SIMPLE — Untyped

Grigore Roşu and Traian Florin Şerbănuţă
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 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 statements 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 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 references. Functions can have zero or more parameters and can return abruptly using a `return` statement. SIMPLE is call-by-value with static scoping. Function names evaluate to function references, which hereby become ordinary values in the language, same like the array references.
- Blocks with locals. SIMPLE allows variables to be declared anywhere, their scope being the most nested block containing them.
- 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 re-entriant locks can be acquired and released, as well as through rendezvouz.

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

Note: This $\mathbb{K}$ semantic definition of SIMPLE below is more commented than most of our other $\mathbb{K}$ definitions, because it is intended to be the first non-trivial $\mathbb{K}$ definition that the new user of the $\mathbb{K}$ framework sees, and because it actually can serve as a quick introduction to $\mathbb{K}$. The reader interested in reading more about $\mathbb{K}$ is refered to

> Grigore Roşu, Traian-Florin Şerbănuţă: An overview of the K semantic framework.
>
> Journal of Logic and Algebraic Programming, 79(6): 397-434 (2010)

MODULE SIMPLE-UNTYPED-SYNTAX
IMPORTS K-LATEX+PL-ID+PL-INT

Identifiers

The special identifier for the function "main" belongs to all programs. Each program may use additional identifiers, which need to be declared either automatically (when one uses an external parser) or manually (when one writes the program).

Id ::= `main`

Declarations

There are two types of declarations: for variables (including arrays) and for functions. As syntactic sugar, functions taking zero arguments are given separate syntax, so one needs not write the empty list constant.

Decl ::= `var` *List{Exp}* `;`
  | `function` *Id* `()` *Stmt*
  | `function` *Id* `(` *List{Id}* `)` *Stmt*

Expressions

The expression constructs below are standard. Increment (++) takes an expression rather than a variable because it can also increment an array element. 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. Like for the declaration of functions above, we have separate syntax for function calls with zero arguments. Note that almost all constructs are strict. Exceptions are the increment (since its first argument gets updated, so it cannot be evaluated), the function call with zero arguments (because it will get desugared), and the assignment which is only strict in its second argument (for the same reason as the increment).

Exp ::= *Bool*
  | *Id*
  | *Int*
  | ++ *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* `[` *List{Exp}* `]` [strict]
  | `sizeOf (` *Exp* `)` [strict]
  | *Exp* `()`
  | *Exp* `(` *List{Exp}* `)` [strict]
  | `read()`
  | *Exp* = *Exp* [strict(2)]

Statements

Most of the statement constructs are standard for imperative languages. We also syntactically distinguish between empty and non-empty blocks. 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 `acquire`, `release` and `randezvous`. 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 loops constructs are not strict in any arguments.

Stmt ::= {}
  | { *Stmts* }
  | *Exp* `;` [strict]
  | `if` *Exp* `then` *Stmt* `else` *Stmt* [strict(1)]
  | `if` *Exp* `then` *Stmt*
  | `while` *Exp* `do` *Stmt*
  | `for` *Id* = *Exp* `to` *Exp* `do` *Stmt*
  | `return` *Exp* `;` [strict]
  | `write (` *Exp* `) ;` [strict]
  | `try` *Stmt* `catch (` *Id* `)` *Stmt*
  | `throw` *Exp* `;` [strict]
  | `spawn` *Stmt*
  | `acquire` *Exp* `;` [strict]
  | `release` *Exp* `;` [strict]
  | `rendezvous` *Exp* `;` [strict]
Stmts ::= *Decl*
  | *Stmt*
  | *Stmts* *Stmts*

Lists

Currently we have to explicitly declare the syntactic lists (they will be eventually builtin). Note that they are associative, strict and hybrid, which means that once they reach the top of the computation each of their elements is evaluated to a result and the obtained list of results is automatically considered to be a result. We prefer to tag their units with the corresponding sort to avoid confusion.

List{Id} ::= *Id*
  | $\bullet_{List\{Id\}}$
  | *List{Id}* `,` *List{Id}* [assoc hybrid id: $\bullet_{List\{Id\}}$ strict]
List{Exp} ::= *Exp*
  | *List{Id}*
  | *List{Exp}* `,` *List{Exp}* [assoc ditto id: $\bullet_{List\{Id\}}$]
END MODULE

MODULE SIMPLE-UNTYPED-DESUGARED-SYNTAX
IMPORTS SIMPLE-UNTYPED-SYNTAX

This module 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 straighforward, except, perhaps, the one for multi-dimensional arrays. For the semantics, we can therefore assume that all functions take a list of arguments, that each conditional has both branches, that there are only `while` loops, and that each variable is declared alone and is initialized.

MACRO: `if` E `then` S = `if` E `then` S `else` {}
MACRO: `for` $X = E_1$ `to` E_2 `do` S = { `var` X `;` $X = E_1$ `;` `while` X <= E_2 `do` { S $X = X + 1$ `;` } }
MACRO: `function` F `()` S = `function` F `(` $\bullet_{List\{Id\}}$ `)` S
MACRO: E `()` = E `(` $\bullet_{List\{Id\}}$ `)`
MACRO: `var` E `,` E' `,` El `;` = `var` E `;` `var` E' `,` El `;`
MACRO: `var` X `;` = `var` $X = 0$ `;`
END MODULE

MODULE SIMPLE-UNTYPED-SEMANTICS
IMPORTS K+SIMPLE-UNTYPED-DESUGARED-SYNTAX

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*, *computations* and the *configuration*. The values are needed to know when to stop applying the heating rules corresponding to strictness or context declarations; unfortunately, values cannot be detected automatically, so one cannot avoid this step. The computation structures, or simply computations, are needed to properly include the language syntax under the K sort, as the $\mathbb{K}$ semantics operates only on terms of sort K; fortunately, this can be done mostly automatically, except for the special result computations. Finally, 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; the configuration could arguably be automatically inferred from the rules, but it turned out 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).

Val ::= *Bool*
  | *Int*
  | `array (` *Nat* `,` *Nat* `)`
  | λ*List{Id}* `.` *Stmt*
Exp ::= *Val*

Computations

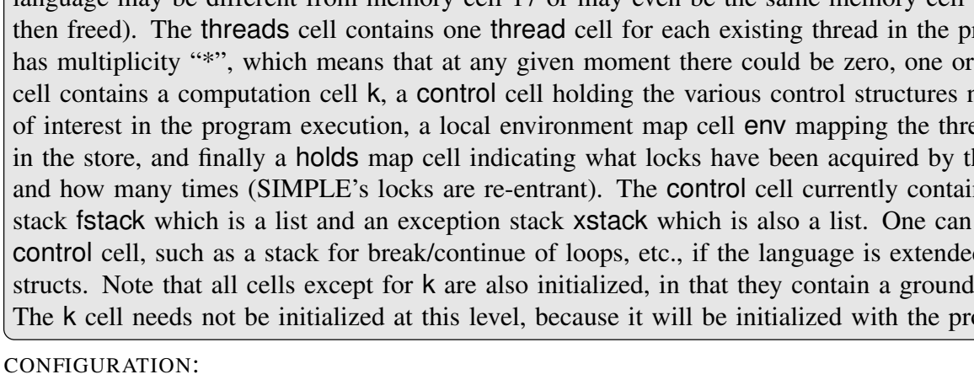
Recall that in $\mathbb{K}$, the *computational structures* or simply *computations* "swallow" the entire language syntax. Thus, every fragment of program is a particular computation, that is, a term of sort K. Also, recall that computations have quite a simple syntax: abstract syntax trees (where the original language constructs become AST node labels, i.e., constants of sort *KLabel*) extended with the special *task sequentialization* construct $K ::= K \curvearrowright K$. In addition to the sorts K and *KLabel*, $\mathbb{K}$ also has one more important sort related to computations, namely *KResult*. Distinguishing terms of sort *KResult* is not necessary in theory, but it is crucial in $\mathbb{K}$ tools, because they interpret such terms as irreducible and thus know when to stop the heating process associated to the strictness attributes. Below we state that all lists of values are terms of sort *KResult*. We use lists of values instead of values because, as discussed above, we want lists of expressions to evaluate to lists of values whenever scheduled for evaluation (by placing them on the top of the k cell).

KResult ::= *List{Val}*

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, input and output list cells, and a nextLoc cell holding a natural number indicating the next available location (one can think of locations as symbolic entities; e.g., location 17 means the 17'th allocated location, which in implementations of the language may be different from memory cell 17 or may even be the same memory cell that was previously allocated and then freed). 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 needs not be initialized at this level, because it will be initialized with the program to evaluate later.

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 is also initialized. The context declaration below says that the expression used for the initialization of the variable should be first evaluated (that expression can be any arithmetic expression, including array lookups, function calls, etc.). The semantic rule below matches resulting variable declarations of the form "`var` $X = V$`;`" (with V a value) 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 to hold the value V (indeed, the unit "$\bullet$", or nothing, is matched anywhere in the map—note the tears at both sides—and replaced with the mapping $L \mapsto V$), and binds X to L in the local environment shadowing previous declarations of X, if any. It is this possible shadowing of X which disallows us to use a similar technique for updating the environment as for updating the store, as we know that L is not already bound in the store when we add $L \mapsto V$. We prefer the approach used for updating the store whenever possible, because it offers more true concurrency than the latter; indeed, according to the concurrent semantics of K, the store is not frozen while $L \mapsto V$ 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. All the above happen in one 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.
Note: The "trick" with using a nextLoc cell to generate fresh locations is rather low level and hopefully temporary; we intend to soon allow instead a side-condition of the form "where L fresh".

CONTEXT: `var` $X = \square$ `;`

RULE: $\langle \frac{\texttt{var } X = V\texttt{;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{Env}{Env[L/X]} \rangle_{\mathsf{env}}$ $\langle \frac{\bullet_{Map}}{L \mapsto V} \rangle_{\mathsf{store}}$ $\langle \frac{L}{L +_{Nat} 1} \rangle_{\mathsf{nextLoc}}$

Array Declaration

SIMPLE allows multi-dimensional arrays. For semantic simplicity, we desugar them all into uni-dimensional arrays by code transformation. This way, we will only need to give semantics to uni-dimensional arrays. The expressions given as array dimensions are first evaluated and then the array is dynamically allocated the corresponding number of locations. Thus, we can dynamically allocate memory in SIMPLE by means of array declarations. First, we need to evaluate all the dimensions of the array:

CONTEXT: `var` X `[` — `,` $\square$ `,` — `] ;`

The $\mathbb{K}$ semantics of the uni-dimensional array declaration is somehow similar to the above declaration of ordinary variables. $N + 1$ 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. Recall that $L..L'$ is the list of locations between L and L' and that $L..L' \mapsto V$ initializes each location in the list $L..L'$ to value V.

RULE: $\langle \frac{\texttt{var } X\texttt{ [}N\texttt{] ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{Env}{Env[L/X]} \rangle_{\mathsf{env}}$ $\langle \frac{\bullet_{Map}}{L \mapsto \texttt{array}(L +_{Nat} 1, N)\ \ L +_{Nat} 1 \texttt{ .. } N +_{Nat} L +_{Nat} 1 \mapsto 0} \rangle_{\mathsf{store}}$ $\langle \frac{L}{N +_{Nat} L +_{Nat} 1} \rangle_{\mathsf{nextLoc}}$

To desugar multi-dimensional arrays, 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:

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

RULE: `var` X `[` N_1 `,` N_2 `,` Vl `] ;` $\rightharpoonup$ `var` X `[` N_1 `] ;` { `var $1 =` X `;` `for $2 =` 0 `to` $N_1 - 1$ `do` { `var` X `[` N_2 `,` Vl `] ;` `$1 [ $2 ] =` X `;` } }

Ideally, one would like to perform syntactic desugarings like above before we start the actual semantics. Unfortunately, that was not possible in this case because we need to first evaluate the dimension expressions of the multi-dimensional array. 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 need to evaluate expressions given originally 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. SIMPLE only allows function declarations at the top level of the program, like C. 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: $\langle \frac{\texttt{function } F\texttt{ (}Xl\texttt{) }S}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{Env}{Env[L/F]} \rangle_{\mathsf{env}}$ $\langle \frac{\bullet_{Map}}{L \mapsto \lambda Xl\,.\,S} \rangle_{\mathsf{store}}$ $\langle \frac{L}{L +_{Nat} 1} \rangle_{\mathsf{nextLoc}}$

When we are done with the first pass (pre-processing), the computation cell k contains only the token `execute` (see the module SIMPLE-UNTYPED in file `simple-untyped.k`, which adds the token `extend` at the end of the program) 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.

K ::= `execute`

RULE: $\langle \frac{\texttt{execute}}{\texttt{main () ;}} \rangle_{\mathsf{k}}$ $\langle Env \rangle_{\mathsf{env}}$ $\langle \frac{\bullet_{Map}}{Env} \rangle_{\mathsf{genv}}$

Expressions

We next define the $\mathbb{K}$ semantics of all the expression constructs, in the order in which their syntax was declared.

Variable lookup

When a variable X is the first computational task (note the rupture of the k cell to the right), and X is bound to some location L in the environment (note the rupture of the env cell at both sides), and L is mapped to some value V in the store, then rewrite X by V:

RULE: $\langle \frac{X}{V} \rangle_{\mathsf{k}}$ $\langle X \mapsto L \rangle_{\mathsf{env}}$ $\langle L \mapsto V \rangle_{\mathsf{store}}$

Variable/Array increment

This is tricky, because we want to allow both `++x` and `++a[5]`. Therefore, we need to extract the l-value of the expression to increment. To do that, we state that the expression to increment should be wrapped by the auxiliary "l-value" operation and then evaluated. The semantics of the auxiliary l-value operation is defined below. For now, all we need to know is that it takes an expression and evaluates to a location value, also introduced below with the auxiliary operations.

CONTEXT: ++ $\frac{\square}{\texttt{l-value (}\square\texttt{)}}$

RULE: $\langle \frac{\texttt{++ loc (}L\texttt{)}}{I +_{Int} 1} \rangle_{\mathsf{k}}$ $\langle L \mapsto \frac{I}{I +_{Int} 1} \rangle_{\mathsf{store}}$

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: $I_1 + I_2 \Rightarrow I_1 +_{Int} I_2$

RULE: $I_1 - I_2 \Rightarrow I_1 -_{Int} I_2$

RULE: $I_1 \star I_2 \Rightarrow I_1 \star_{Int} I_2$

RULE: $I_1 \,/\, I_2 \Rightarrow I_1 \div_{Int} I_2$ when $I_2 \neq_{Bool} 0$

RULE: $I_1 \,\%\, I_2 \Rightarrow I_1 \,\%_{Int}\, I_2$ when $I_2 \neq_{Bool} 0$

RULE: $- I \Rightarrow -_{Int} I$

RULE: $I_1 < I_2 \Rightarrow I_1 <_{Int} I_2$

RULE: $I_1 \texttt{<=} I_2 \Rightarrow I_1 \leq_{Int} I_2$

RULE: $I_1 > I_2 \Rightarrow I_1 >_{Int} I_2$

RULE: $I_1 \texttt{>=} I_2 \Rightarrow I_1 \geq_{Int} I_2$

RULE: $V_1 \texttt{==} V_2 \Rightarrow V_1 =_{Bool} V_2$

RULE: $V_1 \texttt{!=} V_2 \Rightarrow V_1 \neq_{Bool} V_2$

RULE: T_1 `and` $T_2 \Rightarrow T_1 \wedge_{Bool} T_2$

RULE: T_1 `or` $T_2 \Rightarrow T_1 \vee_{Bool} T_2$

RULE: `not` $T \Rightarrow \neg_{Bool} T$

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 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. Recall that "–" is an anonymous variable in $\mathbb{K}$ matching any subterm (like in Prolog); informally, "there is something there but we don't care what". The semantics of the `lookup` operation is straightforward.

RULE: V `[` N_1 `,` N_2 `,` Vl `]` $\rightharpoonup$ V `[` N_1 `] [` N_2 `,` Vl `]`

RULE: `array (` L `,` — `) [` N `]` $\rightharpoonup$ `lookup (` $N +_{Int} L$ `)`

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

RULE: $\langle \frac{\texttt{lookup (}L\texttt{)}}{V} \rangle_{\mathsf{k}}$ $\langle L \mapsto V \rangle_{\mathsf{store}}$

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: `sizeOf ( array (` — `,` N `) )` $\Rightarrow N$

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 0;`" (here 0 is a default return value 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. 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).

ListItem ::= `(` *Map* `,` *K* `,` *Bag* `)`

RULE: $\langle \frac{\lambda Xl\,.\,S\texttt{ (}Vl\texttt{)} \curvearrowright K}{S \curvearrowright \texttt{return 0 ;}} \rangle_{\mathsf{k}}$ $\langle \langle \frac{\bullet_{List}}{(Env, K, C)} \rangle_{\mathsf{fstack}}\ C \rangle_{\mathsf{control}}$ $\langle \frac{Env}{GEnv\,[\,N \texttt{ .. } N +_{Nat} |Xl| \,/\, Xl\,]} \rangle_{\mathsf{env}}$ $\langle GEnv \rangle_{\mathsf{genv}}$ $\langle \frac{\bullet_{Map}}{N \texttt{ .. } N +_{Nat} |Xl| \mapsto Vl} \rangle_{\mathsf{store}}$ $\langle \frac{N}{N +_{Nat} |Xl|} \rangle_{\mathsf{nextLoc}}$

RULE: $\langle \frac{\texttt{return } V\texttt{ ;} \curvearrowright \text{—}}{V \curvearrowright K} \rangle_{\mathsf{k}}$ $\langle \langle \frac{(Env, K, C)}{\bullet_{List}} \rangle_{\mathsf{fstack}}\ \frac{\text{—}}{C} \rangle_{\mathsf{control}}$ $\langle \frac{\text{—}}{Env} \rangle_{\mathsf{env}}$

Read

The `read()` expression construct simply evaluates to the next input value, at the same time discarding the input value from the in cell. In an implementation of the SIMPLE language, one would likely want to make the input buffer interactive, prompting the user to provide a new value whenever needed. In a semantics, however, it is acceptable to assume that all the input values are given before the program is executed; thus, the rewrite process gets stuck if the next computation item is a `read()` expression construct and no input item is available.

RULE: $\langle \frac{\texttt{read()}}{I} \rangle_{\mathsf{k}}$ $\langle \frac{I}{\bullet_{List}} \rangle_{\mathsf{in}}$

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 l-value of the left-hand-side expression that appears in an assignment. Like for the increment, all we need to know is that `l-value(...)` eventually evaluates to a location value `loc(...)`.

CONTEXT: $\frac{\square}{\texttt{l-value (}\square\texttt{)}} = \text{—}$

RULE: $\langle \frac{\texttt{loc (}L\texttt{) = }V}{V} \rangle_{\mathsf{k}}$ $\langle L \mapsto \frac{\text{—}}{V} \rangle_{\mathsf{store}}$

Statements

We next define the $\mathbb{K}$ semantics of statements, also in the order their syntax was given.

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; the environment recovery computation task `env(...)` is used in other places, too, so we define it together with other auxiliary operations at the end of the semantics. 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 doesn't want them to count as computational steps.

RULE: $\{\} \rightharpoonup \bullet_K$

RULE: $\langle \frac{\{\,Ss\,\}}{Ss \curvearrowright \texttt{env (}Env\texttt{)}} \rangle_{\mathsf{k}}$ $\langle Env \rangle_{\mathsf{env}}$

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, so it is the one that we follow in general. Recall that variables in rules are typically represented as pointers in implementations of rewriting, so "storing/recovering the current environment in/from the computation" are computationally very cheap operations.

Sequential composition

Sequential composition is desugared into $\mathbb{K}$'s builtin sequentialization operation (recall that 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 computational, so it does count as a computational step. One can make it structural if one does not want 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 other explicit or implicit additional rules.

RULE: $S_1\ S_2 \Rightarrow S_1 \curvearrowright S_2$

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: V `;` $\Rightarrow \bullet_K$

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: `if` true `then` S `else` — $\Rightarrow S$

RULE: `if` false `then` — `else` $S \Rightarrow S$

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: $\langle \frac{\texttt{while } B\texttt{ do }S}{\texttt{if } B\texttt{ then \{ }S\texttt{ while }B\texttt{ do }S\texttt{ \} else \{\}}} \rangle_{\mathsf{k}}$

Write

The write statement appends its evaluated argument (recall that `write` was declared strict) to the output list. Like in the case of the `read()` construct, implementations of SIMPLE may choose to display the output incrementally, as it is being generated; in semantics, we do not need to worry about that and simply collect all the output in the out cell.

RULE: $\langle \frac{\texttt{write (}I\texttt{) ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{\bullet_{List}}{I} \rangle_{\mathsf{out}}$

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 achived by pushing/popping the entire current control context onto the exception stack. The three rules below modularly do precisely all the above.

ListItem ::= `(` *Id* `,` *Stmt* `,` *K* `,` *Map* `,` *Bag* `)`
K ::= `popx`

RULE: $\langle \frac{\texttt{try } S_1\texttt{ catch (}X\texttt{) }S_2 \curvearrowright K}{S_1 \curvearrowright \texttt{popx}} \rangle_{\mathsf{k}}$ $\langle \langle \frac{\bullet_{List}}{(X, S_2, K, Env, C)} \rangle_{\mathsf{xstack}}\ C \rangle_{\mathsf{control}}$ $\langle Env \rangle_{\mathsf{env}}$

RULE: $\langle \frac{\texttt{popx}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{\text{—}}{\bullet_{List}} \rangle_{\mathsf{xstack}}$

RULE: $\langle \frac{\texttt{throw } V\texttt{ ;} \curvearrowright \text{—}}{S_2 \curvearrowright \texttt{env (}Env\texttt{)} \curvearrowright K} \rangle_{\mathsf{k}}$ $\langle \langle \frac{(X, S_2, K, Env, C)}{\bullet_{List}} \rangle_{\mathsf{xstack}}\ \frac{\text{—}}{C} \rangle_{\mathsf{control}}$ $\langle \frac{\text{—}}{Env\,[\,L \,/\, X\,]} \rangle_{\mathsf{env}}$ $\langle \frac{\bullet_{Map}}{L \mapsto V} \rangle_{\mathsf{store}}$ $\langle \frac{L}{L +_{Nat} 1} \rangle_{\mathsf{nextLoc}}$

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 7 rules giving the semantics of these operations below.

Thread creation

Threads can be created by any other threads using the "`spawn(`S`)`" statement. The spawn statement in the creating thread dissolves, at the same time adding a new thread cell to the configuration sharing the same environment. Note that the created thread cell is torn. That means that the remining cells are added and initialized automatically as described in the definition of SIMPLE's configuration. This is part of $\mathbb{K}$'s configuration abstraction/concretization mechanism.

RULE: $\langle \langle \frac{\texttt{spawn } S}{\bullet_K} \rangle_{\mathsf{k}}\ \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{thread}}$ $\frac{\bullet_{Bag}}{\langle \langle S \rangle_{\mathsf{k}}\ \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{thread}}}$

Thread termination

Dually to the above, when a thread terminates its assigned computation (so the contents of its k cell is empty) it 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 achive 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 (`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.

RULE: $\frac{\langle \langle \bullet_K \rangle_{\mathsf{k}}\ \langle H \rangle_{\mathsf{holds}} \rangle_{\mathsf{thread}}}{\bullet_{Bag}}$ $\langle \frac{Busy}{Busy -_{Set} \texttt{keys } H} \rangle_{\mathsf{busy}}$

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: $\langle \frac{\texttt{acquire } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{\bullet_{Map}}{V \mapsto 0} \rangle_{\mathsf{holds}}$ $\langle Busy\ \frac{\bullet_{Set}}{V} \rangle_{\mathsf{busy}}$ when $\neg_{Bool}\ V$ `in` $Busy$

RULE: $\langle \frac{\texttt{acquire } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle V \mapsto \frac{N}{N +_{Nat} 1} \rangle_{\mathsf{holds}}$

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: $\langle \frac{\texttt{release } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle V \mapsto \frac{N}{N -_{Int} 1} \rangle_{\mathsf{holds}}$ when $N >_{Nat} 0$

RULE: $\langle \frac{\texttt{release } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{V \mapsto 0}{\bullet_{Map}} \rangle_{\mathsf{holds}}$ $\langle \frac{V}{\bullet_{Set}} \rangle_{\mathsf{busy}}$

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 get. 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: $\langle \frac{\texttt{rendezvous } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{\texttt{rendezvous } V\texttt{ ;}}{\bullet_K} \rangle_{\mathsf{k}}$

Auxiliary declarations and operations

Environment recovery

We needed the environment recovery operation twice in our semantics above, the first time for blocks and the second time for exceptions. Its definition is straightforward:

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

RULE: $\langle \frac{\texttt{env (}Env\texttt{)}}{\bullet_K} \rangle_{\mathsf{k}}$ $\langle \frac{\text{—}}{Env} \rangle_{\mathsf{env}}$

In fact, the above follows a common convention in $\mathbb{K}$ for recovery operations of cell contents. More precisely, the conventional meaning of a computation task of the form `cell(`C`)` that reaches the top of the computation is that the current contents of cell cell is discarded and gets replaced with C. We did not add support for these special computation tasks in our current implementation of $\mathbb{K}$, so we need to define them as above.

l-value 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 auxilliary `l-value(`E`)` construct which was expected to evaluate to the l-value of the expression E. This is only defined when E has an l-value, that is, when E is either a variable or evauates to an array element. The value that `l-value(`E`)` evaluates to has the form `loc(`L`)`, where L is a location, i.e., a natural number; we prefer to structually distinguish natural numbers from location values for clarity. Therefore, 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.

Exp ::= `l-value (` *K* `)`
Val ::= `loc (` *Nat* `)`

RULE: $\langle \frac{\texttt{l-value (}X\texttt{)}}{\texttt{loc (}L\texttt{)}} \rangle_{\mathsf{k}}$ $\langle X \mapsto L \rangle_{\mathsf{env}}$

CONTEXT: `l-value (` — `[` — `,` $\square$ `,` — `] )`
CONTEXT: `l-value (` $\square$ `[` — `] )`

RULE: $\langle \frac{\texttt{l-value (lookup (}L\texttt{))}}{\texttt{loc (}L\texttt{)}} \rangle_{\mathsf{k}}$

Length

The following operation calculates the length of a list of identifiers. We make the two rules structural so thay they do not count as computations.

Nat ::= |*List{Id}*|
RULE: $|\bullet_{List\{Id\}}| \rightharpoonup 0$

RULE: $|X\ ,\ Xl| \rightharpoonup 1 +_{Nat} |Xl|$

Sequences of locations

The following operation expands to the list of natural numbers between two given numbers. The first number is expected to be no larger than the second. The two rules below are structural, for the same reason as above.

List{K} ::= *Nat* `..` *Nat*
RULE: $N_1 \texttt{ .. } N_1 \rightharpoonup \bullet_{List\{K\}}$

RULE: $N_1 \texttt{ .. } N_2 \rightharpoonup N_1\ ,\ N_1 +_{Nat} 1 \texttt{ .. } N_2$ when $N_1 <_{Nat} N_2$

Lists of values

Currently we have to explicitly declare the syntactic lists (they will be eventually builtin). We need lists of values as a separate syntactic category because we want to allow lists of expressions (e.g., function arguments) to evalaute to such lists of values. Note also that we defined lists of values as $\mathbb{K}$ results (at the begining of the SIMPLE semantics module).

List{Val} ::= *List{Val}* `,` *List{Val}* [assoc ditto id: $\bullet_{List\{Id\}}$]
  | *Val*
List{Exp} ::= *List{Val}*
END MODULE

MODULE SIMPLE-UNTYPED
IMPORTS SIMPLE-UNTYPED-INPUT-TEST+SIMPLE-UNTYPED-PROGRAMS+SIMPLE-UNTYPED-SEMANTICS
Bag ::= `run (` *KLabel* `)`

MACRO: `run (` *PgmName* `)` = $\langle \langle PgmName\texttt{ (}\bullet_{List\{K\}}\texttt{)} \curvearrowright \texttt{execute} \rangle_{\mathsf{k}}\ \langle \texttt{input} \rangle_{\mathsf{in}} \rangle_{\mathsf{T}}$

END MODULE