Obj | return: T' ⇒ T | env: Env ⇒ $\bullet_{Map}$ | store: OL …

RULE k: `return` V ;↷ — ⇒ `subtype` (`typeOf` (V), $\bullet_{Types}$, T, $\bullet_{Types}$) ↷`true?`↷`unsafeCast` (V, T) ↷ K | control: (fstack: (Env, K, C) ⇒ $\bullet_{List}$ | return: T) — ⇒ C | env: — ⇒ Env

RULE `objectClosure` (— envStack: $Class$ `::` —) `instanceOf` $Class$ ⇒ `true`

RULE `objectClosure` (— envStack: $Class1$ `::` — ⇒ $\bullet_{List}$) `instanceOf` $Class2$ when $Class1$ =/=K $Class2$

RULE `objectClosure` (— envStack: $\bullet_{List}$) `instanceOf` $Class$ ⇒ `false`

RULE k: `cast objectClosure` (crntClass: — Obj) `to` $Class$ ⇒ `objectClosure` (Obj) `instanceOf` $Class$ ↷`true?`↷`objectClosure` (crntClass: $Class$ Obj)

RULE k: `lvalue` (X ⇒ '_._(`this`, X)) | env: Env when $\neg_{Bool} X$ `in keys` Env

CONTEXT `lvalue` ('_._(□, —))

RULE k: `lvalue` ('_._(`objectClosure` (crntClass: $Class$ | envStack: $Class$ `::` $EnvC$ $EStack$), X)) ⇒ `lookupMember` (envStack: $Class$ `::` $EnvC$ $EStack$, X)

SYNTAX *K* ::= *Id* `::` *Bag*

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

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

RULE `lookupMember` (envStack: (— `::` env: Env ⇒ $\bullet_{List}$), X)   when $\neg_{Bool} X$ `in keys` Env

RULE `typeOf` (`objectClosure` (— envStack: $Class$ `::` —)) ⇒ `class` $Class$

RULE `typeOf` (`methodClosure` (—, —, —, —, T)) ⇒ T

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

RULE `subtype` (T , $\bullet_{Types}$, T , $\bullet_{Types}$) ⇒ `true`

RULE k: `subtype` (`class` $Class1$, $\bullet_{Types}$, `class` $Class$, $\bullet_{Types}$) ⇒ `subtype` (`class` $Class2$, $\bullet_{Types}$, `class` $Class$, $\bullet_{Types}$) | className: $Class1$ | extends: $Class2$ when $Class1$ =/=K $Class$

RULE `subtype` ($T1$, $T2$, Ts, $T1'$, $T2'$, Ts') ⇒ `subtype` ($T1$, $\bullet_{Types}$, $T1'$, $\bullet_{Types}$) ↷`true?`↷`subtype` ($T2$, Ts, $T2'$, Ts')

RULE `subtype` ($\bullet_{Types}$, $\bullet_{Types}$) ⇒ `true`

RULE `subtype` (`class object` , $\bullet_{Types}$, `class` $Class$, $\bullet_{Types}$) ⇒ `false`   when $Class$ =/=K `object`

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

RULE `unsafeCast` (`objectClosure` (crntClass: — Obj), `class` $Class$) ⇒ `objectClosure` (crntClass: $Class$ Obj)

RULE `unsafeCast` (V, T) ⇒ V when `typeOf` (V) ==K T

SYNTAX *K* ::= `true?`

RULE `true` ↷`true?` ⇒ $\bullet_K$

END MODULE