SIMPLE — Typed — Dynamic

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

Abstract

This is the $\mathbb{K}$ dynamic semantics of the typed SIMPLE language. It is very similar to the semantics of the untyped SIMPLE, the difference being that we now dynamically check the typing policy described in the static semantics of typed SIMPLE. Because of the dynamic nature of the semantics, we can also perform some additional checks which were not possible in the static semantics, such as memory leaks due to accessing an array out of its bounds. We will highlight the differences between the dynamically typed and the untyped SIMPLE as we proceed with the semantics. We recommend the reader to consult the typing policy and the syntax of types discussed in the static semantics of the typed SIMPLE language.

MODULE SIMPLE-TYPED-DYNAMIC-SYNTAX

Syntax

The syntax of typed SIMPLE extends that of untyped SIMPLE with support for declaring types to variables and functions.

The syntax below is identical to that of the static semantics of typed SIMPLE. However, the $\mathbb{K}$ strictness attributes are like those of the untyped SIMPLE, to capture the desired evaluation strategies of the various language constructs.

SYNTAX *Id* ::= `main`

Types

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

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

Declarations

SYNTAX *Param* ::= *Type Id*

SYNTAX *Params* ::= $List\{$*Param*, “,”$\}$

SYNTAX *Decl* ::= *Type Exps* `;`
| *Type Id*(*Params*)*Block*

Expressions

SYNTAX *Exp* ::= *Int*
| *Bool*
| *String*
| *Id*
| (*Exp*) [bracket]
| `++` *Exp*
| *Exp*[*Exps*] [strict]
| *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(1)]
| *Exp* `||` *Exp* [strict(1)]
| `spawn` *Block*
| *Exp* `=` *Exp* [strict(2)]

Like in the static semantics, there is no need for lists of identifiers (because we now have lists of parameters).

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*

The same desugaring macros like in the statically typed SIMPLE.

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

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

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

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

END MODULE

MODULE SIMPLE-TYPED-DYNAMIC

Semantics

Values and results

These are similar to those of untyped SIMPLE, except that the array references and the function abstrations now also hold their types. These types are needed in order to easily compute the type of any value in the language (see the auxiliary `typeOf` operation at the end of this module).

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

SYNTAX *Vals* ::= $List\{$*Val*, “,”$\}$

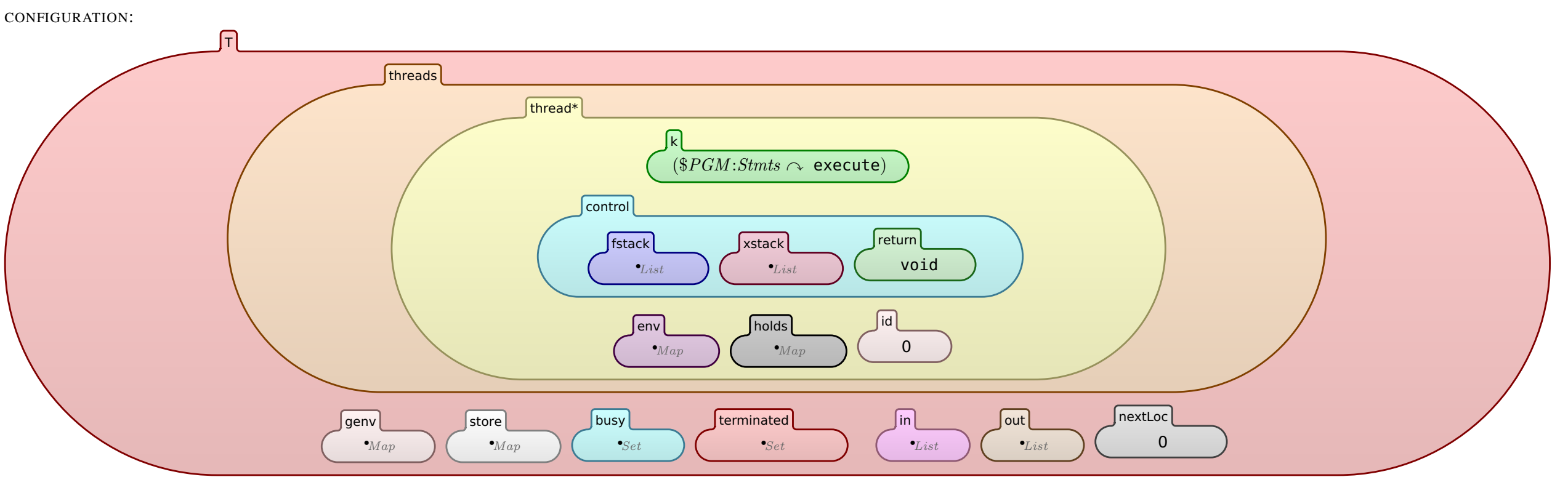
SYNTAX *Exp* ::= *Val*

SYNTAX *KResult* ::= *Val*

Configuration

The configuration is almost identical to that of untyped SIMPLE, except for a `return` cell inside the `control` cell. This `return` cell will hold, like in the static semantics of typed SIMPLE, the expected type of the value returned by the function being executed. The contents of this cell will be set whenever a function is invoked and will be checked whenever the evaluation of the function body encounters an explicit `return` statement.

CONFIGURATION:



Declarations and Initialization

Variable Declaration. The “undefined” construct is now parameterized by a type. A main difference between untyped SIMPLE and dynamically typed SIMPLE is that the latter assigns a type to each of its locations and that type cannot be changed during the execution of the program. We do not do any memory management in our semantic definitions here, so locations cannot be reclaimed, garbage collected and/or reused. Each location corresponds precisely to an allocated variable or array element, whose type was explicitly or implicitly declared in the program and does not change. It is therefore safe to type each location and then never allow that type to change. The typed undefined values effectively assign both a type and an undefined value to a location.

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

RULE k: $\dfrac{T{:}Type\ \ X{:}Id\ \texttt{;}}{\bullet_K}$ env: $\dfrac{Env}{Env[L\ /\ X]}$ store: $\dfrac{\bullet_{Map}}{L \mapsto \bot_T}$ nextLoc: $\dfrac{L{:}Int}{L +_{Int} 1}$

Array Declaration. The dynamic semantics of typed array declarations is similar to that in untyped SIMPLE, but we have to make sure that we associate the right type to the allocated locations.

RULE k: $\dfrac{T{:}Type\ \ X{:}Id[N{:}Int]\ \texttt{;}}{\bullet_K}$ env: $\dfrac{Env}{Env[L\ /\ X]}$ store: $\dfrac{\bullet_{Map}}{L \mapsto \texttt{array}\ (T, L +_{Int} 1, N)\ \ (L +_{Int} 1) \ldots (L +_{Int} N) \mapsto \bot_T}$ nextLoc: $\dfrac{L{:}Int}{L +_{Int} 1 +_{Int} N}$ requires $N \geq_{Int} 0$

CONTEXT —:Type —:Exp[☐] ;

The desugaring of multi-dimensional arrays into unidimensional ones is also similar to that in untyped SIMPLE, although we have to make sure that all the declared variables have the right types. The auxiliary operation `T<Vs>`, defined at the end of the file, adds the length of `Vs` dimensions to the type `T`.

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

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

Function declaration. Store all function parameters, as well as the return type, as part of the lambda abstraction. In the spirit of dynamic typing, we will make sure that parameters are well typed when the function is invoked.

RULE k: $\dfrac{T{:}Type\ \ F{:}Id(Ps{:}Params)S}{\bullet_K}$ env: $\dfrac{Env}{Env[L\ /\ F]}$ store: $\dfrac{\bullet_{Map}}{L \mapsto \texttt{lambda}\ (T, Ps, S)}$ nextLoc: $\dfrac{L}{L +_{Int} 1}$

Calling `main()`

When done with the first pass, call `main()`.

SYNTAX *K* ::= `execute`

RULE k: $\dfrac{\texttt{execute}}{\texttt{main}(\bullet_{Exps})\ \texttt{;}}$ env: Env genv: $\dfrac{\bullet_{Map}}{Env}$ [structural]

Expressions

Variable lookup.

RULE k: $\dfrac{X{:}Id}{V}$ env: $X \mapsto L$ store: $L \mapsto V{:}Val$ [lookup]

Variable/Array increment.

CONTEXT `++` $\dfrac{☐}{\texttt{lvalue}\ (☐)}$

RULE k: $\dfrac{\texttt{++ loc}\ (L)}{I +_{Int} 1}$ store: $L \mapsto \dfrac{I{:}Int}{I +_{Int} 1}$ [increment]

Arithmetic operators.

RULE $\dfrac{I1{:}Int\ \texttt{+}\ I2{:}Int}{I1 +_{Int} I2}$

RULE $\dfrac{Str1{:}String\ \texttt{+}\ Str2{:}String}{Str1 +_{String} Str2}$

RULE $\dfrac{I1{:}Int\ \texttt{-}\ I2{:}Int}{I1 -_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{*}\ I2{:}Int}{I1 *_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{/}\ I2{:}Int}{I1 \div_{Int} I2}$ requires $I2 \neq_K 0$

RULE $\dfrac{I1{:}Int\ \texttt{\%}\ I2{:}Int}{I1\ \%_{Int}\ I2}$ requires $I2 \neq_K 0$

RULE $\dfrac{\texttt{-}\ I{:}Int}{0 -_{Int} I}$

RULE $\dfrac{I1{:}Int\ \texttt{<}\ I2{:}Int}{I1 <_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{<=}\ I2{:}Int}{I1 \leq_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{>}\ I2{:}Int}{I1 >_{Int} I2}$

RULE $\dfrac{I1{:}Int\ \texttt{>=}\ I2{:}Int}{I1 \geq_{Int} I2}$

RULE $\dfrac{V1{:}Val\ \texttt{==}\ V2{:}Val}{V1 =_K V2}$

RULE $\dfrac{V1{:}Val\ \texttt{!=}\ V2{:}Val}{V1 \neq_K V2}$

RULE $\dfrac{\texttt{!}\ T{:}Bool}{\neg_{Bool}(T)}$

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

RULE $\dfrac{\texttt{false \&\& —}}{\texttt{false}}$

RULE $\dfrac{\texttt{true || —}}{\texttt{true}}$

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

Array lookup. Check array bounds, as part of the dynamic typing policy.

RULE $\dfrac{V{:}Val[N1{:}Int\ \texttt{,}\ N2{:}Int\ \texttt{,}\ Vs{:}Vals]}{V[N1][N2\ \texttt{,}\ Vs]}$ [structural, anywhere]

RULE $\dfrac{\texttt{array}\ (\text{—:Type}, L{:}Int, M{:}Int)[N{:}Int]}{\texttt{lookup}\ (L +_{Int} N)}$ requires $N \geq_{Int} 0 \wedge_{Bool} N <_{Int} M$ [structural, anywhere]

Size of an array.

RULE $\dfrac{\texttt{sizeOf}\ (\ \texttt{array}\ (—, —, N))}{N}$

Function call. Define function call and return together, to see their relationship. Note that the operation `mkDecls` now declares properly typed instantiated variables, and that the semantics of `return` also checks that that type of the returned value is expected one.

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

RULE k: $\dfrac{\texttt{lambda}\ (T, Ps, S)(Vs{:}Vals) \curvearrowright K}{\texttt{mkDecls}\ (Ps, Vs)\ \ S\ \ \texttt{return ;}}$ control: (fstack: $\dfrac{\bullet_{List}}{(T', Env, K, C)}$ return: $\dfrac{T'}{T}$ C) env: $\dfrac{Env}{GEnv}$ genv: $GEnv$

RULE k: $\dfrac{\texttt{return}\ V{:}Val\ \texttt{;} \curvearrowright —}{V \curvearrowright K}$ control: (fstack: $\dfrac{(T', Env, K, C)}{\bullet_{List}}$ return: $\dfrac{T}{T'}$ $\dfrac{—}{C}$) env: $\dfrac{—}{Env}$ requires $\texttt{typeOf}\ (V) =_K T$

Like the `undefined` above, `nothing` also gets tagged with a type now. The empty `return` statement is completed to return the `nothing` value tagged as expected.

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

RULE k: $\dfrac{\texttt{return ;}}{\texttt{return nothing}\ (T)\ \texttt{;}}$ return: T [structural]

Read.

RULE k: $\dfrac{\texttt{read}\ ()}{I}$ in: $\dfrac{I{:}Int}{\bullet_{List}}$ [read]

Assignment. The assignment now checks that the type of the assigned location is preserved:

CONTEXT $\dfrac{☐}{\texttt{lvalue}\ (☐)}$ `=` —

RULE k: $\dfrac{\texttt{loc}\ (L)\ \texttt{=}\ V{:}Val}{V}$ store: $L \mapsto \dfrac{V'}{V}$ requires $\texttt{typeOf}\ (V) =_K \texttt{typeOf}\ (V')$ [assignment]

Statements

Blocks.

RULE $\dfrac{\{\}}{\bullet_K}$ [structural]

RULE k: $\dfrac{\{S\}}{S \curvearrowright \texttt{env}\ (Env)}$ env: Env [structural]

Sequential composition.

RULE $\dfrac{S1{:}Stmts\ \ S2{:}Stmts}{S1 \curvearrowright S2}$ [structural]

Expression statements.

RULE $\dfrac{\text{—:Val}\ \texttt{;}}{\bullet_K}$

Conditional.

RULE $\dfrac{\texttt{if (true)}S\ \texttt{else}\ —}{S}$

RULE $\dfrac{\texttt{if (false)— else}\ S}{S}$

While loop.

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

Print. We only allow printing integers and strings:

RULE k: $\dfrac{\texttt{print}\ (V{:}Val,\ Es)\ \texttt{;}}{Es}$ out: $\dfrac{\bullet_{List}}{V}$ requires $\texttt{typeOf}\ (V) =_K \texttt{int} \vee_{Bool} \texttt{typeOf}\ (V) =_K \texttt{string}$ [print]

RULE $\dfrac{\texttt{print}\ (\bullet_{Vals})\ \texttt{;}}{\bullet_K}$ [structural]

Exceptions. Exception parameters are now typed, but note that the semantics below works correctly only when the thrown exception has the same type as the innermost try-catch paramete. To keep things simple, for the time being we can assume that SIMPLE only throws and catches integer values, in which case our semantics below works fine:

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

SYNTAX *K* ::= `popx`

RULE k: $\dfrac{\texttt{try}\ S1\ \texttt{catch}\ (P)S2 \curvearrowright K}{S1 \curvearrowright \texttt{popx}}$ control: (xstack: $\dfrac{\bullet_{List}}{(P, S2, K, Env, C)}$ C) env: Env

RULE k: $\dfrac{\texttt{popx}}{\bullet_K}$ xstack: $\dfrac{\text{—:ListItem}}{\bullet_{List}}$

RULE k: $\dfrac{\texttt{throw}\ V{:}Val\ \texttt{;} \curvearrowright —}{\{T\ \ X\ \texttt{=}\ V\ \texttt{;}\ \ S2\} \curvearrowright K}$ control: (xstack: $\dfrac{(T{:}Type\ \ X{:}Id, S2, K, Env, C)}{\bullet_{List}}$ $\dfrac{—}{C}$) env: $\dfrac{—}{Env}$

Threads.

Thread creation

RULE thread: (k: $\dfrac{\texttt{spawn}\ S}{T}$ env: Env) $\dfrac{\bullet_{Bag}}{\text{thread: (k: } S \text{ env: } Env \text{ id: } T\text{)}}$ requires $\texttt{fresh}\ (T{:}Int)$

Thread termination

RULE $\dfrac{\text{thread: (k: } \bullet_K \text{ holds: } H \text{ id: } T \text{)}}{\bullet_{Bag}}$ busy: $\dfrac{Busy}{Busy -_{Set} \texttt{keys}\ (H)}$ terminated: $\dfrac{\bullet_{Set}}{T}$

Thread joining

RULE k: $\dfrac{\texttt{join}\ T{:}Int\ \texttt{;}}{\bullet_K}$ terminated: T

Acquire lock

RULE k: $\dfrac{\texttt{acquire}\ V{:}Val\ \texttt{;}}{\bullet_K}$ holds: $\dfrac{\bullet_{Map}}{V \mapsto 0}$ busy: $Busy\ \ \dfrac{\bullet_{Set}}{V}$ requires $(\neg_{Bool}(V\ \texttt{in}\ Busy{:}Set))$ [acquire]

RULE k: $\dfrac{\texttt{acquire}\ V\ \texttt{;}}{\bullet_K}$ holds: $V{:}Val \mapsto \dfrac{N{:}Int}{N +_{Int} 1}$

Release lock

RULE k: $\dfrac{\texttt{release}\ V{:}Val\ \texttt{;}}{\bullet_K}$ holds: $V \mapsto \dfrac{N}{N{:}Int -_{Int} 1}$ requires $N >_{Int} 0$

RULE k: $\dfrac{\texttt{release}\ V\ \texttt{;}}{\bullet_K}$ holds: $\dfrac{V{:}Val \mapsto 0}{\bullet_{Map}}$ busy: $\dfrac{V}{\bullet_{Set}}$

Rendezvous synchronization

RULE k: $\dfrac{\texttt{rendezvous}\ V{:}Val\ \texttt{;}}{\bullet_K}$ k: $\dfrac{\texttt{rendezvous}\ V\ \texttt{;}}{\bullet_K}$ [rendezvous]

Auxiliary declarations and operations

Turns a list of parameters and a list of instance values for them into a list of variable declarations.

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

RULE $\dfrac{\texttt{mkDecls}\ ((T{:}Type\ \ X{:}Id\ \texttt{,}\ Ps{:}Params), (V{:}Val\ \texttt{,}\ Vs{:}Vals))}{T\ \ X\ \texttt{=}\ V\ \texttt{;}\ \ \texttt{mkDecls}\ (Ps, Vs)}$

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

Location lookup.

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

RULE k: $\dfrac{\texttt{lookup}\ (L)}{V}$ store: $L \mapsto V{:}Val$ [lookup]

Environment recovery.

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

RULE k: $\dfrac{\texttt{env}\ (Env)}{\bullet_K}$ env: $\dfrac{—}{Env}$ [structural]

RULE $\dfrac{\texttt{env}\ (—)}{\bullet_K} \curvearrowright \texttt{env}\ (—)$ [structural]

lvalue and loc

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

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

RULE k: $\texttt{lvalue}\ (\dfrac{X{:}Id}{\texttt{loc}\ (L)})$ env: $X \mapsto L{:}Int$ [structural]

CONTEXT `lvalue` (—:Exp[☐])

CONTEXT `lvalue` (☐[—:Exps])

RULE $\texttt{lvalue}\ (\dfrac{\texttt{lookup}\ (L{:}Int)}{\texttt{loc}\ (L)})$ [structural]

Adds the corresponding depth to an array type

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

RULE $\dfrac{T{:}Type\ \texttt{<}\ —\texttt{,}\ Vs{:}Vals\ \texttt{>}}{T[]\ \texttt{<}\ Vs\ \texttt{>}}$

RULE $\dfrac{T{:}Type\ \texttt{<}\ \bullet_{Vals}\ \texttt{>}}{T}$

Sequences of locations.

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

RULE $\dfrac{N \ldots M \mapsto —}{\bullet_{Map}}$ requires $N >_{Int} M$

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

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

RULE $\dfrac{\texttt{typeOf}\ (\text{—:Int})}{\texttt{int}}$

RULE $\dfrac{\texttt{typeOf}\ (\text{—:Bool})}{\texttt{bool}}$

RULE $\dfrac{\texttt{typeOf}\ (\text{—:String})}{\texttt{string}}$

RULE $\dfrac{\texttt{typeOf}\ (\ \texttt{array}\ (T, —, —))}{(T[])}$

RULE $\dfrac{\texttt{typeOf}\ (\ \texttt{lambda}\ (T, Ps, —))}{\texttt{getTypes}\ (Ps)\ \texttt{->}\ T}$

RULE $\dfrac{\texttt{typeOf}\ (\bot_T)}{T}$

RULE $\dfrac{\texttt{typeOf}\ (\ \texttt{nothing}\ (T))}{T}$

List of types of a parameter.

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

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

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

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

END MODULE