SIMPLE — Untyped

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

Abstract

This is the $\mathbb{K}$ semantic definition of the untyped SIMPLE language. SIMPLE is intended to be a pedagogical and research language that captures the essence of the imperative programming paradigm, extended with several features often encountered in imperative programming languages. A program consists of a set of global variable declarations and function definitions. Like in C, function definitions cannot be nested and each program must have one function called `main`, which is invoked when the program is executed. To make it more interesting and to highlight some of $\mathbb{K}$'s strengths, SIMPLE includes the following features in addition to the conventional imperative expression and statement constructs:

- Multidimensional arrays and array references. An array evaluates to an array reference, which is a special value holding a location (where the elements of the array start) together with the size of the array; the elements of the array can be array references themselves (particularly when the array is multi-dimensional). Array references are ordinary values, so they can be assigned to variables and passed/received by functions.
- Functions and function values. Functions can have zero or more parameters and can return abruptly using a `return` statement. SIMPLE follows a call-by-value parameter passing style, with static scoping. Function names evaluate to function abstractions, which hereby become ordinary values in the language, same like the array references.
- Blocks with locals. SIMPLE variables can be declared anywhere, their scope being from the place where they are declared until the end of the most nested enclosing block.
- Input/Output. The expression `read()` evaluates to the next value in the input buffer, and the statement `write(e)` evaluates `e` and outputs its value to the output buffer. The input and output buffers are lists of values.
- Exceptions. SIMPLE has parametric exceptions (the value thrown as an exception can be caught and bound).
- Concurrency via dynamic thread creation/termination and synchronization. One can spawn a thread to execute any statement. The spawned thread shares with its parent its environment at creation time. Threads can be synchronized via a join command which blocks the current thread until the joined thread completes, via re-entrant locks which can be acquired and released, as well as through rendezvous commands.

Like in many other languages, some of SIMPLE's constructs can be desugared into a smaller set of basic constructs. We do that at the end of the syntax module, and then we only give semantics to the core constructs.

Note: This definition is commented slightly more than others, because it is intended to be one of the first non-trivial definitions that the new user of $\mathbb{K}$ sees. We recommend the beginner user to first check the language definitions discussed in the $\mathbb{K}$ tutorial.

MODULE SIMPLE-UNTYPED-SYNTAX

Syntax

We start by defining the SIMPLE syntax. The language constructs discussed above have the expected syntax and evaluation strategies. Recall that in $\mathbb{K}$ we annotate the syntax with appropriate strictness attributes, thus giving each language construct the desired evaluation strategy.

Identifiers

Recall from the $\mathbb{K}$ tutorial that identifiers are builtin and come under the syntactic category *Id*. The special identifier for the function "main" belongs to all programs, and plays a special role in the semantics, so we declare it explicitly. This would not be necessary if the identifiers were all included automatically in semantic definitions, but that is not possible because of parsing reasons (e.g., $\mathbb{K}$ variables used to match concrete identifiers would then be ambiguously parsed as identifiers). They are only included in the parser generated to parse programs (and used by the `kast` tool). Consequently, we have to explicitly declare all the concrete identifiers that play a special role in the semantics, like `main` below.

SYNTAX $Id ::= \texttt{main}$

Declarations

There are two types of declarations: for variables (including arrays) and for functions. We are going to allow declarations of the form "`var x=10, a[10,10], y=23;`", which is why we allow the `var` keyword to take a list of expressions. The non-terminals used in the two productions below are defined shortly.

SYNTAX $Decl ::= \texttt{var } Exps\ \texttt{;}$
$\quad\mid\ \texttt{function } Id(Ids)Block$

Expressions

The expression constructs below are standard. Increment (`++`) takes an expression rather than a variable because it can also increment an array element. Recall that the syntax we define in $\mathbb{K}$ is what we call "the syntax of the semantics": while powerful enough to define non-trivial syntaxes (thanks to the underlying SDF technology that we use), we typically refrain from defining precise syntaxes, that is, ones which accept precisely the well-formed programs (that would not be possible anyway in general). That job is deferred to type systems, which can also be defined in $\mathbb{K}$. In other words, we are not making any effort to guarantee syntactically that only variables or array elements are passed to the increment construct, we allow any expression. Nevertheless, we will only give semantics to those, so expressions of the form `++5`, which parse (but which will be rejected by our type system in the typed version of SIMPLE later), will get stuck when executed. Arrays can be multidimensional and can hold other arrays, so their lookup operation takes a list of expressions as argument and applies to an expression (which can in particular be another array lookup), respectively. The construct `sizeOf` gives the size of an array in number of elements of its first dimension. Note that almost all constructs are strict. The only constructs which are not strict are the increment (since its first argument gets updated, so it cannot be evaluated), the input read which takes no arguments so strictness is irrelevant for it, the logical and and or constructs which are short-circuited, the thread spawning construct which creates a new thread executing the argument expression and return its unique identifier to the creating thread (so it cannot just evaluate its argument in place), and the assignment which is only strict in its second argument (for the same reason as the increment).

SYNTAX $Exp ::= Int$
$\quad\mid\ Bool$
$\quad\mid\ String$
$\quad\mid\ Id$
$\quad\mid\ (Exp)$ [bracket]
$\quad\mid\ \texttt{++}\ Exp$
$\quad\mid\ Exp[Exps]$ [strict]
$\quad\mid\ Exp(Exps)$ [strict]
$\quad\mid\ \texttt{-}\ Exp$ [strict]
$\quad\mid\ \texttt{sizeOf}\ (Exp)$ [strict]
$\quad\mid\ \texttt{read}\ ()$
$\quad\mid\ Exp\ \texttt{*}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{/}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{\%}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{+}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{-}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{<}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{<=}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{>}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{>=}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{==}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{!=}\ Exp$ [strict]
$\quad\mid\ \texttt{!}\ Exp$ [strict]
$\quad\mid\ Exp\ \texttt{\&\&}\ Exp$ [strict(1)]
$\quad\mid\ Exp\ \texttt{||}\ Exp$ [strict(1)]
$\quad\mid\ \texttt{spawn}\ Block$
$\quad\mid\ Exp\ \texttt{=}\ Exp$ [strict(2)]

We also need comma-separated lists of identifiers and of expressions. Moreover, we want them to be strict, that is, to evaluate to lists of results whenever requested (e.g., when they appear as strict arguments of the constructs above).

SYNTAX $Ids ::= List\{Id, \text{“,”}\}$

SYNTAX $Exps ::= List\{Exp, \text{“,”}\}$ [strict]

Statements

Most of the statement constructs are standard for imperative languages. We syntactically distinguish between empty and non-empty blocks, because we chose *Stmts* not to be a ("`;`"-separated) list of *Stmt*. Variables can be declared anywhere inside a block, their scope ending with the block. Expressions are allowed to be used for their side effects only (followed by a semicolon "`;`"). Functions are allowed to abruptly return. The exceptions are parametric, i.e., one can throw a value which is bound to the variable declared by `catch`. Threads can be dynamically created and terminated, and can synchronize with `join`, `acquire`, `release` and `rendezvous`. Note that the strictness attributes obey the intended evaluation strategy of the various constructs. In particular, the if-then-else construct is strict only in its first argument (the if-then construct will be desugared into if-then-else), while the loop constructs are not strict in any arguments. The `print` statement construct is variadic, that is, it takes an arbitrary number of arguments.

SYNTAX $Block ::= \{\}$
$\quad\mid\ \{Stmts\}$

SYNTAX $Stmt ::= Decl$
$\quad\mid\ Block$
$\quad\mid\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{if}\ (Exp)Block\ \texttt{else}\ Block$ [avoid, strict(1)]
$\quad\mid\ \texttt{if}\ (Exp)Block$
$\quad\mid\ \texttt{while}\ (Exp)Block$
$\quad\mid\ \texttt{for}\ (Stmt\ \ Exp\ \texttt{;}\ Exp)Block$
$\quad\mid\ \texttt{return}\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{return}\ \texttt{;}$
$\quad\mid\ \texttt{print}\ (Exps)\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{try}\ Block\ \texttt{catch}\ (Id)Block$
$\quad\mid\ \texttt{throw}\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{join}\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{acquire}\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{release}\ Exp\ \texttt{;}$ [strict]
$\quad\mid\ \texttt{rendezvous}\ Exp\ \texttt{;}$ [strict]

SYNTAX $Stmts ::= Stmt$
$\quad\mid\ Stmts\ \ Stmts$

Desugared Syntax

This part desugars some of SIMPLE's language constructs into core ones. We only want to give semantics to core constructs, so we get rid of the derived ones before we start the semantics. All desugaring macros below are straightforward.

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

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

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

For the semantics, we can therefore assume from now on that each conditional has both branches, that there are only `while` loops, and that each variable is declared alone and without any initialization as part of the declaration.

END MODULE

MODULE SIMPLE-UNTYPED

Basic Semantic Infrastructure

Before one starts adding semantic rules to a $\mathbb{K}$ definition, one needs to define the basic semantic infrastructure consisting of definitions for *values* and *configuration*. As discussed in the definitions in the $\mathbb{K}$ tutorial, the values are needed to know when to stop applying the heating rules and when to start applying the cooling rules corresponding to strictness or context declarations. The configuration serves as a backbone for the process of configuration abstraction which allows users to only mention the relevant cells in each semantic rule, the rest of the configuration context being inferred automatically. Although in some cases the configuration could be automatically inferred from the rules, we believe that it is very useful for language designers/semanticists to actually think of and design their configuration explicitly, so the current implementation of $\mathbb{K}$ requires one to define it.

Values

We here define the values of the language that the various fragments of programs evaluate to. First, integers and Booleans are values. As discussed, arrays evaluate to special array reference values holding (1) a location from where the array's elements are contiguously allocated in the store, and (2) the size of the array. Functions evaluate to function values as λ-abstractions (we do not need to evaluate functions to closures because each function is executed in the fixed global environment and function definitions cannot be nested). Like in IMP and other languages, we finally tell the tool that values are $\mathbb{K}$ results.

SYNTAX $Val ::= Int$
$\quad\mid\ Bool$
$\quad\mid\ String$
$\quad\mid\ \texttt{array}\ (Int, Int)$
$\quad\mid\ \texttt{lambda}\ (Ids, Stmt)$

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

SYNTAX $Exp ::= Val$

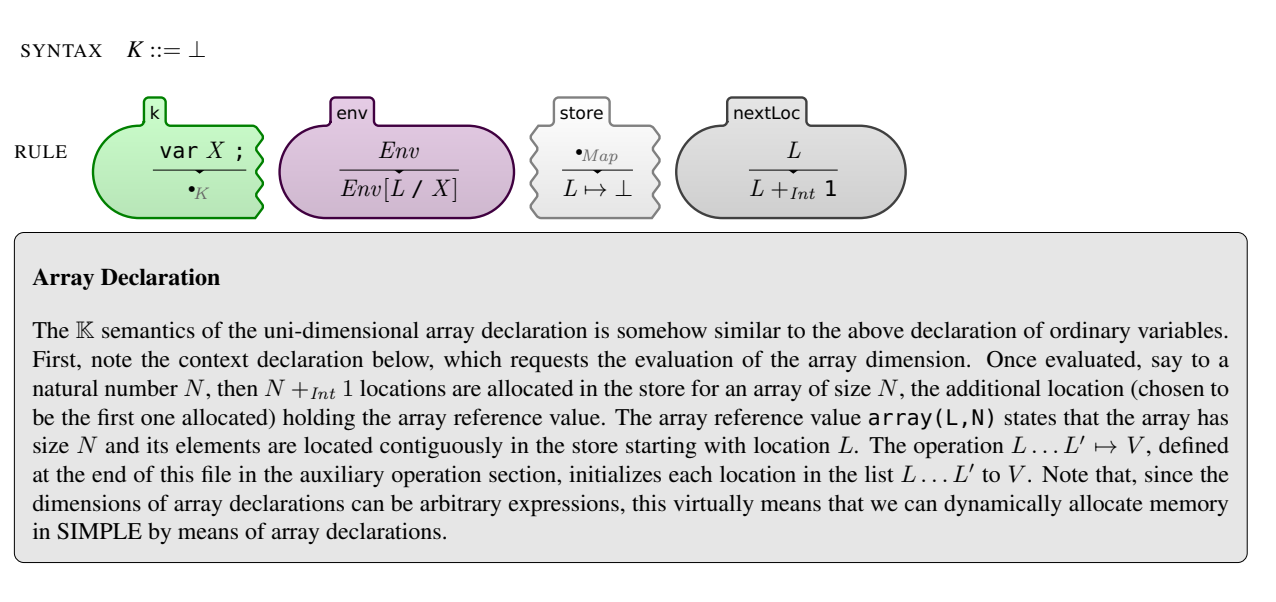
SYNTAX $KResult ::= Val$

The inclusion of values in expressions follows the methodology of syntactic definitions (like, e.g., in SOS): extend the syntax of the language to encompass all values and additional constructs needed to give semantics. In addition to that, it allows us to write the semantic rules using the original syntax of the language, and to parse them with the same (now extended with additional values) parser. If writing the semantics directly on the $\mathbb{K}$ AST, using the associated labels instead of the syntactic constructs, then one would not need to include values in expressions.

Configuration

The $\mathbb{K}$ configuration of SIMPLE consists of a top level cell, `T`, holding a `threads` cell, a global environment map cell `genv` mapping the global variables and function names to their locations, a shared store map cell `store` mapping each location to some value, a set cell `busy` holding the locks which have been acquired but not yet released by threads, a set cell `terminated` holding the unique identifiers of the threads which already terminated (needed for `join`), `input` and `output` list cells, and a `nextLoc` cell holding a natural number indicating the next available location. Unlike in the small languages in the $\mathbb{K}$ tutorial, where we used the fresh predicate to generate fresh locations, in larger languages, like SIMPLE, we prefer to explicitly manage memory. The location counter in `nextLoc` models an actual physical location in the store; for simplicity, we assume arbitrarily large memory and no garbage collection. The `threads` cell contains one `thread` cell for each existing thread in the program. Note that the thread cell has multiplicity "*", which means that at any given moment there could be zero, one or more `thread` cells. Each `thread` cell contains a computation cell `k`, a `control` cell holding the various control structures needed to jump to certain points of interest in the program execution, a local environment map cell `env` mapping the thread local variables to locations in the store, and finally a `holds` map cell indicating what locks have been acquired by the thread and not released so far and how many times (SIMPLE's locks are re-entrant). The `control` cell currently contains only two subcells, a function stack `fstack` which is a list and an exception stack `xstack` which is also a list. One can add more control structures in the `control` cell, such as a stack for break/continue of loops, etc., if the language is extended with more control-changing constructs. Note that all cells except for `k` are also initialized, in that they contain a ground term of their corresponding sort. The `k` cell is initialized with the program that will be passed to the $\mathbb{K}$ tool, as indicated by the *$PGM* variable, followed by the `execute` task (defined shortly).

CONFIGURATION:



Declarations and Initialization

We start by defining the semantics of declarations (for variables, arrays and functions).

Variable Declaration

The SIMPLE syntax was desugared above so that each variable is declared alone and its initialization is done as a separate statement. The semantic rule below matches resulting variable declarations of the form "`var` X`;`" on top of the `k` cell (indeed, note that the `k` cell is complete, or round, to the left, and is torn, or ruptured, to the right), allocates a fresh location L in the store which is initialized with a special value $\bot$ (indeed, the unit "•", or nothing, is matched anywhere in the map—note the tears at both sides—and replaced with the mapping $L \mapsto \bot$), and binds X to L in the local environment shadowing previous declarations of X, if any. This possible shadowing of X requires us to therefore update the entire environment map, which is expensive and can significantly slow down the execution of larger programs. On the other hand, since we know that L is not already bound in the store, we simply add the binding $L \mapsto \bot$ to the store, thus avoiding a potentially complete traversal of the the store map in order to update it. We prefer the approach used for updating the store whenever possible, because, in addition to being faster, it offers more true concurrency than the latter; indeed, according to the concurrent semantics of $\mathbb{K}$, the store is not frozen while $L \mapsto \bot$ is added to it, while the environment is frozen during the update operation $Env[L/X]$. The variable declaration command is also removed from the top of the computation cell and the fresh location counter is incremented. The undefined symbol $\bot$ added in the store is of sort K, instead of *Val*, on purpose; this way, the store lookup rules will get stuck when one attempts to lookup an initialized location. All the above happen in one transactional step, with the rule below. Note also how configuration abstraction allows us to only mention the needed cells; indeed, as the configuration above states, the `k` and `env` cells are actually located within a `thread` cell within the `threads` cell, but one needs not mention these: the configuration context of the rule is automatically transformed to match the declared configuration structure.

SYNTAX $K ::= \bot$

RULE $\left\langle \dfrac{\texttt{var}\ X\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{Env}{Env[L\,/\,X]} \right\rangle_{\mathsf{env}} \left\langle \dfrac{\cdot_{Map}}{L \mapsto \bot} \right\rangle_{\mathsf{store}} \left\langle \dfrac{L}{L +_{Int} 1} \right\rangle_{\mathsf{nextLoc}}$

Array Declaration

The $\mathbb{K}$ semantics of the uni-dimensional array declaration is somehow similar to the above declaration of ordinary variables. First, note the context declaration below, which requests the evaluation of the array dimension. Once evaluated, say to a natural number N, then $N +_{Int} 1$ locations are allocated in the store for an array of size N, the additional location (chosen to be the first one allocated) holding the array reference value. The array reference value `array(L,N)` states that the array has size N and its elements are located contiguously in the store starting with location L. The operation $L \ldots L' \mapsto V$, defined at the end of this file in the auxiliary operation section, initializes each location in the list $L \ldots L'$ to V. Note that, since the dimensions of array declarations can be arbitrary expressions, this virtually means that we can dynamically allocate memory in SIMPLE by means of array declarations.

CONTEXT $\texttt{var}\ \text{—}[\square]\ \texttt{;}$

RULE $\left\langle \dfrac{\texttt{var}\ X[N]\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{Env}{Env[L\,/\,X]} \right\rangle_{\mathsf{env}} \left\langle \dfrac{\cdot_{Map}}{L \mapsto \texttt{array}\ (L +_{Int} 1, N)\ \ (L +_{Int} 1) \ldots (L +_{Int} N) \mapsto \bot} \right\rangle_{\mathsf{store}} \left\langle \dfrac{L}{L +_{Int} 1 +_{Int} N} \right\rangle_{\mathsf{nextLoc}}$ requires $N \geq_{Int} 0$

SIMPLE allows multi-dimensional arrays. For semantic simplicity, we desugar them all into uni-dimensional arrays by code transformation. This way, we only need to give semantics to uni-dimensional arrays. First, note that the context rule above actually evaluates all the array dimensions (that's why we defined the expression lists strict!): Upon evaluating the array dimensions, the code generation rule below desugars multi-dimensional array declaration to uni-dimensional declarations. To this aim, we introduce two special unique variable identifiers, `$1` and `$2`. The first, `$1`, is assigned the array reference value of the current array, so that we can redeclare the array inside the loop body with fewer dimensions. The second variable, `$2`, iterates through and initializes each element of the current dimension:

SYNTAX $Id ::= \texttt{\$1}$
$\quad\mid\ \texttt{\$2}$

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

Ideally, one would like to perform syntactic desugarings like the one above before the actual semantics. Unfortunately, that was not possible in this case because the dimension expressions of the multi-dimensional array need to be evaluated first. Indeed, the desugaring rule above does not work if the dimensions of the declared array are arbitrary expressions, because they can have side effects (e.g., `a[++x,++x]`) and those side effects would be propagated each time the expression is evaluated in the desugaring code (note that both the loop condition and the nested multi-dimensional declaration would need to evaluate the expressions given as array dimensions).

Function declaration

Functions are evaluated to λ-abstractions and stored like any other values in the store. A binding is added into the environment for the function name to the location holding its body. Similarly to the C language, SIMPLE only allows function declarations at the top level of the program. More precisely, the subsequent semantics of SIMPLE only works well when one respects this requirement. Indeed, the simplistic context-free parser generated by the grammar above is more generous than we may want, in that it allows function declarations anywhere any declaration is allowed, including inside arbitrary blocks. However, as the rule below shows, we are *not* storing the declaration environment with the λ-abstraction value as closures do. Instead, as seen shortly, we switch to the global environment whenever functions are invoked, which is consistent with our requirement that functions should only be declared at the top. Thus, if one declares local functions, then one may see unexpected behaviors (e.g., when one shadows a global variable before declaring a local function). The type checker of SIMPLE, also defined in $\mathbb{K}$ (see `examples/simple/typed/static`), discards programs which do not respect this requirement.

RULE $\left\langle \dfrac{\texttt{function}\ F(Xs)S}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{Env}{Env[L\,/\,F]} \right\rangle_{\mathsf{env}} \left\langle \dfrac{\cdot_{Map}}{L \mapsto \texttt{lambda}\ (Xs, S)} \right\rangle_{\mathsf{store}} \left\langle \dfrac{L}{L +_{Int} 1} \right\rangle_{\mathsf{nextLoc}}$

When we are done with the first pass (pre-processing), the computation cell `k` contains only the token `execute` (see the configuration declaration above, where the computation item `execute` was placed right after the program in the `k` cell of the initial configuration) and the cell `genv` is empty. In this case, we have to call `main()` and to initialize the global environment by transferring the contents of the local environment into it. We prefer to do it this way, as opposed to processing all the top level declarations directly within the global environment, because we want to avoid duplication of semantics: the syntax of the global declarations is identical to that of their corresponding local declarations, so the semantics of the latter suffices provided that we copy the local environment into the global one once we are done with the pre-processing. We want this separate pre-processing step precisely because we want to create the global environment. All (top-level) functions end up having their names bound in the global environment and, as seen below, they are executed in that same global environment; all these mean, in particular, that the functions "see" each other, allowing for mutual recursion, etc.

SYNTAX $K ::= \texttt{execute}$

RULE $\left\langle \dfrac{\texttt{execute}}{\texttt{main}(\cdot_{Exps})\ \texttt{;}} \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}} \left\langle \dfrac{\cdot_{Map}}{Env} \right\rangle_{\mathsf{genv}}$ [structural]

Expressions

We next define the $\mathbb{K}$ semantics of all the expression constructs.

Variable lookup

When a variable X is the first computational task, and X is bound to some location L in the environment, and L is mapped to some value V in the store, then we rewrite X into V:

RULE $\left\langle \dfrac{X}{V} \right\rangle_{\mathsf{k}} \left\langle X \mapsto L \right\rangle_{\mathsf{env}} \left\langle L \mapsto V \right\rangle_{\mathsf{store}}$ [lookup]

Note that the rule above excludes reading $\bot$, because $\bot$ is not a value and V is checked at runtime to be a value.

Variable/Array increment

This is tricky, because we want to allow both `++x` and `++a[5]`. Therefore, we need to extract the lvalue of the expression to increment. To do that, we state that the expression to increment should be wrapped by the auxiliary `lvalue` operation and then evaluated. The semantics of this auxiliary operation is defined at the end of this file. For now, all we need to know is that it takes an expression and evaluates to a location value. Location values, also defined at the end of the file, are integers wrapped with the operation `loc`, to distinguish them from ordinary integers.

CONTEXT $\texttt{++}\ \dfrac{\square}{\texttt{lvalue}\ (\square)}$

RULE $\left\langle \dfrac{\texttt{++ loc}\ (L)}{I +_{Int} 1} \right\rangle_{\mathsf{k}} \left\langle L \mapsto \dfrac{I}{I +_{Int} 1} \right\rangle_{\mathsf{store}}$ [increment]

Arithmetic operators

There is nothing special about the following rules. They rewrite the language constructs to their library counterparts when their arguments become values of expected sorts:

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

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

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

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

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

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

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

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

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

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

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

The equality and inequality constructs reduce to syntactic comparison of the two argument values (which is what the equality on K terms does).

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

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

The logical negation is clear, but the logical conjunction and disjunction are short-circuited:

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

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

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

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

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

Array lookup

Untyped SIMPLE does not check array bounds (the dynamically typed version of it, in `examples/simple/typed/dynamic`, does check for array out of bounds). The first rule below desugars the multi-dimensional array access to uni-dimensional array access; recall that the array access operation was declared strict, so all sub-expressions involved are already values at this stage. The second rule rewrites the array access to a lookup operation at a precise location; we prefer to do it this way to avoid locking the store. The semantics of the auxiliary `lookup` operation is straightforward, and is defined at the end of the file.

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

RULE $\dfrac{\texttt{array}\ (L, \text{—})[N]}{\texttt{lookup}\ (L +_{Int} N)}$ [structural, anywhere]

Size of an array

The size of the array is stored in the array reference value, and the `sizeOf` construct was declared strict, so:

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

Function call

Function application was strict in both its arguments, so we can assume that both the function and its arguments are evaluated to values (the former expected to be a λ-abstraction). The first rule below matches a well-formed function application on top of the computation and performs the following steps atomically: it switches to the function body followed by "`return;`" (for the case in which the function does not use an explicit return statement); it pushes the remaining computation, the current environment, and the current control data onto the function stack (the remaining computation can thus also be discarded from the computation cell, because an unavoidable subsequent `return` statement—see above—will always recover it from the stack); it switches the current environment (which is being pushed on the function stack) to the global environment, which is where the free variables in the function body should be looked up; it binds the formal parameters to fresh locations in the new environment, and stores the actual arguments to those locations in the store (this latter step is easily done by reducing the problem to variable declarations, whose semantics we have already defined; the auxiliary operation `mkDecls` is defined at the end of the file). The second rule pops the computation, the environment and the control data from the function stack when a `return` statement is encountered as the next computational task, passing the returned value to the popped computation (the popped computation was the context in which the returning function was called). Note that the pushing/popping of the control data is crucial. Without it, one may have a function that contains an exception block with a return statement inside, which would put the `xstack` cell in an inconsistent state (since the exception block modifies it, but that modification should be irrelevant once the function returns). We add an artificial `nothing` value to the language, which is returned by the nulary `return;` statements.

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

RULE $\left\langle \dfrac{\texttt{lambda}\ (Xs, S)(Vs) \curvearrowright K}{\texttt{mkDecls}\ (Xs, Vs)\ \ S\ \ \texttt{return ;}} \right\rangle_{\mathsf{k}} \left\langle \left\langle \dfrac{\cdot_{List}}{(Env, K, C)} \right\rangle_{\mathsf{fstack}} C \right\rangle_{\mathsf{control}} \left\langle \dfrac{Env}{GEnv} \right\rangle_{\mathsf{env}} \left\langle GEnv \right\rangle_{\mathsf{genv}}$

RULE $\left\langle \dfrac{\texttt{return}\ (V)\ \texttt{;} \curvearrowright \text{—}}{V \curvearrowright K} \right\rangle_{\mathsf{k}} \left\langle \left\langle \dfrac{(Env, K, C)}{\cdot_{List}} \right\rangle_{\mathsf{fstack}} \dfrac{\text{—}}{C} \right\rangle_{\mathsf{control}} \left\langle \dfrac{\text{—}}{Env} \right\rangle_{\mathsf{env}}$

SYNTAX $Val ::= \texttt{nothing}$

RULE $\dfrac{\texttt{return ;}}{\texttt{return nothing ;}}$ [macro]

Like for division-by-zero, it is left unspecified what happens when the `nothing` value is used in domain calculations. For example, from the the perspective of the language semantics, $7 +_{Int}$ `nothing` can evaluate to anything, or may not evaluate at all (be undefined). If one wants to make sure that such artificial values are never misused, then one needs to define a static checker (also using $\mathbb{K}$, like our the type checker in `examples/simple/typed/static`) and reject programs that do. Note that, unlike the undefined symbol $\bot$ which had the sort K instead of *Val*, we defined `nothing` to be a value. That is because, as explained above, we do not want the program to get stuck when nothing is returned by a function. Instead, we want the behavior to be unspecified; in particular, if one is careful to never use the returned value in domain computation, like it happens when we call a function for its side effects (e.g., with a statement of the form "`f(x);`"), then the program does not get stuck.

Read

The `read()` expression construct simply evaluates to the next input value, at the same time discarding the input value from the `in` cell.

RULE $\left\langle \dfrac{\texttt{read}\ ()}{I} \right\rangle_{\mathsf{k}} \left\langle \dfrac{I}{\cdot_{List}} \right\rangle_{\mathsf{in}}$ [read]

Assignment

In SIMPLE, like in C, assignments are expression constructs and not statement constructs. To make it a statement all one needs to do is to follow it by a semi-colon "`;`" (see the semantics for expression statements below). Like for the increment, we want to allow assignments not only to variables but also to array elements, e.g., `e1[e2] = e3` where `e1` evaluates to an array reference, `e2` to a natural number, and `e3` to any value. Thus, we first compute the lvalue of the left-hand-side expression that appears in an assignment, and then we do the actual assignment to the resulting location:

CONTEXT $\dfrac{\square}{\texttt{lvalue}\ (\square)}\ \texttt{=}\ \text{—}$

RULE $\left\langle \dfrac{\texttt{loc}\ (L)\ \texttt{=}\ V}{V} \right\rangle_{\mathsf{k}} \left\langle L \mapsto \dfrac{\text{—}}{V} \right\rangle_{\mathsf{store}}$ [assignment]

Statements

We next define the $\mathbb{K}$ semantics of statements.

Blocks

Empty blocks are simply discarded, as shown in the first rule below. For non-empty blocks, we schedule the enclosed statement but we have to make sure the environment is recovered after the enclosed statement executes. Recall that we allow local variable declarations, whose scope is the block enclosing them. That is the reason for which we have to recover the environment after the block. This allows us to have a very simple semantics for variable declarations, as we did above. One can make the two rules below computational if one wants them to count as computational steps.

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

RULE $\left\langle \dfrac{\{S\}}{S \curvearrowright \texttt{env}\ (Env)} \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}}$ [structural]

The basic definition of environment recovery is straightforward and given in the section on auxiliary constructs at the end of the file.

There are two common alternatives to the above semantics of blocks. One is to keep track of the variables which are declared in the block and only recover those at the end of the block. This way one does more work for variable declarations but conceptually less work for environment recovery; we say "conceptually" because it is not clear that it is indeed the case that one does less work when AC matching is involved. The other alternative is to work with a stack of environments instead of a flat environment, and push the current environment when entering a block and pop it when exiting it. This way, one does more work when accessing variables (since one has to search the variable in the environment stack in a top-down manner), but on the other hand uses smaller environments and the definition gets closer to an implementation. Based on experience with dozens of language semantics and other $\mathbb{K}$ definitions, we have found that our approach above is the best trade-off between elegance and efficiency (especially since rewrite engines have built-in techniques to lazily copy terms, by need, thus not creating unnecessary copies), so it is the one that we follow in general.

Sequential composition

Sequential composition is desugared into $\mathbb{K}$'s builtin sequentialization operation (recall that, like in C, the semi-colon "`;`" is not a statement separator in SIMPLE—it is either a statement terminator or a construct for a statement from an expression). The rule below is structural, so it does not count as a computational step. One can make it computational if one wants it to count as a step. Note that $\mathbb{K}$ allows to define the semantics of SIMPLE in such a way that statements eventually dissolve from the top of the computation when they are completed; this is in sharp contrast to (artificially) "evaluating" them to a special `skip` statement value and then getting rid of that special value, as it is the case in other semantic approaches (where everything must evaluate to something). This means that once S_1 completes in the rule below, S_2 becomes automatically the next computation item without any additional (explicit or implicit) rules.

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

Expression statements

Expression statements are only used for their side effects, so their result value is simply discarded. Common examples of expression statements are ones of the form `++x;`, `x=e;`, `e1[e2]=e3;`, etc.

RULE $\dfrac{\text{—}\ \texttt{;}}{\cdot_K}$

Conditional

Since the conditional was declared with the `strict(1)` attribute, we can assume that its first argument will eventually be evaluated. The rules below cover the only two possibilities in which the conditional is allowed to proceed (otherwise the rewriting process gets stuck).

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

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

While loop

The simplest way to give the semantics of the while loop is by unrolling. Note, however, that its unrolling is only allowed when the while loop reaches the top of the computation (to avoid non-termination of unrolling). We prefer the rule below to be structural, because we don't want the unrolling of the while loop to count as a computational step; this is unavoidable in conventional semantics, but it is possible in $\mathbb{K}$ thanks to its distinction between structural and computational rules. The simple while loop semantics below works because our while loops in SIMPLE are indeed very basic. If we allowed break/continue of loops then we would need a completely different semantics, which would also involve the `control` cell.

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

Print

The `print` statement was strict, so all its arguments are now evaluated (recall that `print` is variadic). We append each of its evaluated arguments to the output buffer, and discard the residual `print` statement with an empty list of arguments.

RULE $\left\langle \texttt{print}\ (\dfrac{V\texttt{,}\ Es}{Es})\ \texttt{;} \right\rangle_{\mathsf{k}} \left\langle \dfrac{\cdot_{List}}{V} \right\rangle_{\mathsf{out}}$ [print]

RULE $\dfrac{\texttt{print}\ (\cdot_{Vals})\ \texttt{;}}{\cdot_K}$ [structural]

Exceptions

SIMPLE allows parametric exceptions, in that one can throw and catch a particular value. The statement "`try` S_1 `catch(`X`)` S_2" proceeds with the evaluation of S_1. If S_1 evaluates normally, i.e., without any exception thrown, then S_2 is discarded and the execution continues normally. If S_1 throws an exception with a statement of the form "`throw` E", then E is first evaluated to some value V (`throw` was declared to be strict), then V is bound to X, then S_2 is evaluated in the new environment while the reminder of S_1 is discarded, then the environment is recovered and the execution continues normally with the statement following the "`try` S_1 `catch(`X`)` S_2" statement. Exceptions can be nested and the statements in the "catch" part (S_2 in our case) can throw exceptions to the upper level. One should be careful with how one handles the control data structures here, so that the abrupt changes of control due to exception throwing and to function returns interact correctly with each other. For example, we want to allow function calls inside the statement S_1 in a "`try` S_1 `catch(`X`)` S_2" block which can throw an exception that is not caught by the function but instead is propagated to the "`try` S_1 `catch(`X`)` S_2" block that called the function. Therefore, we have to make sure that the function stack as well as other potential control structures are also properly modified when the exception is thrown to correctly recover the execution context. This can be easily achieved by pushing/popping the entire current control context onto the exception stack. The three rules below modularly do precisely the above.

SYNTAX $K ::= (Id, Stmts, K, Map, Bag)$

SYNTAX $K ::= \texttt{popx}$

RULE $\left\langle \dfrac{\texttt{try}\ S1\ \texttt{catch}\ (X)\{S2\} \curvearrowright K}{S1 \curvearrowright \texttt{popx}} \right\rangle_{\mathsf{k}} \left\langle \left\langle \dfrac{\cdot_{List}}{(X, S2, K, Env, C)} \right\rangle_{\mathsf{xstack}} C \right\rangle_{\mathsf{control}} \left\langle Env \right\rangle_{\mathsf{env}}$

RULE $\left\langle \dfrac{\texttt{popx}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{\text{—}}{\cdot_{List}} \right\rangle_{\mathsf{xstack}}$

RULE $\left\langle \dfrac{\texttt{throw}\ V\ \texttt{;} \curvearrowright \text{—}}{\{\ \texttt{var}\ X\ \texttt{=}\ V\ \texttt{;}\ \ S2\} \curvearrowright K} \right\rangle_{\mathsf{k}} \left\langle \left\langle \dfrac{(X, S2, K, Env, C)}{\cdot_{List}} \right\rangle_{\mathsf{xstack}} \dfrac{\text{—}}{C} \right\rangle_{\mathsf{control}} \left\langle \dfrac{\text{—}}{Env} \right\rangle_{\mathsf{env}}$

The catch statement S_2 needs to be executed in the original environment, but where the thrown value V is bound to the catch variable X. We here chose to rely on two previously defined constructs when giving semantics to the catch part of the statement: (1) the variable declaration with initialization, for binding X to V; and (2) the block construct for preventing X from shadowing variables in the original environment upon the completion of S_2.

Threads

SIMPLE's threads can be created and terminated dynamically, and can synchronize by acquiring and releasing re-entrant locks and by rendezvous. We discuss the seven rules giving the semantics of these operations below.

Thread creation. Threads can be created by any other threads using the "`spawn` S" construct. The spawn expression construct evaluates to the unique identifier of the newly created thread and, at the same time, a new thread cell is added into the configuration, initialized with the S statement and sharing the same environment with the parent thread. Note that the newly created `thread` cell is torn. That means that the remaining cells are added and initialized automatically as described in the definition of SIMPLE's configuration. This is part of $\mathbb{K}$'s configuration abstraction mechanism.

RULE $\left\langle \left\langle \dfrac{\texttt{spawn}\ S}{T} \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}} \right\rangle_{\mathsf{thread}} \ \dfrac{\cdot_{Bag}}{\left\langle \left\langle S \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}} \left\langle T \right\rangle_{\mathsf{id}} \right\rangle_{\mathsf{thread}}}$ requires $\texttt{fresh}\ (T)$

Thread termination. Dually to the above, when a thread terminates its assigned computation (the contents of its `k` cell) is empty, so the thread can be dissolved. However, since no discipline is imposed on how locks are acquired and released, it can be the case that a terminating thread still holds locks. Those locks must be released, so other threads attempting to acquire them do not deadlock. We achieve that by removing all the locks held by the terminating thread in its `holds` cell from the set of busy locks in the `busy` cell ($\texttt{keys}(H)$ returns the domain of the map H as a set, that is, only the locks themselves ignoring their multiplicity). As seen below, a lock is added to the `busy` cell as soon as it is acquired for the first time by a thread. The unique identifier of the terminated thread is also collected into the `terminated` cell, so the `join` construct knows which threads have terminated.

RULE $\dfrac{\left\langle \left\langle \cdot_K \right\rangle_{\mathsf{k}} \left\langle H \right\rangle_{\mathsf{holds}} \left\langle T \right\rangle_{\mathsf{id}} \right\rangle_{\mathsf{thread}}}{\cdot_{Bag}} \left\langle \dfrac{Busy}{Busy -_{Set} \texttt{keys}\ (H)} \right\rangle_{\mathsf{busy}} \left\langle \dfrac{\cdot_{Set}}{T} \right\rangle_{\mathsf{terminated}}$

Thread joining. Thread joining is now straightforward: all we need to do is to check whether the identifier of the thread to be joined is in the `terminated` cell. If yes, then the `join` statement dissolves and the joining thread continues normally; if not, then the joining thread gets stuck.

RULE $\left\langle \dfrac{\texttt{join}\ T\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle T \right\rangle_{\mathsf{terminated}}$

Acquire lock. There are two cases to distinguish when a thread attempts to acquire a lock (in SIMPLE any value can be used as a lock):
(1) The thread does not currently have the lock, in which case it has to take it provided that the lock is not already taken by another thread (see the side condition of the first rule).
(2) The thread already has the lock, in which case it just increments its counter for the lock (the locks are re-entrant). These two cases are captured by the two rules below:

RULE $\left\langle \dfrac{\texttt{acquire}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{\cdot_{Map}}{V \mapsto 0} \right\rangle_{\mathsf{holds}} \left\langle Busy\ \ \dfrac{\cdot_{Set}}{V} \right\rangle_{\mathsf{busy}}$ requires $(\neg_{Bool}(V\ \texttt{in}\ Busy))$ [acquire]

RULE $\left\langle \dfrac{\texttt{acquire}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle V \mapsto \dfrac{N}{N +_{Int} 1} \right\rangle_{\mathsf{holds}}$

Release lock. Similarly, there are two corresponding cases to distinguish when a thread releases a lock:
(1) The thread holds the lock more than once, in which case all it needs to do is to decrement the lock counter.
(2) The thread holds the lock only once, in which case it needs to remove it from its `holds` cell and also from the the shared `busy` cell, so other threads can acquire it if they need to.

RULE $\left\langle \dfrac{\texttt{release}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle V \mapsto \dfrac{N}{N -_{Int} 1} \right\rangle_{\mathsf{holds}}$ requires $N >_{Int} 0$

RULE $\left\langle \dfrac{\texttt{release}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{V \mapsto 0}{\cdot_{Map}} \right\rangle_{\mathsf{holds}} \left\langle \dfrac{V}{\cdot_{Set}} \right\rangle_{\mathsf{busy}}$

Rendezvous synchronization. In addition to synchronization through acquire and release of locks, SIMPLE also provides a construct for rendezvous synchronization. A thread whose next statement to execute is `rendezvous(`V`)` gets stuck until another thread reaches an identical statement; when that happens, the two threads drop their rendezvous statements and continue their executions. If three threads happen to have an identical rendezvous statement as their next statement, then precisely two of them will synchronize and the other will remain blocked until another thread reaches a similar rendezvous statement. The rule below is as simple as it can be. Note, however, that, again, it is $\mathbb{K}$'s mechanism for configuration abstraction that makes it work as desired: since the only cell which can multiply containing a `k` cell inside is the `thread` cell, the only way to concretize the rule below to the actual configuration of SIMPLE is to include each `k` cell in a `thread` cell.

RULE $\left\langle \dfrac{\texttt{rendezvous}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{\texttt{rendezvous}\ V\ \texttt{;}}{\cdot_K} \right\rangle_{\mathsf{k}}$ [rendezvous]

Auxiliary declarations and operations

In this section we define all the auxiliary constructs used in the above semantics.

Making declarations

The `mkDecls` auxiliary construct turns a list of identifiers and a list of values in a sequence of corresponding variable declarations.

SYNTAX $Decl ::= \texttt{mkDecls}\ (Ids, Vals)$ [function]

RULE $\dfrac{\texttt{mkDecls}\ ((X, Xs), (V, Vs))}{\texttt{var}\ X\ \texttt{=}\ V\ \texttt{;}\ \ \texttt{mkDecls}\ (Xs, Vs)}$

RULE $\dfrac{\texttt{mkDecls}\ (\cdot_{Ids}, \cdot_{Vals})}{\{\}}$

Location lookup

The operation below is straightforward. Note that we tag it with the same `lookup` tag as the variable lookup rule defined above. This way, both rules will be considered transitions when we include the `lookup` tag in the transition option of `kompile`.

SYNTAX $K ::= \texttt{lookup}\ (Int)$

RULE $\left\langle \dfrac{\texttt{lookup}\ (L)}{V} \right\rangle_{\mathsf{k}} \left\langle L \mapsto V \right\rangle_{\mathsf{store}}$ [lookup]

Environment recovery. We have already discussed the environment recovery auxiliary operation in the IMP++ tutorial:

SYNTAX $K ::= \texttt{env}\ (Map)$

RULE $\left\langle \dfrac{\texttt{env}\ (Env)}{\cdot_K} \right\rangle_{\mathsf{k}} \left\langle \dfrac{\text{—}}{Env} \right\rangle_{\mathsf{env}}$ [structural]

While theoretically sufficient, the basic definition for environment recovery alone is suboptimal. Consider a loop `while (E)S`, whose semantics (see above) was given by unrolling. `S` is a block. Then the semantics of blocks above, together with the unrolling semantics of the while loop, will yield a computation structure in the `k` cell that increasingly grows, adding a new environment recovery task right in front of the already existing sequence of similar environment recovery tasks (this phenomenon is similar to the "tail recursion" problem). Of course, when we have a sequence of environment recovery tasks, we only need to keep the last one. The elegant rule below does precisely that, thus avoiding the unnecessary computation explosion problem:

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

lvalue and loc

For convenience in giving the semantics of constructs like the increment and the assignment, that we want to operate the same way on variables and on array elements, we used an auxiliary `lvalue(`E`)` construct which was expected to evaluate to the lvalue of the expression E. This is only defined when E has an lvalue, that is, when E is either a variable or evaluates to an array element. `lvalue(`E`)` evaluates to a value of the form `loc(`L`)`, where L is the location where the value of E can be found; for clarity, we use `loc` to structurally distinguish natural numbers from location values. In giving semantics to `lvalue` there are two cases to consider. (1) If E is a variable, then all we need to do is to grab its location from the environment. (2) If E is an array element, then we first evaluate the array and its index in order to identify the exact location of the element of concern, and then return that location; the last rule below works because its preceding context declarations ensure that the array and its index are evaluated, and then the rule for array lookup (defined above) rewrites the evaluated array access construct to its corresponding store lookup operation.

SYNTAX $Exp ::= \texttt{lvalue}\ (K)$

SYNTAX $Val ::= \texttt{loc}\ (Int)$

RULE $\left\langle \texttt{lvalue}\ (\dfrac{X}{\texttt{loc}\ (L)}) \right\rangle_{\mathsf{k}} \left\langle X \mapsto L \right\rangle_{\mathsf{env}}$ [structural]

CONTEXT $\texttt{lvalue}\ (\text{—}[\square])$

CONTEXT $\texttt{lvalue}\ (\square[\text{—}])$

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

Initializing multiple locations

The following operation initializes a sequence of locations with the same value:

SYNTAX $Map ::= Int \ldots Int \mapsto K$ [function]

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

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

The semantics of SIMPLE is now complete. Make sure you kompile the definition with the right options in order to generate the desired model. No kompile options are needed if you only only want to execute the definition (and thus get an interpreter), but if you want to search for a different program behaviors then you need to kompile with the transition option including rule tags such as lookup, increment, acquire, etc. See the IMP++ tutorial for what the transition option means how to use it.

END MODULE