Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 3–56

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

$\mathbb{K}$ Overview and SIMPLE Case Study

Grigore Roșu[1]

University of Illinois at Urbana-Champaign

Traian Florin Șerbănuță[2]

University Alexandru Ioan Cuza of Iași

Abstract

This paper gives an overview of the tool-supported $\mathbb{K}$ framework for semantics-based programming language design and formal analysis. $\mathbb{K}$ provides a convenient notation for modularly defining the syntax and the semantics of a programming language, together with a series of tools based on these, including a parser and an interpreter. A case study is also discussed, namely the $\mathbb{K}$ definition of the dynamic and static semantics of SIMPLE, a non-trivial imperative programming language. The material discussed in this paper was presented in an invited talk at the $\mathbb{K}$'11 workshop.

Keywords: Programming Languages, Rewriting-Based Semantics, $\mathbb{K}$

1 Introduction

Introduced by the first author in 2003 [40] for teaching a programming languages class, and continuously refined and developed ever since, the $\mathbb{K}$ framework [43,52] is a programming language definitional framework based on rewriting which brings together the strengths of existing frameworks (expressiveness, modularity, concurrency, and simplicity) while avoiding their weaknesses. The $\mathbb{K}$ framework consists of (1) the $\mathbb{K}$ *technique*, which can be, and has already been used to define real-life programming languages, such as Java, C, and Scheme, and program analysis tools (see Section 5 and the references there), and of (2) $\mathbb{K}$ *rewriting*, a rewriting semantics allowing $\mathbb{K}$ definitions to capture true concurrency with sharing of resources.

To give semantics to programming language constructs, the $\mathbb{K}$ framework relies on computational structures, configurations, and $\mathbb{K}$ rules. Computational structures, which are more simply called *computations* and which inspired the name "$\mathbb{K}$" of the

[1] grosu@illinois.edu

[2] traian.serbanuta@info.uaic.ro

http://dx.doi.org/10.1016/j.entcs.2014.05.002

framework, are sequences of computational tasks, where each computational task is a term over the possibly extended language syntax; computations are typically used to handle the sequential fragment of the defined programming language and the evaluation strategies of the various language constructs. *Configurations* of running programs are represented as bags (multisets) of nested cells, with a great potential for concurrency and modularity. $\mathbb{K}$ *rules* distinguish themselves by specifying only what is needed from a configuration and by clearly identifying what changes; thus, they are more concise, more modular, and more concurrent than regular rewrite rules.

If one ignores its concurrent semantics, $\mathbb{K}$ can be seen as a notation within rewriting logic [31], the same as most other semantic frameworks, such as natural (or big-step) semantics, (small-step) SOS, Modular SOS, reduction semantics with evaluation contexts, and so on [55]. However, unlike these other semantic frameworks enumerated above, $\mathbb{K}$ cannot be easily captured step-for-step in rewriting logic, due to its enhanced concurrency which is best described in terms of ideas from graph rewriting [43,52].

We will only focus on the essential/fundamental aspects of the $\mathbb{K}$ framework, not on particular implementations of it; the $\mathbb{K}$ Primer included in this volume [53] provides a more in-depth view of the current implementation. Thus, we do not insist on implementation-specific annotations of a $\mathbb{K}$ definition in what follows. Instead, we refer the reader interested in implementation details to the current distribution of the $\mathbb{K}$ tool (reachable from `k-framework.org`), where commented versions of the subsequent $\mathbb{K}$ definitions can be found. Moreover, we do not insist on executability aspects of $\mathbb{K}$ either, that is, on foundational aspects of a definition which make it (efficiently) executable. We limit ourselves to purely theoretical and high-level aspects of $\mathbb{K}$ in this paper.

The overview of the $\mathbb{K}$ framework presented in Section 2 is complemented by the complete literate definitions of the dynamic (Section 3) and static (Section 4) semantics of SIMPLE, a non-trivial programming language characteristic of the imperative programming paradigm. These sections demonstrate both the expressiveness and the modularity of the framework, as well as give compelling evidence that $\mathbb{K}$ can scale to larger languages. References to other research projects making use of $\mathbb{K}$ in achieving their goals are presented in Section 5. Section 6 concludes.

2 Overview of the $\mathbb{K}$ Framework

Here we give an overview of $\mathbb{K}$, focusing on its overall capabilities and objectives. As one may expect, a relatively new and actively used framework changes often due to user requests/complaints. $\mathbb{K}$ is no exception. In this section we also focus on recent developments, notations and terminology. We will attempt to justify our design decisions where appropriate, because we believe that other semantic framework designers may need to take similar or related decisions. For a more technical (but older) presentation of $\mathbb{K}$ we refer the reader to [43]. Sections 3 and 4 discuss the complete $\mathbb{K}$ definitions of the SIMPLE programming language and of its type system, both part of the 𝕂 tool distribution, which we will also use in this section to illustrate 𝕂's features.

𝕂 is a programming language definitional framework based on context insensitive term rewriting. 𝕂 builds upon the following three main ideas:

(i) Flatten syntax into special *computational structures*, called *computations* for simplicity, which include abstract syntax and are reminiscent of refocusing [14] in reduction semantics with evaluation contexts [20], of continuations [39], and to computations in monads [34].

(ii) Represent the state, or the configuration, of an executing program as a potentially nested structure of labeled *cells*. This is reminiscent of solutions in the chemical abstract machine (CHAM) [10]. 𝕂 rewrite rules (explained next) then iteratively transform such configurations, starting with a configuration holding the original program and ending with a configuration holding the result.

(iii) Give semantics to language constructs using 𝕂 *rewrite rules*, typically a small number of independent rules for each language construct. The precise semantics of 𝕂 is given in terms of graph rewriting intuitions, in order to properly yield truly concurrent language semantics (see Section 2.5 for more details). Moreover, 𝕂 rules are split into *structural* and *computational*, the former's role being only to rearrange the configuration so that the latter can match and apply. This is reminiscent of rewriting logic's split of sentences into equations and rules [32], and also to the distinction between heating/cooling and reaction rules in the CHAM [10].

𝕂 additionally brings a series of semantic innovations and notations dictated by practical needs. For example, 𝕂 rules are regarded as *transactions*, stating what is only read, what is both read and written, and what is irrelevant. This allows for true concurrency even in the presence of sharing. Also, a *configuration abstraction* mechanism allows definitions to be both compact and modular, often requiring no changes to existing rules when the configuration changes.

2.1 Case study: the SIMPLE language

Throughout this paper, we will introduce and exemplify 𝕂 using the definition of the SIMPLE language, which we briefly describe here. 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 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 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 are ordinary values in the language, like the array references.
- Blocks with locals. SIMPLE variables can be declared anywhere, their scope being from their declaration place until the end of the enclosing block.
- Input/Output. The expression `read()` evaluates to the next value in the input buffer, and the statement `print(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.

2.2 𝕂 Syntax

The 𝕂 syntax of languages, calculi or systems, as well as the additional syntax needed for the semantics of these, is defined using context-free grammars (CFG) or, equivalently, algebraic signatures written using the mixfix notation (i.e., operation names include underscores "_" as argument placeholders) [23,22,12]. We take the freedom to borrow from the algebraic universe any structures of interest on a by-need basis. In this paper we use *List*{*Nonterminal*, *terminal*} to refer to the nonterminal corresponding to *terminal*-separated lists of *Nonterminal* elements; for example, *List*{*Exp*,@} stands for @-separated lists of expressions. We skip the terminal when it is a comma; e.g., *List*{*Exp*} stands for comma-separated lists of expressions. In this paper and in 𝕂 in general, we uniformly use a dot "$\cdot$", read "nothing" and possibly tagged with its type as a subscript, as the unit of all structures mentioned above. If one prefers a different unit then one should mention it as an additional argument to *List*, e.g., *List*{*Exp*,@, `nil`}, etc.

Syntax definition and parsing are difficult topics in their full generality, which have been extensively researched and implemented over several decades. Implementations of 𝕂 would likely make use of existing techniques and tools for defining syntax and for parsing. However, at its very core, 𝕂 is actually not concerned with concrete syntax at all. More precisely, the syntax of 𝕂 currently consists of one syntactic category K for computational structures, or compactly just computations[3], i.e., structures which have the capability to compute when put in the right context,

[3] The syntax of 𝕂 can be extended to include other syntactic categories besides K. There are several current 𝕂 projects which appear to need such extensions, but here we limit ourselves to a minimal setting.

together with another syntactic category, *KLabel*, for abstract syntax tree labels:

$$\begin{array}{rcll} K & ::= & \mathit{KLabel}\;(\;\mathit{List}\{K\}\;) \;\mid\; \mathit{List}\{K,\curvearrowright\} & \text{(generic)} \\ \mathit{KLabel} & ::= & 0 \mid 1 \mid 2 \mid \cdots \mid \texttt{while(\_)\_} \mid \texttt{\{\_\}} \mid \cdots & \text{(language-specific)} \end{array}$$

A programming language, calculus or system syntax, including constants such as primitive values, is eventually regarded as a set of K labels by simply associating a unique K label to each production and discarding all the concrete syntactic categories. This way, any program or fragment of program can be regarded (for semantic reasons) as a K *abstract syntax tree (KAST)* whose nodes are K labels and whose leaves are “•”. By default, we follow the mixfix notational philosophy [23,22,12] when choosing the label names, but one is free to use any naming conventions. With our convention, for example, the fragment of SIMPLE program “`while(x > 0) {x = x - 1;}`” is regarded as the KAST “`while(_)_(_>_(x(•), 0(•)), {_}(_=_(x(•), _-_(x(•), 1(•)))))`”.

The KAST notation is convenient for both theoretical and practical reasons. Theoretically, it allows one to give language-independent and thus modular semantics to constructs that require one to visit the entire language syntax, such as substitution or code generation, by simply giving their semantics in terms of KASTs and thus not worrying about the concrete language syntax. Practically, it gives a uniform means to separate syntactic concerns from semantic ones, leaving the translation from concrete syntax to KAST to tools. $\mathbb{K}$ tools may (and do [53]) provide more user-friendly means to define language syntax than as sets of K labels.

In addition to capturing language/calculus/system syntax as KAST structures as explained above, the K syntactic category also provides a *task sequentialization* list construct, written “$\curvearrowright$” and read “followed by”. If t_1, t_2, ..., t_n are computations, then $t_1 \curvearrowright t_2 \curvearrowright \cdots \curvearrowright t_n$ can be thought of as the computation consisting of t_1 followed by t_2 followed by $\ldots$, followed by t_n. As seen shortly in Section 2.4, “$\curvearrowright$” plays a crucial role in defining evaluation strategies for language constructs. For example, if $s_1, s_2 \in K$ are KASTs corresponding to two statements in SIMPLE, then the semantics of sequential composition will reduce “$s_1\ s_2$” to “$s_1 \curvearrowright s_2$”, which will further be processed using other rules as expected: first s_1 will be fully evaluated and then s_2 will be evaluated. Similarly, if $e_1, e_2 \in K$ are KASTs corresponding to two expressions in SIMPLE, then the rewrite rules defining the evaluation strategy of addition will allow the expression “$e_1\texttt{+}e_2$” to non-deterministically rewrite to either “$e_1 \curvearrowright \square\texttt{+}e_2$” or “$e_2 \curvearrowright e_1\texttt{+}\square$”, where $\_\texttt{+}\square$ and $\square\texttt{+}\_$ are two new $\mathbb{K}$ labels specifically added for this purpose (in other words, $\square$ is part of a label name and not an explicit “hole” terminal). Other evaluation strategies are also possible and easy to imagine, as well as techniques reminiscent of refocusing [14] to support defining evaluation contexts.

2.3 $\mathbb{K}$ Configurations

A programming language semantics is typically driven by the syntax, but it often needs additional semantic data in order to properly capture the desired semantics of each language construct. Such data may include a program environment mapping program variables to memory locations, a store mapping memory locations to values,

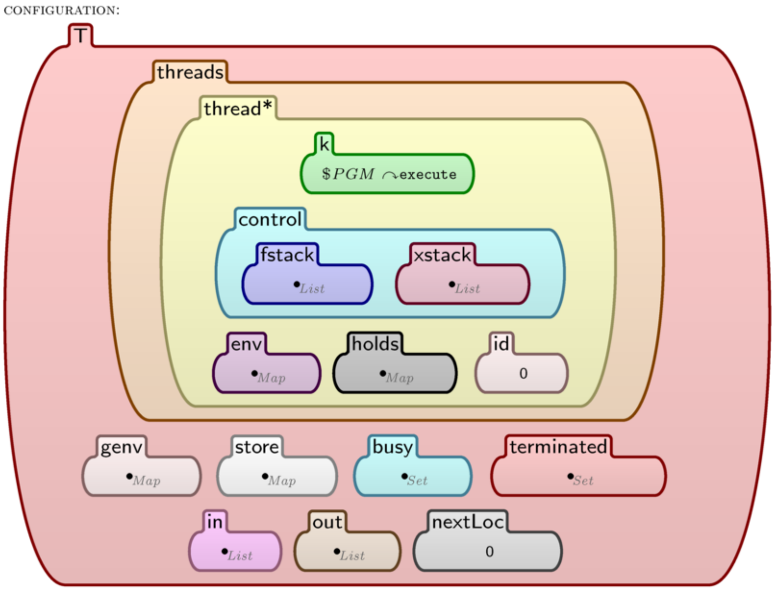


Fig. 1. The 𝕂 configuration of SIMPLE

one or more stacks for functions and exceptions, a multi-set (or bag) of threads, a set of held locks associated to each thread, and so on. Such list, set, multi-set, and map structures are well-known algebraic structures, with many papers describing various ways to define them as algebraic or logical structures (see, e.g., the CASL [35] and Maude [12] manuals). To distinguish the various semantic components from each other, in 𝕂 we "wrap" them within suggestively named *cells* when we structure them together in a configuration. These cells are nothing but constructors taking the desired structure and yielding a configuration item. For example, a cell called store can be defined as an operation

$$\mathsf{store} : \mathit{Map} \to \mathit{CfgItem}$$

where *Map* is the sort of maps from say natural numbers to integer numbers. Cells can be nested. We do not insist on how one can/should define configurations, as different implementations/realizations/encodings of 𝕂 may choose different representations and notations. The important point is that configurations, no matter how complex, can be defined as appropriate algebraic specifications. Moreover, 𝕂 assumes such configurations to be defined upfront, before the semantic rules are given, since the structure of program configurations is an important aspect that gives 𝕂 its modularity (as discussed in the next section).

We next informally discuss the 𝕂 configuration of the SIMPLE language, depicted graphically in Figure 1. Recall from Section 2.1 that SIMPLE has functions with abrupt termination, exceptions, dynamic threads with lock synchronization and with memory sharing, and input/output. Its configuration 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 smaller languages, in 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, signified graphically in the configuration by the "*" attached to the thread label, which specifies the *multiplicity* of the cell, i.e., that at any given moment there could be either 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 each lock has been acquired without being released (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 (KAST of the) program to be executed, as indicated by the *$PGM* placeholder, followed by the `execute` task (whose semantics will be given in Section 3).

A configuration declaration in 𝕂 does several things at the same time, in a compact and intuitive way:

(i) It defines an algebraic signature for configurations, as explained in the first paragraph of this section;

(ii) It tells how to initialize the configuration;

(iii) It gives a basis for concretizing semantic rewrite rules, which for modularity reasons can be given more abstractly, as seen in the next section.

Implementations of 𝕂 may choose different ways to define configurations, and additional ways to initialize them. For example, our current implementation uses XML to delimit cells (e.g., `<threads>...</threads>`), it allows users to further initialize the configuration by means of custom *$PGM*-like placeholders, and even allows for connecting certain cells to the standard input/output in order to obtain realistic interpreters when executing 𝕂 definitions [53].

𝕂 provides support for defining complex configurations, but that does not mean that 𝕂 definitions are always expected to use it, not even that 𝕂 encourages environment-based semantics. For example, to define a purely syntactic, substitution-based semantics of a language or calculus, we only need a one-cell configuration initialized with *$PGM*. The reader is referred to the 𝕂 tool distribution for several substitution-based definitions.

2.4 $\mathbb{K}$ Rules

As seen in Section 2.3, the configuration is initialized by placing the target program at its specified position and initializing any other cell with its declared contents. From here on, the $\mathbb{K}$ rewrite rules giving the language semantics (non-deterministically and concurrently) match and apply, potentially generating any possible behavior of the target program. There are two types of rules in $\mathbb{K}$, namely *structural* and *computational* rules. Intuitively, structural rules decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. In other words, the structural rules do not count as observable steps, while the computational rules do. The formal semantic difference between the two is discussed in Section 2.5. Unless explicitly tagged "structural", we assume $\mathbb{K}$ rules to be computational by default.

Structural rules

We start by discussing an important category of structural $\mathbb{K}$ rules, the *heating/cooling* rules. Our terminology for these rules is inspired from the chemical abstract machine (CHAM) [10]. One is free to use different arrows for heating/cooling in particular or for structural rules in general, for example $\rightharpoonup$ or $\rightharpoondown$ instead of $\Rightarrow$ like in the CHAM, but we prefer to not enforce any particular notation. In our current $\mathbb{K}$ tool, for example, we use the rule tag called "structural". Heating/cooling rules have the role to re-arrange the computation according to the desired evaluation strategies, so that the "hot spots" are pushed to the front of the computation structure. The overall effect of this structural process is that, unlike in reduction semantics with evaluation contexts [20], rewriting in $\mathbb{K}$ needs *not* be context-sensitive. Consider, for example, the addition operation in SIMPLE, which is intended to be non-deterministic. We then add two pairs of reversible structural rules for it, namely

RULE
$$\frac{E_1 + E_2}{E_1 \curvearrowright \square + E_2}$$
[structural]

RULE
$$\frac{E_1 \curvearrowright \square + E_2}{E_1 + E_2}$$
[structural]

RULE
$$\frac{E_1 + E_2}{E_2 \curvearrowright E_1 + \square}$$
[structural]

RULE
$$\frac{E_2 \curvearrowright E_1 + \square}{E_1 + E_2}$$
[structural]

Note that we used a two-dimensional notation for rules, with the left term of the rewrite rule above a horizontal line and the right term underneath the line. This notation is an instance of a more general principle that will be explained shortly. The first and third rules say that we can at any moment "heat" an addition by pulling any of its two arguments and schedule it for reduction, thus *splitting* syntax into an *evaluation context* (the tail of the resulting sequence of computational tasks) and a *redex* (the head of the sequence); we call such rules "heating" rules. The second and fourth rules say that we can at any moment "cool" the addition by plugging its heated argument back into its context. This way, we can non-deterministically explore all possible orders in which the two arguments of a sum can be reduced (all their evaluation interleavings).

There are two problems with writing rules like the ones above. First, writing such rules is tedious, boring and error-prone. To address this problem, in $\mathbb{K}$ we adopted a simple and intuitive syntactic annotation:

SYNTAX $\mathit{Exp} ::= \mathit{Exp}$ `+` Exp [strict]

The "strict" annotation, or attribute, associated to a syntactic construct states that it is intended to be (non-deterministically) strict in its arguments, and is equivalent to giving the four rules above. If one wants a construct to be strict only in some of its arguments, then one is expected to enumerate the positions of those arguments in parentheses after the "strict" attribute; for example, a construct which is intended to be strict only in its first and third arguments would be annotated with "strict(1 3)". Sometimes one needs to evaluate the arguments of a construct in a given order, say from left to right. Then one can use the attribute "seqstrict" (from sequentially strict) instead of "strict". It is not hard to see how all these can be easily translated into structural heating/cooling rules like those shown above. For example, if the sum construct were annotated "seqstrict", then the last two rules above would require E_1 to be evaluated (that is, to be an integer):

RULE
$$\frac{V_1 \texttt{ + } E_2}{E_2 \curvearrowright V_1 \texttt{ + } \square}$$
[structural]

RULE
$$\frac{E_2 \curvearrowright V_1 \texttt{ + } \square}{V_1 \texttt{ + } E_2}$$
[structural]

The second problem with writing heating/cooling rules like above is that they are not immediately executable, as they lead to non-termination. This is similar to how rules stating the commutativity of certain constructs may lead to non-termination if executed naively. The same way the theory of term rewriting allows for rewriting *modulo* axioms like commutativity (and associativity, idempotency, etc.) and rewrite engines provide decision procedures to implement it, in the theory of $\mathbb{K}$ we assume that rewriting with computational rules takes place modulo the structural rules and that implementations of $\mathbb{K}$ provide heuristics or procedures to deal with structural rules. Such procedures are beyond the scope of this paper. We only mention that if one is willing to trade (some) non-determinism for performance, then one can break the circularity of heating/cooling rules like above by adding side-conditions saying that the heated expression is always a non-value (e.g., E_1 in the first rule) and that the cooled expression is always a value (e.g., E_1 in the second rule).

More generally, we can specify any evaluation context in terms of heating/cooling structural rules like above, by simply pulling the "hole" from the context and scheduling it in front of the context. Consider, for example, the following $\mathbb{K}$ evaluation context in the definition of SIMPLE stating that in order to calculate the lvalue of an array element we need to evaluate the index of that element (here A ranges over array expressions):

CONTEXT
`lvalue` $(A[\square])$

This context can be represented with the following two structural rules:

RULE

$$\frac{\texttt{lvalue(}\,A[E]\,\texttt{)}}{E \curvearrowright \texttt{lvalue(}\,A[\square]\,\texttt{)}}$$

[structural]

RULE

$$\frac{E \curvearrowright \texttt{lvalue(}\,A[\square]\,\texttt{)}}{\texttt{lvalue(}\,A[E]\,\texttt{)}}$$

[structural]

Note: In reduction semantics, the evaluation context above would be specified using a syntactic declaration of the form "$\mathit{Cxt} ::= \texttt{lvalue}(\texttt{A}(\mathit{Cxt}))$" Our current implementation of $\mathbb{K}$ allows users to specify evaluation contexts using a notation like the **lvalue** $(A[\square])$ above, in addition to specifying the more particular strictness attributes. Such declarations of evaluation contexts are then automatically translated into heating/cooling rules like above.

The heating/cooling rules are not restricted to only defining evaluation strategies by means of conventional evaluation contexts. For example, stimulated by practical needs, the current $\mathbb{K}$ tool allows users to also specify "contexts" like the following three:

CONTEXT

$I * \square$

when $I \neq_{\mathit{Int}} 0$

CONTEXT

$\square\ .\ M$

when $\square \neq_{K} \texttt{super}$

CONTEXT

$$\texttt{++}\ \frac{\square}{\texttt{lvalue(}\,\square\,\texttt{)}}$$

The first context above states that the second argument of a multiplication operation is evaluated only if the value of the first argument is different from zero (since one may want to give a shortcircuited semantics to multiplication and to reduce the amount of unnecessary non-determinism). The second states that the object in a member access expression is evaluated whenever it is different from `super` (since `super` member accesses are resolved statically, so they have a different semantics). The third context declaration above is more special and makes use of $\mathbb{K}$'s in-place rewriting notation which will be explained shortly in more detail. It basically says that when the argument of the increment construct is evaluated, it should be wrapped with the `lvalue` construct. The role of the wrapper is to allow a special treatment for the expression being evaluated. This is precisely what is needed in the case of l-values, as we want to "almost" evaluate the expressions (which could be variables, array elements, object fields), but stop once we compute the location. The heating rules corresponding to the three contexts above are, respectively:

RULE

$$\frac{I * E}{E \curvearrowright I * \square}$$

when $I \neq_{\mathit{Int}} 0$
[structural]

RULE

$$\frac{E\,.\,M}{E \curvearrowright \square\,.\,M}$$

when $E \neq_{K} \texttt{super}$
[structural]

RULE

$$\frac{\texttt{++}\ E}{\texttt{lvalue(}\ E\ \texttt{)} \curvearrowright \texttt{++}\ \square}$$

[structural]

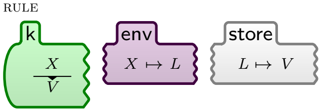
The corresponding cooling rules reverse the above rules. Note that, in particular, the cooling rule for the third context expects the wrapper to still be wrapping the expression, and it removes it upon plugging the expression back. This is to say that this wrapper should only have contextual meaning, used for altering the semantics, but being preserved by it. In some sense, these wrappers are actually providing locally typed evaluation contexts. One can devise other similar notations and it is likely that the $\mathbb{K}$ tool will incorporate new ones or different ones in the future. The point here is that the heating/cooling rules are quite powerful, giving $\mathbb{K}$ flexibility in defining complex evaluation strategies.

Not all structural rules need to be heating/cooling rules like above. Sometimes we want to desugar some language constructs into others or into built in 𝕂 constructs and we do not want such steps to be observable. For example, one may want to desugar a "for" loop into a while loop, or one may want to eliminate the sequential composition construct replacing it with the 𝕂 built-in $\curvearrowright$ construct. A language designer may not want such structural rules to be reversible. In fact, making all structural rules reversible may yield undesirable non-determinism in the language.

Computational rules

Many 𝕂 rules, particularly those which are computational, involve more than one cell. For example, the variable lookup rule of SIMPLE (presented below) grabs the program variable to lookup from the k cell, then grabs its location from the env cell, then accesses the value at that location in the store cell, and finally rewrites the program variable to that value. Thus, a significant amount of configuration structure needs be specified in the rule for variable lookup and in the end only a little bit of that structure gets changed, while the rest remains the same. Conventional rewrite rules of the form *left* ⇒ *right* have two drawbacks, one practical and one theoretical. On the practical side, one has to mention the entire configuration context in *both* the left- and the right-hand-side terms of the rules, which is tedious, error-prone, and non-modular. On the theoretical side, such rules enforce an interleaving semantics for concurrency where one may want a true concurrency semantics; for example, two threads reading the same location in the store would have to interleave, simply because the left-hand-sides of the corresponding variable lookup rule instances overlap. 𝕂 addresses these problems by introducing an *in-place* style for writing rewrite rules, which we discuss next, and a semantics for it which is not based on translation to conventional rules, which we briefly discuss in Section 2.5.

Here is the 𝕂 rule for variable lookup in SIMPLE:



Thus, the configuration context is mentioned only once, and the parts which change are underlined with the changes written underneath. A conventional rule *left* ⇒ *right* is a particular 𝕂 rule where the entire *left* term is underlined, with *right* underneath (like the structural rules shown above).

There are two additional 𝕂-specific aspects in the rule above that need to be discussed. First, note that the cells involved are either round or torn on their sides. A torn cell side means that it may contain more data there, but that data is irrelevant. For example, the variable X to be looked up is required to appear first in the computation cell k, but the remaining computation context is irrelevant. Similarly, the remaining bindings in the environment as well as the remaining locations in the store are irrelevant. We assumed a definition of maps as sets of pairs *key* ↦ *value*.

A second 𝕂-specific aspect to note in the rule above is that the configuration context does not match. Indeed, as the configuration of SIMPLE in Figure 1 shows, the cell store is not located within the same cell as k and env, so the three cells cannot be matched together as the lookup rule states. This rule takes advantage of 𝕂's *configuration abstraction* mechanism (previously called *context transforming* [43]), which allows us to only specify the needed cells in each rule, with the rest of the configuration context being inferred from the defined configuration (e.g., the one in Figure 1 for SIMPLE). The concretization of such abstract configuration rule contexts is based on several principles and criteria that help disambiguate among possible concrete rules. They are thoroughly discussed in [43]. Here we only mention that one of them, the *locality principle*, states that the configuration will be completed such that a minimum number of cells will be added. This ensures that, in a multithreaded SIMPLE program, the cells k and env in the rule above will not be assigned to different thread cells. Although that would be consistent with the configuration structure, having them in the same thread is "more local".

The motivation for configuration abstraction comes from the practical need for modular language semantics, more precisely from desired modularity under changes of configuration. As new features are added to a programming language, its configuration tends to changes its structure many times. In the case of SIMPLE, for example, it is natural to start with a non-concurrent fragment of it (in order to first focus on the sequential language constructs), where one does not include the threads and thread cells, their now inner cells being at the same top-level as the now shared cells. Then the lookup rule above matches as is, as its three cells are located at the same level in the cell structure. When we add threads, we realize that we need a more structured configuration, so we reorganize it as in Figure 1. In a conventional structural operational semantics (SOS) [37], changes of the configuration structure require revisiting all the existing rules, which is very inconvenient. Modular SOS [36] fixes this problem of SOS when the configuration consists of a flat (not nested) set of semantic cells, allowing rules to only grab from the configuration those cells that are needed. $\mathbb{K}$ pushes this idea further, allowing the same to also happen across cell boundaries. This way, we do not have to change the rule for variable lookup when threads are added to the SIMPLE language.

The overall principle underlying the abstraction capabilities above, as well as much of $\mathbb{K}$'s design in general, has always been the following:

> *Everything we write in a rule may work against us when the language is extended. The more the framework can automatically infer for us, the better.*

Informally, the cell and configuration abstractions above can be thought of as $\mathbb{K}$'s rewriting applying "modulo the configuration". Implementation-wise, this rule completion process can be applied statically or dynamically. In our current implementation based on translation to Maude [53], we apply them statically; e.g., the $\mathbb{K}$ rule above translates into a conventional rewrite rule:

$$\begin{array}{ll}\textsc{rule} & \mathsf{threads}(\mathsf{thread}(\mathsf{k}(X \curvearrowright K)\ \mathsf{env}(X \mapsto L,\ \mathit{Env})\ \mathit{Thread})\ \mathit{Threads}) \\ & \mathsf{store}(L \mapsto V,\ \mathit{Store}) \\ & \Rightarrow \\ & \mathsf{threads}(\mathsf{thread}(\mathsf{k}(V \curvearrowright K)\ \mathsf{env}(X \mapsto L,\ \mathit{Env})\ \mathit{Thread})\ \mathit{Threads}) \\ & \mathsf{store}(L \mapsto V,\ \mathit{Store})\end{array}$$

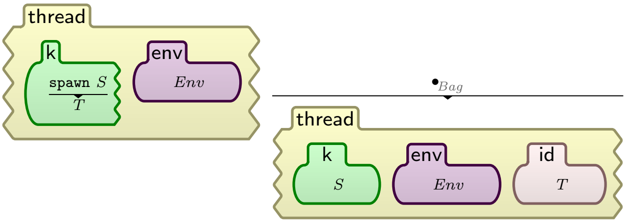
K, *Env*, *Thread*, *Threads*, and *Store* are cell frame variables corresponding to the tears. Our current $\mathbb{K}$ tool makes use of Maude's strengths, such as its multi-set and context-insensitive rewriting. Other backends may require more complex translations.

For example, one may need to complete the configuration context all the way to the top (if the target language does not support context-insensitive rewriting) or to add both left and right frames to cells holding maps, sets or multi-sets (when multi-set rewriting is not available).

In the remainder of this section we discuss two more 𝕂 rules, also part of the semantics of SIMPLE, which illustrate two more features of 𝕂's configuration abstraction. The former is motivated by the need to add new cell instances to the configuration dynamically, and the latter is motivated by the need to match items which reside in different instances of the same cell.

Let us first consider the rule which gives the semantics of thread spawning:

RULE

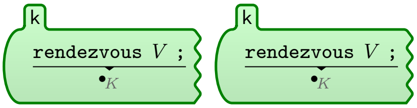


when fresh (T)

The spawning thread evaluates the spawn expression to the fresh (thread) identifier T while adding a child thread to the pool of threads; recall that "•" is the unit of any collection data-type in 𝕂 and it reads "nothing". The spawned thread is given its parent thread's environment, but nothing else is said about the other cells within the spawned thread. However, note that the spawned thread cell is torn. That tells 𝕂 to fill in the missing parts with the default cells and with their default contents, as defined in the configuration. This is another reason why the user is asked to provide default cell contents when defining the configuration (the other reason is to initialize the configuration).

The next rule gives the semantics of rendezvous synchronization, stating that two threads whose next statement is a same-value rendezvous synchronization request can discard their rendezvous statements and continue their execution:

RULE



[rendezvous]

How does 𝕂 know that the two k cells above are meant to appear in two different threads and not in the same thread? Again, the defined configuration in Figure 1 tells us how to disambiguate this rule: the thread cell is declared to have its multiplicity "*", which means that at any given moment during the execution of the semantics we may have zero, one or more thread cells within the threads cell. No other cells have multiplicity "*", so the only way for the rule above to match the configuration is for the two k cells to appear each inside a thread cell. 𝕂's configuration abstraction mechanism takes all these into account [43]. Moreover, as the semantics of 𝕂 is given through rewriting, the rendezvous rule needs to match two *distinct* cells, requiring at that at least two threads have reached a rendez-vous point before applying.

2.5 The Semantics of $\mathbb{K}$

The semantics of $\mathbb{K}$ is given in terms of transition systems. Any $\mathbb{K}$ definition can be regarded as a generator of transition systems, one for each program in the defined language. As expected, the states of these transition systems are given by program configurations and the transitions are given by instances of computational rules. The structural rules do not yield transitions, their role is to structurally rearrange the configuration so that computational rules match and apply. What is less obvious is that $\mathbb{K}$ allows (but does not enforce) more rule instances to apply concurrently on a given configuration as part of the same transition, even if they overlap; the only restriction is that a rule instance is not allowed to rewrite a subterm that another concurrent rule instance needs to access. This is reminiscent of how transactions work: concurrent reads are allowed, but no read/write or write/write conflicts.

For the time being, we leave $\mathbb{K}$'s configuration abstraction without a semantics. Instead, we prefer to think of it as how we are currently implementing it in our $\mathbb{K}$ tool, namely as syntactic sugar which is desugared statically. We do this for several reasons: first, configuration abstraction has no effect on the resulting transition systems, its role being to simply adapt the rules to fit the configuration as intended; second, it buys us and others time to better understand, evaluate and converge on what configuration abstraction should mean in its full generality; and finally and perhaps relatedly, configuration abstraction as it is now seems hard to formalize any other way than algorithmically. Thus, a rigorous formalization of configuration abstraction would currently give us little or no benefits, so it is not worth the effort.

$\mathbb{K}$ rules describe how terms can be transformed by altering some of their parts. $\mathbb{K}$ shares the idea of match-and-replace with standard term rewriting, but each $\mathbb{K}$ rule also specifies which part of the pattern is read-only. Let us next formally define the notion of a $\mathbb{K}$ *rule* and the desired concurrent $\mathbb{K}$ semantics.

Given a signature Σ and a (potentially infinite) set of variables X, let $T_\Sigma(X)$ denote the universe of Σ-terms with variables from X. Given $\mathcal{W} = \{\square_1, \ldots, \square_n\}$, named *context variables*, or *holes*, a $\mathcal{W}$*-context over* $\Sigma(X)$ (assume that $X \cap \mathcal{W} = \emptyset$) is a term $k \in T_\Sigma(X \cup \mathcal{W})$ in which each variable in $\mathcal{W}$ occurs once. The instantiation of a $\mathcal{W}$-context k with an n-tuple $\bar{t} = (t_1, \ldots, t_n)$, written $k[\bar{t}]$ or $k[t_1, \ldots, t_n]$, is the term $k[t_1/\square_1, \ldots, t_n/\square_n]$. One can regard $\bar{t}$ as a substitution $\bar{t} : \mathcal{W} \to T_\Sigma(X)$, defined by $\bar{t}(\square_i) = t_i$, in which case $k[\bar{t}] = \bar{t}(k)$. In what follows we fix a signature Σ and a set of variables X.

Definition 2.1 [43,54,52] A $\mathbb{K}$ *rule* $\rho : k[\underset{R}{\underline{\;L\;}}]$ is a triple where: k is a $\mathcal{W}$-context over $\Sigma(X)$, called the *rule pattern*, where $\mathcal{W}$ are the *holes* of k; k can be thought of as the "read-only" part or the "local" context of ρ; and $L, R : \mathcal{W} \to T_\Sigma(X)$ associate to each hole in $\mathcal{W}$ the *original term* and its *replacement term*, resp.; L, R can be thought of as the "read/write" part of ρ. When $\mathcal{W} = \{\square_1, \cdots, \square_n\}$ and $L(\square_i) = l_i$ and $R(\square_i) = r_i$, we may write

$$k[\underset{r_1}{\underline{\;l_1\;}}, \ldots, \underset{r_n}{\underline{\;l_n\;}}]$$

instead of $k[\underset{R}{\underline{\;L\;}}]$, since the holes are implicit and need not be mentioned.

The variables in $\mathcal{W}$ are only used to identify the positions in k where rewriting takes place; in practice we typically use the compact notation above, that is, underline the to-be-rewritten subterms in place and write their replacement underneath. Σ includes all the needed syntactic categories, that is, the language syntax, the configuration syntax, auxiliary operations, etc.

We can associate to any $\mathbb{K}$ rule $\rho : k[\underset{R}{\underline{\;L\;}}]$ a regular rewrite rule $K2R(\rho)$: $L(k) \to R(k)$. This translation is used, for example, in our current implementation of $\mathbb{K}$ (by translation to Maude; see Section 2.6). Although the potential for concurrency with sharing of resources is reduced by this translation (as concurrent applications of rules in rewriting logic are only allowed if the rules do not overlap), it is acceptable in many cases. Conversely, given a conventional rewrite rule τ : *left* $\to$ *right*, we can generate an obvious (zero-sharing) $\mathbb{K}$ rule $R2K(\tau) : \square[\underset{\mathit{right}}{\underline{\;\mathit{left}\;}}]$. For this reason, we sometimes take the liberty to write zero-sharing $\mathbb{K}$ rules using the conventional rewrite rule notation. If τ is a rewriting logic rule, then $t \xrightarrow{\tau} t'$ denotes the binary rewrite relation generated by τ, i.e: t rewrites to t' via an instance of τ. As usual, $\xrightarrow{\tau^*}$ is the reflexive and transitive closure of $\xrightarrow{\tau}$.

The concurrent $\mathbb{K}$ rewriting relation is more complex to define than the conventional concurrent term rewriting relation. That is because we want it to be *as concurrent as possible*, so that concurrent languages or calculi defined in $\mathbb{K}$ do not just have the standard concurrent semantics of rewriting logic, which forbids overlaps between concurrent redexes, but instead have greater concurrency by allowing overlaps between redexes, provided the overlaps only happen in their read-only portions. This means that two or more concurrent rewrites can simultaneously *share* some common portion of the state.

The key to achieving the above is to take into account the specifics of the $\mathbb{K}$ rules, namely the fact that they are explicit about which parts are shared and which parts are rewritten. Non-conflicting $\mathbb{K}$ rules are expected to possibly be applied concurrently, like transactions, where by "non-conflicting" rules we mean that neither of them rewrites portions of the term that are accessed (shared or written) by the other. We currently define $\mathbb{K}$'s concurrent rewrite relation in terms of *graph rewriting* (the double pushout approach), making crucial use of the notion of *parallel independence* [13]. We refer the interested reader to [54,52] for details. What is relevant here is that a $\mathbb{K}$ concurrent rewrite relation that captures the desired rules-as-transactions informal semantics above *can* be defined; we denote it $\Rrightarrow$ instead of $\to$.

If *cfg* is a SIMPLE configuration term (following the signature in Figure 1) holding two threads whose computations start with variables `x` and `y`, respectively, which therefore need to be looked up, then the two instances of the lookup rule (see Section 2.4) can be applied concurrently and the two occurrences of `x` and `y` get rewritten to their store values in *one step*; that is, if *cfg'* is the new configuration then $cfg \Rrightarrow cfg'$. Note that this is not possible when the $\mathbb{K}$ rule for lookup is regarded as a conventional rewrite rule (like in the *K2R* map): the two rule instances overlap on the store cell, so they need to interleave yielding either $cfg \Rightarrow cfg_x \Rightarrow cfg'$ or $cfg \Rightarrow cfg_y \Rightarrow cfg'$, where cfg_x is the configuration replacing only the `x` in the first thread with its value (and similarly for cfg_y). Moreover, $cfg \Rrightarrow cfg'$ (in one concurrent step) also when `x` and `y` are the same (shared) variable, as the two rule instances read but do not change the store location corresponding to `x` (a read/read access). On the other hand, if one thread reads and another writes (see the rule for variable assignment in Section 3) the same variable, then the two accesses are not allowed to proceed concurrently, they need to interleave. If the read and the written variables are distinct, then the two accesses can proceed concurrently.

$\mathbb{K}$'s rewriting has the following properties, where $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$ means that t can be rewritten in *one concurrent step* to t' using rules $\rho_1, \ldots, \rho_n$:

Theorem 2.2 *[54,52] Let ρ, $\rho_1, \ldots, \rho_n$ be not necessarily distinct $\mathbb{K}$ rules.*

Completeness: *If $t \xrightarrow{K2R(\rho)} t'$ then $t \overset{\rho}{\Rrightarrow} t'$.*

Soundness: *If $t \overset{\rho}{\Rrightarrow} t'$ then $t \xrightarrow{K2R(\rho)^*} t'$.*

Serializability: *If $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$, then there exists a sequence of terms $t_0, \cdots, t_n$, such that $t_0 = t$, $t_n = t'$, and $t_{i-1} \overset{\rho_i^*}{\Rrightarrow} t_i$.*

Completeness says that any steps made using rewriting logic can also be made using $\mathbb{K}$ rewriting. Soundness states that any non-concurrent step made using $\mathbb{K}$ rewriting corresponds to zero, one or more rewriting logic steps; this is due to the fact that the term to be rewritten is represented as a graph in $\mathbb{K}$, and zero, one or more term-rewrite steps are needed to mimic a graph rewrite step (zero when the rewritten part is unreachable). Serializability says that the concurrent rewrite relation $\Rrightarrow$ does not reach any other terms than the rewrite relation $\rightarrow$: it just reaches them in a possibly smaller number of steps.

From a practical viewpoint, the theorem above tells us that it may be acceptable, in many situations, to translate $\mathbb{K}$ rules into conventional rewrite rules using the *K2R* map. The only thing lost in translation is the amount of true concurrency available in the original $\mathbb{K}$ definition. Note, however, that most semantic frameworks for programming languages follow an interleaving philosophy by their nature, so "losing some true concurrency" cannot even be formulated in those frameworks. Nevertheless, we believe that with the advance of massively parallel architectures, maximizing the true concurrency capability of a semantic framework will be increasingly desirable, so $\mathbb{K}$ makes no compromises w.r.t. its theoretical support for concurrency. That being said, the reader who thinks that $\mathbb{K}$'s concurrent rewrite relation $\Rrightarrow$ is hard to

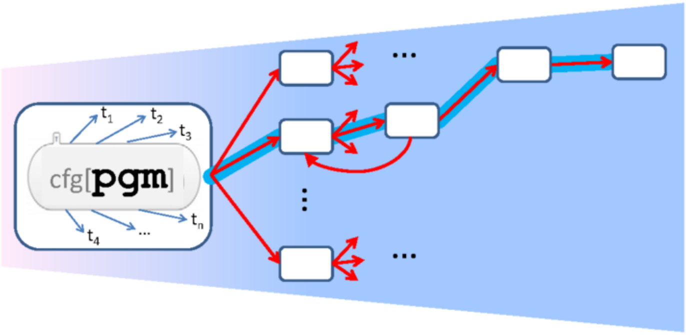


Fig. 2. The state space associated to a program.

realize, or who does not want to get into the technicalities of graph rewriting, or who simply does not believe in true concurrency, is free to replace it in the rest of this section with the (still truly concurrent but not structure-sharing) rewriting logic relation → associated to it via *K2R*. The remainder of this section is parametric in the relation $\Rrightarrow$.

Definition 2.3 A $\mathbb{K}$ *(rewrite) system* (or $\mathbb{K}$ *theory* or $\mathbb{K}$ *definition*) is a triple $\mathcal{K} = (\Sigma, \mathcal{S}, \mathcal{C})$, where Σ is its *signature* and $\mathcal{S}$ and $\mathcal{C}$ are sets of *structural* and *computational* $\mathbb{K}$ rules, respectively. Let $\Rrightarrow_{\mathcal{S}}$ and $\Rrightarrow_{\mathcal{C}}$ be the corresponding concurrent rewrite relations, and let $\Rrightarrow_{\mathcal{K}}$ be the relation $\Rrightarrow^*_{\mathcal{S}} \circ \Rrightarrow_{\mathcal{C}} \circ \Rrightarrow^*_{\mathcal{S}}$.

Note that $\Rrightarrow_{\mathcal{S}}$ is not necessarily symmetric. Moreover, note that $t \Rrightarrow^*_{\mathcal{S}} u$ and $t \Rrightarrow_{\mathcal{K}} t'$ and $u \Rrightarrow_{\mathcal{K}} u'$ do not necessarily imply $t' \Rrightarrow^*_{\mathcal{S}} u'$ (i.e., we do not enforce coherence [57]). To see that this makes practical sense, consider a hypothetical programming language which already provides a statement `halt` for abrupt termination whose semantics is given with a computational rule (dissolving the entire contents of the `k` cell) and suppose that we want to add a non-deterministic halting statement, say `ndhalt`. One way to do it is to add a structural rule rewriting `ndhalt` to `halt` and a computational rule dissolving the `ndhalt` statement (as if it was the empty statement). Then take t to be some configuration *cfg*[`ndhalt;rest`], u to be *cfg*[`halt;rest`], t' to be *cfg*[`rest`], and u' to be *cfg*[•] (i.e., *cfg* with an emptied computation cell). Similarly, $t \Rrightarrow^*_{\mathcal{S}} u$ and $t \Rrightarrow_{\mathcal{K}} t'$ and $t' \Rrightarrow^*_{\mathcal{S}} u'$ do not necessarily imply $u \Rrightarrow_{\mathcal{K}} u'$. For example, take the same t, u and t' as above, but $u' = t'$.

The rewrite relation $\Rrightarrow_{\mathcal{K}}$ associated to a $\mathbb{K}$ rewrite system $\mathcal{K} = (\Sigma, \mathcal{S}, \mathcal{C})$ gives us an obvious transition system on T_{Σ}, which can be regarded as the semantics of $\mathcal{K}$. Alternatively, one can use the Kripke structure, or the graph representation of the transition system, as the semantics. If $\mathcal{K}$ is a language definition whose initial configuration pattern has the form *cfg*[*$PGM*], e.g., the one in Figure 1, then this transition system gives us for any program `pgm` a transition subsystem formed with all the configurations reachable from *cfg*[`pgm`] with the relation $\Rrightarrow_{\mathcal{K}}$. This transition system, or its Kripke or graph representation, can be regarded as the semantics of `pgm`. Figure 2 illustrates the $\mathbb{K}$ semantics of `pgm`. Each box represents a reachable configuration term. The thin arrows inside the box represent applications of structural rules, or structural rearrangements of the configuration, and the thick arrows between boxes represent actual computational steps. Here we are not concerned with minimizing transition systems (by collapsing equivalent configurations).

To conclude, the semantics of $\mathbb{K}$ is given in terms of transition systems, based on a concurrent rewrite relation that takes the specific nature (e.g., explicit sharing) of the $\mathbb{K}$ rules into account. If one forgets the specific nature of the $\mathbb{K}$ rules then one still gets a valid semantics, amenable for execution on existing rewrite engines like Maude, but one which looses some of the true concurrency of the original $\mathbb{K}$ definition. $\mathbb{K}$ tools can implement different techniques and algorithms that work with $\mathbb{K}$ definitions. For example, thanks to excellent support from the underlying Maude system, our current implementation provides support for execution (highlighted as a bold path in Figure 2), for state-space search, and for explicit-state LTL model-checking.

2.6 The $\mathbb{K}$ Tool

This section only describes the current implementation choices made by the $\mathbb{K}$ tool to provide a meaningful, yet not prohibitively expensive (in terms of time and resources) implementation of the theoretical ideas explained above. We refer the reader interested in details on using the $\mathbb{K}$ tool to [53].

Currently, the $\mathbb{K}$ tool translates $\mathbb{K}$ specifications into rewrite theories to be executed, explored, and analyzed using the Maude [12] rewrite engine. Therefore, let us first briefly give some context about the differences between $\mathbb{K}$ and rewriting logic, and the implementation of rewriting logic in Maude, and then present the challenges posed to our implementation by these differences.

Rewriting Logic and Maude

Similarly to $\mathbb{K}$, rewriting logic [32] also exhibits two categories of sentences. Equations are akin to $\mathbb{K}$ structural rules: they specify deterministic behavior, defining classes of states on which transitions occur. However, unlike structural rules, equations are thought of as always applying both ways, and thus defining an equivalence class of states. Rewrite rules, similar to $\mathbb{K}$ computational rules, specify transitions between classes of states.

To guarantee that the semantics obtained through rewriting corresponds to the initial model semantics for a rewrite theory, Maude requires rewrite theories to satisfy certain properties. First it assumes that the equations, when oriented from left to right are (ground) confluent and terminating; this ensures that for each equivalence class of states one can obtain a unique canonical form, thus enabling one to easily check equality between states by rewriting to normal form. Moreover, rewrite rules are assumed coherent [57] with respect to the equations, this allowing Maude to always reduce a state to the equational normal form before applying a rule without losing any possible behaviors.

Restricting concurrency

One theoretically significant implementation choice, more or less dictated by our implementation target, was to restrict the potential for concurrency by using the straight-forward *K2R* translation of $\mathbb{K}$ rules into rewrite rules described in Section 2.5, which disregards the potential for sharing provided by the $\mathbb{K}$ rules. We see this as a reasonable restriction, with the immediate benefit of being able to use the state-of-art state space exploration and model checking capabilities offered by Maude without additional development effort.

Restricting heating/cooling

As the heating and cooling rules used for defining evaluation strategies in Section 2.4 are structural rules, and, moreover, bidirectional, the above comparison with rewriting logic would suggest that they should be represented directly as equations. However, this would be problematic in the context of the Maude implementation because, while Maude (assuming confluence) simply orients equations from left to right, both heating (left-to-right) and cooling (right-to-left) rules need to be applied during an execution.

To address that, the default behavior of the $\mathbb{K}$ tool is to generate two equations, one for heating and the other for cooling; however, to ensure termination the type of the computations which are allowed to be heated/cooled by these rules need to be restricted. To achieve that, the $\mathbb{K}$ tool requires the user to define values, and only applies heating rules to schedule for evaluation computations which are not already values. Also, by default, it only applies cooling rules when the computation to be cooled is a value. This behavior is very useful when interpreting programs, as having a redex always at the top of the computation is usually enough; however, it can miss behaviors in the case that the evaluation order is non-deterministic and expressions have side-effects. Some ways to address that are presented in the paragraphs below.

Additionally, for efficiency reasons, the $\mathbb{K}$ tool only applies heating/cooling rules at the top of a computation cell. We consider this restriction reasonable, as most of the redexes must rise to the top of the computation cell to reduce.

Restricting the transition system

Again, from the above comparison with rewriting logic, it would seem natural to encode computational rules as rewrite rules in Maude, to capture them in the transition system. Nevertheless, our experience of working with the $\mathbb{K}$ tool shows that in the presence of concurrency, the transition systems are too large to be explored and analyzed. Moreover, most of the computational transitions play no role in testing/verifying important concurrency properties. Therefore, the default behavior of the $\mathbb{K}$ tool is to require the user to annotate the rules which should generate transitions in the Maude transition system, and represent all other rules through equations, assuming they are deterministic.

Modeling all behaviors?

With the above restrictions in place, a reasonable question to be asked is whether the transition system generated by the $\mathbb{K}$ tool for a $\mathbb{K}$ definition and a term captures the $\mathbb{K}$ transition system associated to that definition and term. In particular, is the $\mathbb{K}$ tool transition system sound and complete with respect to checking properties of the $\mathbb{K}$ transition system?

We can answer this positively for the soundness half of this question with respect to reachability properties. As most of the structural rules are encoded as equations without being changed, and as the $\mathbb{K}$ tool heating/cooling rules are only restricted versions of the corresponding rules in the definition, the transition system generated by the $\mathbb{K}$ tool is a collapsed version of the $\mathbb{K}$ transition system associated to the definition. In this transition system multiple states may be collapsed through what in rewriting logic is called equational abstractions [33], obtained by encoding some of the computational rules as equations. Moreover, some additional transitions observable in the $\mathbb{K}$ transition system might be inhibited here by the rule not being applicable on the particular normal form obtained by orienting the equations.

Capturing all the transitions of the $\mathbb{K}$ transition system using Maude would be possible (modulo interleaving some $\mathbb{K}$ concurrent transitions) by encoding all rules (both structural and computational) as rewrite rules. However, although providing completeness, this approach would have two drawbacks. First, to obtain the intended transition system from the one generated by Maude, all transitions obtained by applying structural rules will have to be regarded as internal transitions, hence the states they relate would have to be collapsed into a single state. Second, as mentioned above, even without the structural rules, the transition system quickly grows unfeasible to explore; with the addition of the structural rules (some of which are inverses of each other), this state space would become too large even for small and deterministic programs.

Support for non-deterministic evaluation order

When defining languages with non-deterministic evaluation orders for certain operators, encoding heating/cooling rules as equations leads to non-confluent specifications (as the rules for heating different arguments compete). This leads to certain transitions being potentially missed, which becomes even more problematic when side effects are permitted in expressions, as this leads to observable behaviors being missed. For example, assuming that "‖" is an operation with non-deterministic order of evaluation, and `print` is a function for printing a value to the standard output, the observable behavior of the program `print "a"` ‖ `print "b"` would be to display either "ab" or "ba". However, with the mechanisms described above, the $\mathbb{K}$ tool could only capture one of these behaviors in the transition system. To alleviate that, the $\mathbb{K}$ tool allows certain heating rules to "superheat" the computation, forcing all superheat rules to be considered when building the transition system.

However, this only partially solves the problem. Consider the program `print "a"` ‖ (`print "b"` ‖ `print "c"`), whose observable behavior is to display any permutation of "a", "b", and "c". Assuming that the heating rules for "‖" are superheat rules, the $\mathbb{K}$ tool would only generate as observable behaviors the permutations "abc", "acb", "bca", and "cba", missing "bac" and "cab". The reason is that the restriction of the cooling rules to only apply when the computation to be cooled is already a value, meaning that once the evaluation of the (`print "b"` ‖ `print "c"`) subexpression has started, it must be evaluated until completion. To alleviate that, the $\mathbb{K}$ tool allows the user to annotate certain rules (typically those exhibiting side effects) as "supercool", which will force cooling rules to apply without the restriction that the computation is a value, and thus (in combination with the superheat rules) to allow the choice of another redex.

Typesetting the definition in LaTeX

The rules and configurations displayed throughout this paper were generated from LaTeX code produced by the $\mathbb{K}$ tool from their ASCII representations. Following the literate programming paradigm, the $\mathbb{K}$ tool allows users to annotate definitions with comments. The definitions in Sections 3 and 4 are only slight adaptations of the LaTeX code obtained through the $\mathbb{K}$ tool.

The $\mathbb{K}$ LaTeX style also provides a mathematical mode, which may be preferred in formal writing. For example, here is how the rule for spawning a thread presented above is typeset using the mathematical notation:

RULE

$$\left\langle \cdots \left\langle \frac{\texttt{spawn}\ S}{T} \cdots \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}} \cdots \right\rangle_{\mathsf{thread}} \quad \frac{\bullet Bag}{\left\langle \cdots \left\langle S \right\rangle_{\mathsf{k}} \left\langle Env \right\rangle_{\mathsf{env}} \left\langle T \right\rangle_{\mathsf{id}} \cdots \right\rangle_{\mathsf{thread}}}$$

when **fresh** (T)

Note that the contents of a cell are wrapped using angle brackets and that the label of the cell is added as a subscript to the right side angle bracket. Moreover, “torn” cells, i.e., the fact that some contents of the cell were omitted, is represented here using the ellipses symbol.

3 $\mathbb{K}$ Formal Semantics of Untyped SIMPLE

This section presents the full semantic definition of the untyped SIMPLE language (introduced in Section 2.1) in $\mathbb{K}$.

3.1 Syntax

We start by defining the SIMPLE syntax. The SIMPLE language constructs 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

Identifiers are built in 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 for convenience. 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. 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;`”, so 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 ::= var Exps ;
                | 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 𝕂 is what we call "the syntax of the semantics": while powerful enough to define non-trivial syntax (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 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 an argument and applies to an expression (which can in particular be another array lookup), respectively. The construct sizeOf gives the size of an array defined as the number of elements of its first dimension. Note that almost all constructs are strict in all their arguments. The only constructs which are not fully 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 binary Boolean constructs which are short-circuited (and thus are strict only in the first argument), the thread spawning construct which creates a new thread executing the argument block and returns 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). The bracket specifies that the corresponding syntactic production is only used for grouping purposes (like a bracket) and should not be included in the abstract syntax tree.

```
SYNTAX   Exp ::= Int | Bool | String | Id
               | (Exp) [bracket]
               | ++ Exp
               | Exp[Exps] [strict]
               | Exp(Exps) [strict]
               | - Exp [strict]
               | sizeOf (Exp) [strict]
               | read ()
               | Exp * Exp [strict]
               | Exp / Exp [strict] | Exp % Exp [strict]
               | Exp + Exp [strict] | Exp - Exp [strict]
               | Exp < Exp [strict] | Exp <= Exp [strict]
               | Exp > Exp [strict] | Exp >= Exp [strict]
               | Exp == Exp [strict] | Exp != Exp [strict]
               | ! Exp [strict]
               | Exp && Exp [strict(1)] | Exp || Exp [strict(1)]
               | spawn Block
               | Exp = Exp [strict(2)]
```

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).

$$\textsc{syntax}\quad \mathit{Ids} ::= \mathit{List}\{\mathit{Id}, \text{“,”}\}$$

$$\textsc{syntax}\quad \mathit{Exps} ::= \mathit{List}\{\mathit{Exp}, \text{“,”}\}\ [\text{strict}]$$

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 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 avoid attribute of if-then-else informs the parser to avoid using this production when parsing ambiguities involving it arise, which in this case means whenever the simpler if-then can be used. The `print` statement construct is variadic, that is, it takes an arbitrary number of arguments.

$$\textsc{syntax}\quad \mathit{Block} ::= \{\} \mid \{\mathit{Stmts}\}$$

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

$$\textsc{syntax}\quad \mathit{Stmts} ::= \mathit{Stmt} \mid \mathit{Stmts}\ \ \mathit{Stmts}$$

3.2 Desugared Syntax

Like in many other languages, some of SIMPLE’s constructs can be desugared into a smaller set of basic constructs. We only want to give semantics to core constructs, so we first eliminate the derived ones before we start the semantics. All desugaring macros below are straightforward.

Note that all the rules in this subsection are tagged with the `macro` attribute. That signals that they are to be regarded as AST manipulation rules preprocessing the program before being executed using the other rules provided by the definition.

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

RULE
$$\frac{\texttt{for }(\mathit{Start}\ \ \mathit{Cond}\ \texttt{;}\ \mathit{Step})\{S\}}{\{\mathit{Start}\ \texttt{while }(\mathit{Cond})\{S\ \ \mathit{Step}\ \texttt{;}\}\}}$$
[macro]

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

RULE
$$\frac{\texttt{var }X\texttt{ = }E\texttt{ ;}}{\texttt{var }X\texttt{ ; }X\texttt{ = }E\texttt{ ;}}$$
[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. Our semantics for desugaring initialized declarations "`var` X `=` E `;`" is inspired from C, where any occurrence of X in E refers to the X in the same declaration and not to an X in an outer scope shadowed by the current declaration. Other dynamic semantics are also possible, as well as static semantics that reject uses of variables in their own declarations.

3.3 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 *configuration* and *values*. As the configuration of SIMPLE, depicted in Figure 1, was thoroughly explained in Section 2.3, we only discuss values in this section.

Values

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. We here define the values 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). The last line indicates that values are $\mathbb{K}$ results.

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

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

SYNTAX $\mathit{Exp} ::= \mathit{Val}$

SYNTAX $\mathit{KResult} ::= \mathit{Val}$

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, one would not need to include values in expressions.

3.4 Declarations and Initialization

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

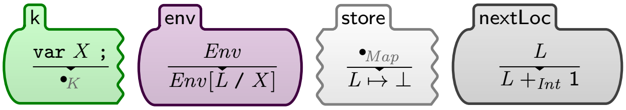
Variable Declaration

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 "$\bullet$", 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 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 K, the store is not frozen while $L \mapsto \bot$ is added to it, while the environment is frozen during the update operation $\mathit{Env}[L/X]$.

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 transactional step, with the rule below. Note also how configuration abstraction allows us to only mention the needed cells; indeed, 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

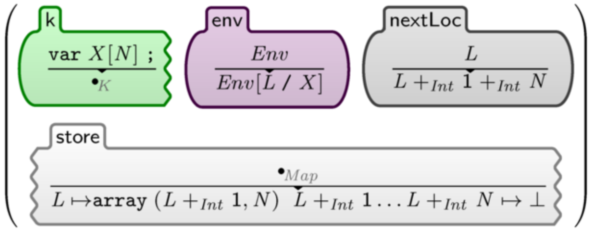


Array Declaration

The $\mathbb{K}$ semantics of the uni-dimensional array declaration is similar to the above declaration of ordinary variables. First, we use a context declaration, which requests the evaluation of the array dimension. Once evaluated, say to a natural number N, $N +_{\mathit{Int}} 1$ locations are allocated in the store for an array of size N, the additional location (chosen to be the first one) 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 definition in the auxiliary operation section, initializes each location in the list $L \ldots L'$ to V. Note that, since the array dimensions can be arbitrary expressions, we can dynamically allocate memory in SIMPLE by means of array declarations.

CONTEXT
`var` $X[\square]$ `;`

RULE



when $N \geq_{Int} 0$

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 declarations 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 ::= `$1` | `$2`

RULE

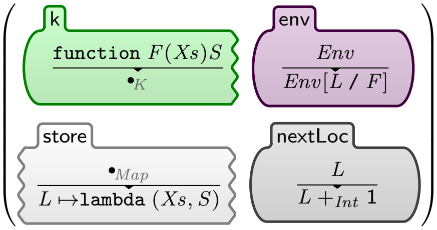
```
          var X[N1, N2, Vs] ;
var X[N1] ;
{
  var $1 = X ;
  for (var $2 = 0 ;  $2 <= N1 - 1 ; ++ $2){
    var X[N2, Vs] ;  $1[$2] = X ;
  }
}
```

[structural]

Ideally, one would like to perform syntactic desugarings like the one above before the actual semantics. Unfortunately, that is 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 Section 4), discards programs which do not respect this requirement.

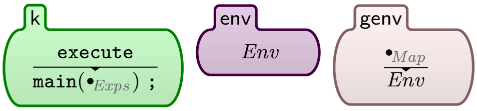
RULE



When we are done with the first pass (pre-processing), the computation cell `k` contains only the token `execute` (the computation item `execute` was placed right after the program in the `k` cell of the initial configuration in Figure 1) 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 $K ::=$ `execute`

RULE



[structural]

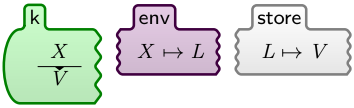
3.5 Expressions

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 to V:

RULE



[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.

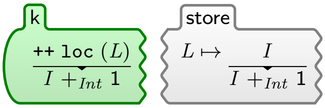
Variable/Array increment

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 use the special kind of context specified in Section 2.4, stating that the expression to increment should be wrapped by the auxiliary `lvalue` construct when evaluated. The semantics of expressions wrapped by `lvalue` is defined at the end of this definition (Section 3.7). For now, all we need to know is that, under the `lvalue` wrapper, an expression and evaluates to a value representing its location. Location values, also defined in Section 3.7, are integers wrapped with the construct `loc`, to distinguish them from ordinary integers.

CONTEXT

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

RULE



[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
$$\frac{I1 \texttt{ + } I2}{I1 +_{Int} I2}$$

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

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

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

RULE
$$\frac{I1 \texttt{ / } I2}{I1 \div_{Int} I2}$$
when $I2 \neq_{I} nt\ 0$

RULE
$$\frac{I1 \texttt{ \% } I2}{I1 \%_{Int} I2}$$
when $I2 \neq_{I} nt\ 0$

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

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

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

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

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

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

RULE $\dfrac{\mathit{V1} \;\texttt{==}\; \mathit{V2}}{\mathit{V1} =_{K} \mathit{V2}}$

RULE $\dfrac{\mathit{V1} \;\texttt{!=}\; \mathit{V2}}{\mathit{V1} \neq_{K} \mathit{V2}}$

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

RULE $\dfrac{\texttt{!}\; T}{\neg_{\mathit{Bool}}\, T}$

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

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

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

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

Array lookup

Untyped SIMPLE does not check array 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 towards the end of the definition.

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

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

The anywhere attribute attached to the two rules above instructs the $\mathbb{K}$ tool that these rules should be applied in any context, not only at the top of the computation cell as all other rules; this is needed for giving semantics to lvalues.

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

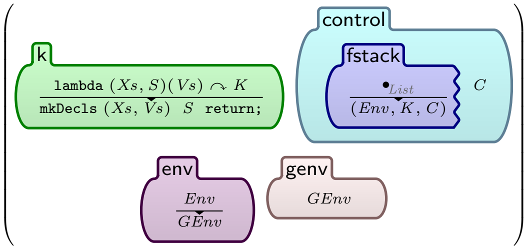
Function call

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 in 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 definition).

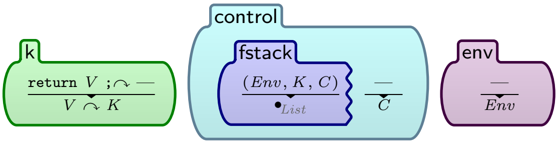
The second rule pops the computation, the environment and the control data from the function stack when a **return** statement is encountered as the next computational task, passing the returned value to the popped computation (the popped computation was the context in which the returning function was 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 nullary **return;** statements.

SYNTAX $\mathit{ListItem} ::= (\mathit{Map}, K, \mathit{Bag})$

RULE



RULE



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

RULE

$$\frac{\texttt{return;}}{\texttt{return nothing ;}}$$

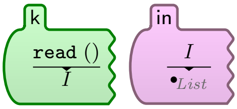
[macro]

Like for division-by-zero, it is left unspecified what happens when the **nothing** value is used in domain calculations. For example, from the 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 (like the type checker in Section 4) and reject programs that do. Unlike the undefined symbol $\perp$ which had the sort K instead of *Val*, we defined **nothing** to be a value. That is because, as explained, 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, such as what happens when we call a function for its side effects (e.g., with a statement "`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



[read]

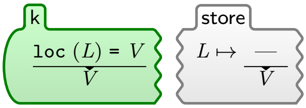
Assignment

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

$$\underset{\texttt{lvalue}\ (\square)}{\underline{\square}}\ \texttt{=}\ \_$$

RULE



[assignment]

3.6 Statements

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

Blocks

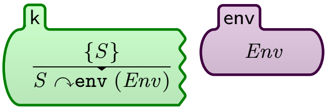
Empty blocks are simply discarded, as shown in the first rule below. For non-empty blocks, we schedule the enclosed statement but we have to make sure the environment is recovered after the enclosed statement executes. Recall that we allow local variable declarations, whose scope is the block enclosing 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

$$\frac{\{\}}{\bullet_K}$$

[structural]

RULE



[structural]

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

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 built in 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 treating an expression as a statement). 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 one 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

$$\frac{S1\ \ S2}{S1 \curvearrowright S2}$$

[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

$$\frac{V\ ;}{\bullet_K}$$

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

$$\frac{\texttt{if}\ (\mathsf{true})S\ \texttt{else}\ —}{S}$$

RULE

$$\frac{\texttt{if}\ (\mathsf{false})—\ \texttt{else}\ S}{S}$$

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

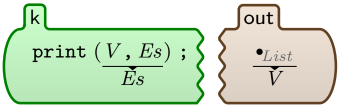
$$\frac{\texttt{while}\ (E)S}{\texttt{if}\ (E)\{S\ \ \texttt{while}\ (E)S\}}$$

[structural]

Print

The `print` statement was strict, so all its arguments are now evaluated (`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



[print]

RULE

$$\frac{\texttt{print}\ (\bullet_{Vals})\ ;}{\bullet_K}$$

[structural]

Exceptions

SIMPLE allows parametric exceptions, i.e., one can throw and catch a particular value. The statement "`try` S_1 `catch(`X`)` S_2" proceeds with the evaluation of S_1.

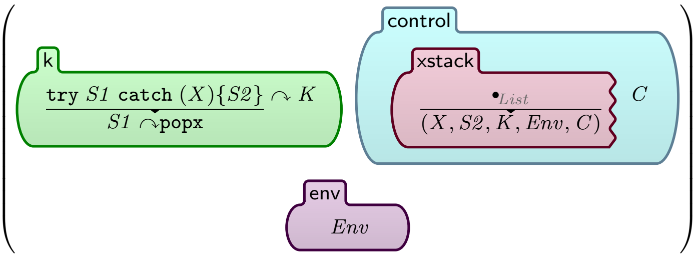
If S_1 evaluates normally, i.e., without any exception thrown, then S_2 is discarded and the execution continues normally. If S_1 throws an exception with a statement of the form "`throw` E", then E is first evaluated to some value V (`throw` was declared to be strict), then V is bound to X, then S_2 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.

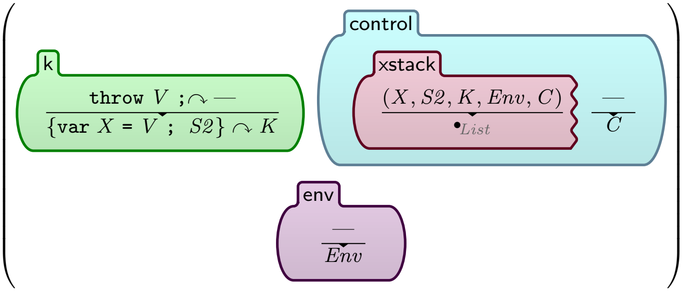
SYNTAX $\mathit{ListItem} ::= (\mathit{Id}, \mathit{Stmt}, K, \mathit{Map}, \mathit{Bag})$

SYNTAX $K ::=$ `popx`

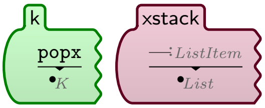
RULE



RULE



RULE



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.

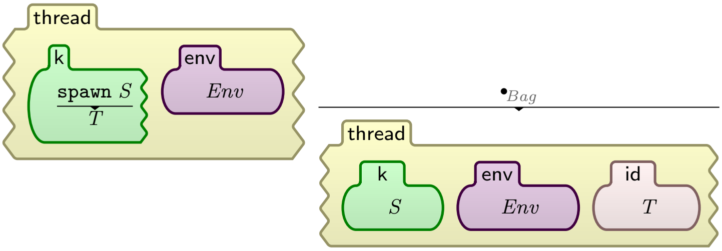
Threads

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 (Figure 1). This is part of $\mathbb{K}$'s configuration abstraction mechanism.

RULE

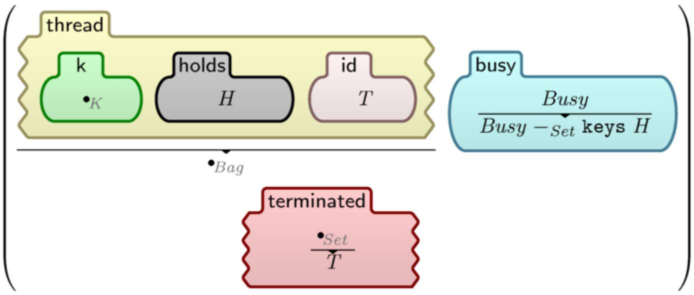


when fresh (T)

Thread termination

Dually to the above, when a thread terminates its assigned computation (the contents of its k cell is empty), 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 (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, for the join construct.

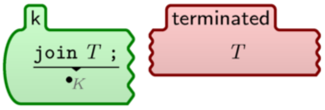
RULE



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



Acquire lock

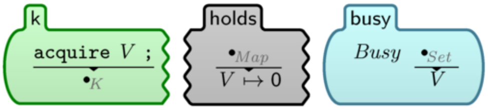
There are two cases to distinguish when a thread attempts to acquire a lock (in SIMPLE any value can be used as a lock):

(i) 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).

(ii) The thread already has the lock, in which case it just increments its counter for the lock (the locks are re-entrant).

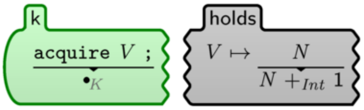
These two cases are captured by the two rules below:

RULE



when $\neg_{Bool}$ V `in` *Busy*
[acquire]

RULE



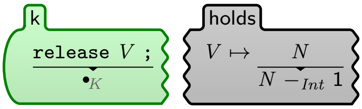
Release lock

Similarly, there are two corresponding cases to distinguish when a thread releases a lock:

(i) The thread holds the lock more than once, in which case all it needs to do is to decrement the lock counter.

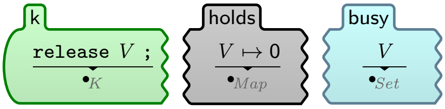
(ii) The thread holds the lock only once, in which case it needs to remove it from its holds cell and also from the shared busy cell, so other threads can acquire it if they need to.

RULE



when $N >_{Int} 0$

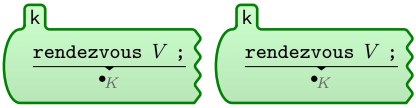
RULE



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



[rendezvous]

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

RULE

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

RULE

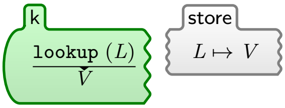
$$\frac{\texttt{mkDecls}\ (\bullet_{Ids}, \bullet_{Vals})}{\{\}}$$

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 we can specify that both rules should be considered transitions when exploring the state space by mentioning that the `lookup` tag defines transitions.

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

RULE



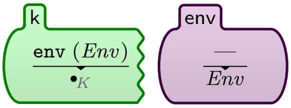
[lookup]

Environment recovery

The role of the following rule is to discard the current environment in the `env` cell and replace it with the environment that it holds. This rule is structural: we do not want it to count as a computational step in the transition system of a program.

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

RULE



[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

$$\frac{\texttt{env}\ (\text{—})}{\bullet_K} \curvearrowright \texttt{env}\ (\text{—})$$

[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 acts like a constraining context for the expression E, forcing it to evaluate to its lvalue. More precisely, although `lvalue` does not itself evaluate, it is used to constrain the evaluation of the expression it wraps. The rules below specify semantics only when E is an l-value, that is, when

E is either a variable or evaluates to an array element. When that happens, E evaluates in this l-value context 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 expression E in an `lvalue` context, 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.

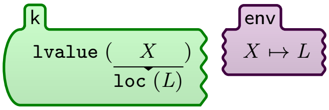
SYNTAX $\mathit{Exp} ::= \texttt{lvalue}\ (K)$

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

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

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

RULE



[structural]

RULE
$\texttt{lvalue}\ (\underset{\texttt{loc}\ (L)}{\underline{\texttt{lookup}\ (L)}})$

[structural]

Recall that, as mentioned in Section 2.4, the `lvalue` construct serves as a locally typed evaluation context. Therefore, the rules above preserve the lvalue context when evaluating expressions to their corresponding location values; the construct can only be added/removed by the heating/cooling rules which introduce it. For example, for the assignment evaluation context, the generated heating/cooling rules are:

RULE
$$\frac{E_1 = E_2}{\texttt{lvalue}\ (E_1) \curvearrowright \square = E_2}$$
[structural]

RULE
$$\frac{\texttt{lvalue}\ (E_1) \curvearrowright \square = E_2}{E_1 = E_2}$$
[structural]

Initializing multiple locations

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

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

RULE
$$\frac{N \ldots M \mapsto \text{—}}{\cdot_{\mathit{Map}}}$$
when $N >_{\mathit{Int}} M$

RULE
$$\frac{N \ldots M \mapsto K}{N \mapsto K \quad N +_{\mathit{Int}} 1 \ldots M \mapsto K}$$
when $N \leq_{\mathit{Int}} M$

The semantics of SIMPLE is now complete.

4 $\mathbb{K}$ Type System of SIMPLE

Here we discuss the $\mathbb{K}$ static semantics of the SIMPLE language, or in other words, a type system for it in $\mathbb{K}$. Following the imperative paradigm, we assume that all variables and functions explicitly declare their types. This is done by a slight modification of the syntax of SIMPLE; we call the resulting language "typed SIMPLE". We here only focus on the new and interesting problems raised by the addition of type declarations, and what it takes to devise a type system/checker for the language.

When designing a type system for a language, no matter in what paradigm, we have to decide upon the intended typing policy. Note that we can have multiple type systems for the same language, one for each typing policy. For example, should we accept programs which don't have a main function? Or should we allow functions that do not return explicitly? Or should we allow functions whose type expects them to return a value (say an `int`) to use a plain "`return;`" statement, which returns no value, like in C? And so on.

Typically, there are two opposite tensions when designing a type system. On the one hand, you want your type system to be as permissive as possible, that is, to accept as many programs that do not get stuck when executed with the untyped semantics as possible; this will keep the programmers using your language happy. On the other hand, you want your type system to have a reasonable performance when implemented; this will keep both the programmers and the implementers of your language happy. For example, a type system for rejecting programs that could perform division-by-zero is not expected to be feasible in general. A simple guideline when designing typing policies is to imagine how the semantics of the untyped language may get stuck and try to prevent those situations from happening.

Before we give the $\mathbb{K}$ type system of SIMPLE formally, we discuss, informally, the intended typing policy:

- Each program should contain a `main()` function. Indeed, the untyped SIMPLE semantics gets stuck on programs without a `main` function.
- Each primitive value has its own type, i.e., `int`, `bool`, or `string`. There is also a type `void` for nonexistent values, e.g., for the result of a function meant to return no value (but only be used for its side effects, like a procedure).
- The syntax of untyped SIMPLE is extended to allow type declarations for all the variables, including array variables. This is done in a C/Java-style. For example, "`int x;`" or "`int x=7, y=x+3;`", or "`int[][] a[10,20];`" (the latter defines a 10×20 matrix of arrays of integers). Recall from untyped SIMPLE that, unlike in C/Java, our multi-dimensional arrays use comma-separated arguments, although they have the array-of-array semantics.
- Functions are also typed in a C/Java style. However, since in SIMPLE we allow functions to be passed to and returned by other functions, we also need function types. We will use the conventional higher-order arrow-notation for function types, but will separate the argument types with commas. For example, a function `f` returning an array of `bool` elements and taking as argument an array `x` of

two-integer-argument functions returning an integer is declared using a syntax of the form

```
bool[] f(((int,int)->int)[] x) { ... }
```

and has the type `((int,int)->int)[] -> bool[]`.

- We allow any variable declarations at the top level. Functions can only be declared at the top level. Each function can only access the other functions and variables declared at the top level, or its own locally declared variables. SIMPLE has static scoping.
- The various expression and statement constructs take only elements of the expected types.
- Increment and assignment can operate both on variables and on array elements. For example, if `f` has type `int->int[][]` and function `g` has the type `int->int`, then the increment expression `++f(7)[g(2),g(3)]` is valid.
- Functions should only return values of their declared type. To give the programmers more flexibility, we allow functions to use "`return;`" statements to terminate without returning an actual value, or to not explicitly use any return statement, regardless of their declared return type. This flexibility can help when writing programs using certain functions only for their side effects. Nevertheless, as the dynamic semantics shows, a return value is automatically generated when an explicit `return` statement is not encountered.
- For simplicity, exceptions only throw and catch integer values. We leave it as an exercise to the reader to extend the semantics to allow throwing and catching arbitrary-type exceptions. To keep the definition simple, here we do not attempt to reject programs which throw uncaught exceptions.

Like in untyped SIMPLE, some constructs can be desugared into a smaller set of basic constructs. In general, it should be clear why a program does not type by looking at the top of the `k` cells in its stuck configuration.

4.1 Syntax

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

Types

Primitive, array and function types, as well as lists (or tuples) of types are supported. The lists of types are useful for function arguments.

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

SYNTAX   Types ::= List{Type,“,”}
```

Declarations

Variable and function declarations have the expected syntax. For variables, we just replaced the `var` keyword of untyped SIMPLE with a type. For functions, besides replacing the `function` keyword with a type, we also introduce a new syntactic category for typed variables, *Param*, and lists over it.

SYNTAX $\mathit{Param} ::= \mathit{Type}\ \ \mathit{Id}$

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

SYNTAX $\mathit{Decl} ::= \mathit{Type}\ \ \mathit{Exps}$;
$\quad\mid\ \mathit{Type}\ \ \mathit{Id}(\mathit{Params})\mathit{Block}$

Expressions

The syntax of expressions is identical to that in untyped SIMPLE, except for the logical conjunction and disjunction which have different strictness attributes, because they now have different evaluation strategies.

SYNTAX *Exp* ::= *Int* | *Bool* | *String* | *Id*
| (*Exp*) [bracket]
| `++` *Exp*
| *Exp*[*Exps*] [strict]
| *Exp*(*Exps*) [strict]
| `-` *Exp* [strict]
| `sizeOf` (*Exp*) [strict]
| `read` ()
| *Exp* `*` *Exp* [strict]
| *Exp* `/` *Exp* [strict] | *Exp* `%` *Exp* [strict]
| *Exp* `+` *Exp* [strict] | *Exp* `-` *Exp* [strict]
| *Exp* `<` *Exp* [strict] | *Exp* `<=` *Exp* [strict]
| *Exp* `>` *Exp* [strict] | *Exp* `>=` *Exp* [strict]
| *Exp* `==` *Exp* [strict] | *Exp* `!=` *Exp* [strict]
| `!` *Exp* [strict]
| *Exp* `&&` *Exp* [strict] | *Exp* `||` *Exp* [strict]
| `spawn` *Block*
| *Exp* `=` *Exp* [strict(2)]

Note that `spawn` has not been declared strict. This may seem unexpected, because the child thread shares the same environment with the parent thread, so from a typing perspective the spawned statement makes the same sense in a child thread as it makes in the parent thread. The reason for not declaring it strict is because we want to disallow programs where the spawned thread calls the `return` statement, because those programs would get stuck in the dynamic semantics. The type semantics of spawn below will reject such programs.

We still need lists of expressions, defined below, but we do not need lists of identifiers anymore. They have been replaced by the lists of parameters.

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

Statements

The statements have the same syntax as in untyped SIMPLE, except for the exceptions, which now type their parameter. Unlike in untyped SIMPLE, all statement constructs which have arguments and are not desugared are strict, including the conditional and the `while`. Indeed, from a typing perspective, they are all strict: first type their arguments and then type the actual construct.

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

SYNTAX $\mathit{Stmt} ::= \mathit{Decl}\ \mid \mathit{Block}$
| *Exp* ; [strict]

| `if` (*Exp*)*Block* `else` *Block* [strict] | `if` (*Exp*)*Block*
| `while` (*Exp*)*Block* [strict]
| `for` (*Stmt* *Exp* `;` *Exp*)*Block*
| `return` *Exp* `;` [strict] | `return;`
| `print` (*Exps*) `;` [strict]
| `try` *Block* `catch` (*Param*)*Block* [strict(1)]
| `throw` *Exp* `;` [strict]
| `join` *Exp* `;` [strict]
| `acquire` *Exp* `;` [strict] | `release` *Exp* `;` [strict]
| `rendezvous` *Exp* `;` [strict]

Statement composition is now sequentially strict, because, unlike in the dynamic semantics where statements dissolved, they now reduce to a type.

SYNTAX *Stmts* ::= *Stmt* | *Stmts* *Stmts* [seqstrict]

Desugaring macros

We use the same desugaring macros like in untyped SIMPLE, but, of course, adapted to the new syntax (e.g., including the types of the declared variables).

RULE

$$\frac{\texttt{if }(E)S}{\texttt{if }(E)S\texttt{ else }\{\}}$$

[macro]

RULE

$$\frac{\texttt{for }(\mathit{Start}\ \mathit{Cond}\texttt{ ; }\mathit{Step})\{S\}}{\{\mathit{Start}\ \texttt{while }(\mathit{Cond})\{S\ \mathit{Step}\texttt{ ;}\}\}}$$

[macro]

RULE

$$\frac{T\ E1\texttt{ , }E2\texttt{ , }Es\texttt{ ;}}{T\ E1\texttt{ ; }T\ E2\texttt{ , }Es\texttt{ ;}}$$

[macro]

RULE

$$\frac{T\ X\texttt{ = }E\texttt{ ;}}{T\ X\texttt{ ; }X\texttt{ = }E\texttt{ ;}}$$

[macro]

4.2 Static semantics

Here we define the type system of SIMPLE. Like concrete semantics, type systems defined in $\mathbb{K}$ are also executable. However, $\mathbb{K}$ type systems turn into type checkers instead of interpreters when executed.

The typing process is done in two (overlapping) phases. In the first phase the global environment is built, which contains type bindings for all the globally declared variables and functions. For functions, the declared types will be “trusted” during the first phase and simply bound to their corresponding function names and placed in the global type environment. At the same time, type-checking tasks that the function bodies indeed respect their claimed types are generated. All these tasks are verified during the second phase. This way, all the global variable and function declarations are available in the global type environment and can be used to type-check each function body. This is consistent with the semantics of untyped SIMPLE, where functions can access all the global variables and can call any other function declared in the same program. The two phases may overlap because of the $\mathbb{K}$ concurrent semantics. For example, a function task can be started while the first phase is still running; moreover, it may even complete before the first phase does, namely when all the global variables and functions that it needs have already been processed and made available in the global environment by the first phase task.

Extended syntax and results

The idea is to start with a configuration holding the program to type in one of its cells, then apply rewrite rules on it mixing types and language syntax, and eventually obtain a type instead of the original program. In other words, the program reduces to its type using the $\mathbb{K}$ rules giving the type system of the language. Additional typing tasks for function bodies are generated and solved the same way. If this rewriting process gets stuck, then the program is not well-typed; otherwise the program is well-typed (by definition). We did not need types for statements and blocks as part of the typed SIMPLE syntax, since programmers are not allowed to use such types explicitly. However, we need them in the type system, as blocks and statements reduce to them.

We start by allowing types to be used inside expressions and statements in our language. This way, types can be used together with language syntax in subsequent $\mathbb{K}$ rules without any parsing errors. We prefer to group the block and statement types under one syntactic sub-category of types, because this allows us to more compactly state that certain terms can be either blocks or statements. Also, since programs and fragments of program will reduce to their types, in order for the strictness and context declarations to be executable we state that types are results.

SYNTAX *BlockOrStmtType* ::= `block` | `stmt`

SYNTAX *Type* ::= *BlockOrStmtType*

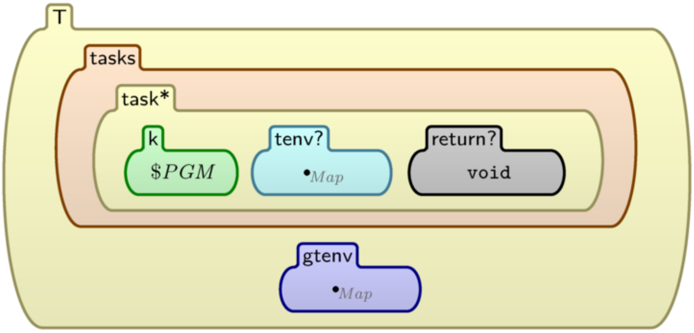
SYNTAX *Exp* ::= *Type*

SYNTAX *KResult* ::= *Type*

Configuration

The configuration of our type system consists of a tasks cell holding various typing task cells, and a global type environment.

CONFIGURATION:

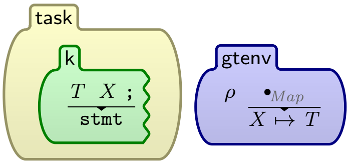


Each task includes a k cell holding the code to type, a tenv cell holding the local type environment, and a return cell holding the return type of the currently checked function. The latter is needed in order to check whether return statements return values of the expected type. Initially, the program is placed in a k cell inside a task cell. Since the cells with multiplicity "?" are not included in the initial configuration, the task cell holding the original program in its k cell will contain no other subcells.

Variable declarations

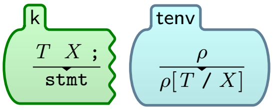
Variable declarations type as statements, that is, they reduce to the type `stmt`. There are only two cases that need to be considered: when a simple variable is declared and when an array variable is declared. The macros at the end of the syntax above take care of reducing other variable declarations, including ones where the declared variables are initialized, to only these two cases. The first case has two subcases: when the variable declaration is global (i.e., the `task` cell contains only the `k` cell), in which case it is added to the global type environment checking at the same time that the variable has not been already declared; and when the variable declaration is local (i.e., a `tenv` cell is available), in which case it is simply added to the local type environment, possibly shadowing previous homonymous variables. The third case reduces to the second, incrementally moving the array dimension into the type until the array becomes a simple variable.

RULE



when $\neg_{Bool} X$ **in keys** ρ

RULE



CONTEXT
$T\ \ X[\square]$;

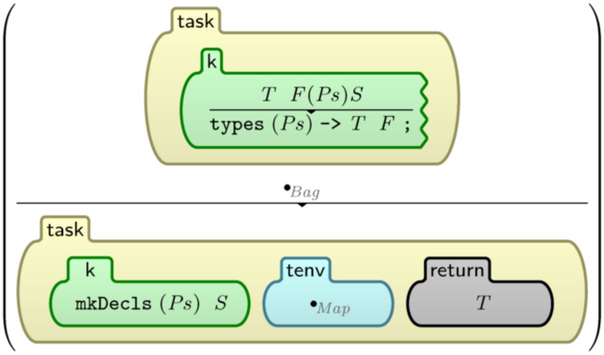
RULE
$$\frac{T\ \ E[\mathtt{int}\,,\ Ts]\ ;}{T[]\ \ E[Ts]\ ;}$$
[structural]

RULE
$$\frac{T\ \ E[\bullet_{Types}]\ ;}{T\ \ E\ ;}$$
[structural]

Function declarations

Functions are allowed to be declared only at the top level (the `task` cell holds only its `k` subcell). Each function declaration reduces to a variable declaration (a binding of its name to its declared function type), but also adds a task into the `tasks` cell. The task consists of a typing of the statement declaring all the function parameters followed by the function body, together with the expected return type of the function. The `types` and `mkDecls` functions, defined at the end of the definition in the section on auxiliary operations, extract the list of types and make a sequence of variable declarations from a list of function parameters, respectively. Note that, although in the dynamic semantics we include a terminating `return` statement at the end of the function body to eliminate from the analysis the case when the function does not provide an explicit return, we do not need to include such a similar `return` statement here. That's because the `return` statements type to `stmt` anyway, and the entire code of the function body needs to type.

RULE

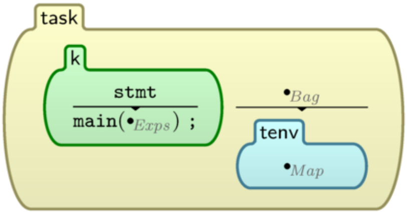


[structural]

Checking if `main()` *exists*

Once the entire program is processed (generating appropriate tasks to type check its function bodies), we can dissolve the main `task` cell (the one holding only a `k` subcell). Since we want to enforce that programs include a main function, we also generate a function task executing `main()` to ensure that it types (remove this task creation if you do not want your type system to reject programs without a `main` function).

RULE

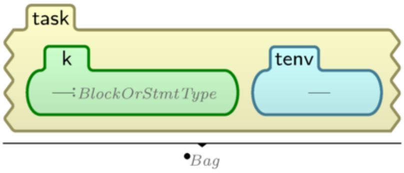


[structural]

Collecting the terminated tasks

Similarly, once a non-main task (i.e., one which contains a `tenv` subcell) is completed using the subsequent rules (i.e., its `k` cell holds only the `block` or `stmt` type), we can dissolve its corresponding cell. Note that it is important to ensure that we only dissolve tasks containing a `tenv` cell with the rule below, because the main task should *not* dissolve this way! It should do what the above rule says. In the end, there should be no task cell left in the configuration when the program correctly type checks (—:$Sort$ stands for an anonymous variable, —, enforced to have the sort *Sort* in order for the rule to apply).

RULE



Basic values

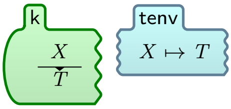
The first three rewrite rules below reduce the primitive values to their types, as we typically do when we define type systems in $\mathbb{K}$.

RULE $\dfrac{\_ : Int}{\texttt{int}}$ RULE $\dfrac{\_ : Bool}{\texttt{bool}}$ RULE $\dfrac{\_ : String}{\texttt{string}}$

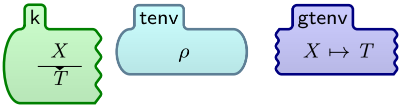
Variable lookup

There are three cases to distinguish for variable lookup: (1) if the variable is bound in the local type environment, then look its type up there; (2) if a local environment exists and the variable is not bound in it, then look its type up in the global environment; (3) finally, if there is no local environment, meaning that we are executing the top-level pass, then look the variable's type up in the global environment, too.

RULE

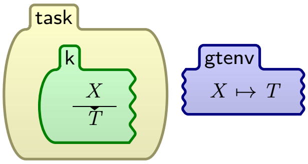


RULE



when $\neg_{Bool} X$ `in keys` ρ

RULE



Increment

We want the increment operation to apply to any lvalue, including array elements, not only to variables. For that reason, we define a special context evaluating the type of the argument of the increment operation only if that argument is an lvalue. Otherwise the rewriting process gets stuck. The `ltype` context is defined in the auxiliary operations section at the end of this definition. It essentially acts as a filter, getting stuck if its argument is not an lvalue and letting it reduce otherwise. The type of the lvalue is expected to be an integer in order to be allowed to be incremented, as seen in the rule "`++ int => int`" below.

CONTEXT $\texttt{++}\ \dfrac{\square}{\texttt{ltype}\,(\square)}$ RULE $\dfrac{\texttt{++ int}}{\texttt{int}}$

Common expression constructs

The rules below are straightforward and self-explanatory:

RULE
$$\frac{\texttt{int + int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{string + string}}{\texttt{string}}$$

RULE
$$\frac{\texttt{int - int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{int * int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{int / int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{int \% int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{- int}}{\texttt{int}}$$

RULE
$$\frac{\texttt{int < int}}{\texttt{bool}}$$

RULE
$$\frac{\texttt{int <= int}}{\texttt{bool}}$$

RULE
$$\frac{\texttt{int > int}}{\texttt{bool}}$$

RULE
$$\frac{\texttt{int >= int}}{\texttt{bool}}$$

RULE
$$\frac{T \texttt{ == } T}{\texttt{bool}}$$

RULE
$$\frac{T \texttt{ != } T}{\texttt{bool}}$$

RULE
$$\frac{\texttt{bool \&\& bool}}{\texttt{bool}}$$

RULE
$$\frac{\texttt{bool || bool}}{\texttt{bool}}$$

RULE
$$\frac{\texttt{! bool}}{\texttt{bool}}$$

Array access and size

Array access requires each index to type to an integer, and the array type to be at least as deep as the number of indexes:

RULE
$$\frac{T[\,][\texttt{int}\texttt{,}\ Ts]}{T[Ts]}$$

RULE
$$\frac{T[\bullet_{Types}]}{T}$$

sizeOf only needs to check that its argument is an array:

RULE
$$\frac{\texttt{sizeOf}\ (T[\,])}{\texttt{int}}$$

Input/Output

The read expression construct types to an integer, while print types to a statement provided that all its arguments type to integers or strings.

RULE
$$\frac{\texttt{read}\ ()}{\texttt{int}}$$

RULE
$$\texttt{print}\ \frac{(T\texttt{,}\ Ts)}{Ts}\ \texttt{;}$$
when $T =_K \texttt{int} \vee_{Bool} T =_K \texttt{string}$

RULE
$$\frac{\texttt{print}\ (\bullet_{Types})\ \texttt{;}}{\texttt{stmt}}$$

Assignment

The special context and the rule for assignment below are similar to those for increment: the left-hand-side of the assignment must be an lvalue and, in that case, it must have the same type as the right-hand-side, which then becomes the type of the assignment.

CONTEXT
$$\frac{\square}{\texttt{ltype}\ (\square)}\ \texttt{=}\ \text{—}$$

RULE
$$\frac{T \texttt{ = } T}{T}$$

Function application and return

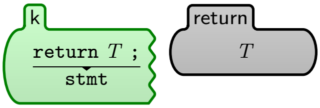
Function application requires the type of the function and the types of the passed values to be compatible. Note that a special case is needed to handle the no-argument case:

RULE
$$\frac{(Ts\ \texttt{->}\ T)(Ts)}{T}$$
when $Ts \neq_K \bullet_{Types}$

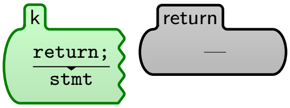
RULE
$$\frac{(\texttt{void}\ \texttt{->}\ T)(\bullet_{Types})}{T}$$

The returned value must have the same type as the declared function return type. If an empty return is encountered, then we should check that we are in a function (and not a thread) context, that is, a `return` cell must be available:

RULE



RULE

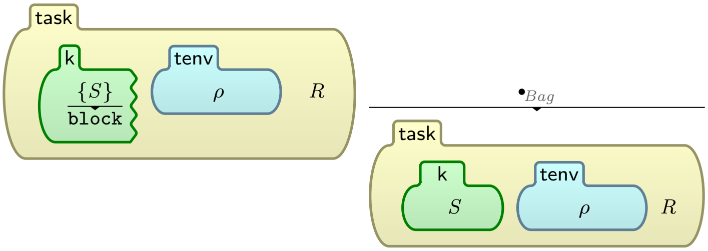


Blocks

To avoid having to recover type environments after blocks, we prefer to start a new task for block body, making sure that the new task is passed the same type environment and return cells.

RULE
$$\frac{\{\}}{\texttt{block}}$$

RULE



Expression statement

RULE
$$\frac{T\ ;}{\texttt{stmt}}$$

Conditional and `while` *loop*

RULE
$$\frac{\texttt{if (bool)block else block}}{\texttt{stmt}}$$

RULE
$$\frac{\texttt{while (bool)block}}{\texttt{stmt}}$$

Exceptions

Recall that the try-catch block was declared strict in its first argument. Therefore, we expect that the first argument has evaluated to the type value stmt. We currently force the parameters of exceptions to only be integers. Moreover, for simplicity, we assume that integer exceptions can be thrown from anywhere, including from functions which do not define any try-catch block (with the currently unchecked—also for simplicity—expectation that the caller functions would catch those exceptions).

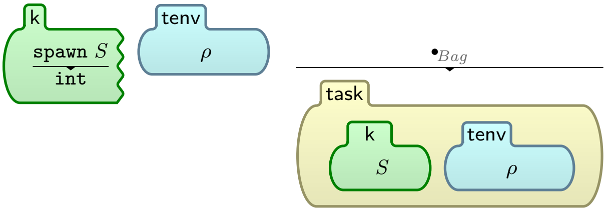
RULE
$$\frac{\texttt{try block catch (int } X)\{S\}}{\{\texttt{int } X\ \texttt{;}\ S\}}$$
[structural]

RULE
$$\frac{\texttt{throw int ;}}{\texttt{stmt}}$$

Concurrency

When typing concurrency constructs we do not want the spawned thread to return, so we do not include any return cell in the new task cell for the thread statement. As with the functions semantics defined above, we do not check for thrown exceptions which are not caught.

RULE



RULE
$$\frac{\texttt{join int ;}}{\texttt{stmt}}$$

RULE
$$\frac{\texttt{acquire } T\ \texttt{;}}{\texttt{stmt}}$$

RULE
$$\frac{\texttt{release } T\ \texttt{;}}{\texttt{stmt}}$$

RULE
$$\frac{\texttt{rendezvous } T\ \texttt{;}}{\texttt{stmt}}$$

RULE
$$\frac{\text{—}{:}\mathit{BlockOrStmtType}\quad \text{—}{:}\mathit{BlockOrStmtType}}{\texttt{stmt}}$$

Auxiliary constructs

The function mkDecls turns a list of parameters into a list of variable declarations.

SYNTAX $\mathit{Decl} ::= \texttt{mkDecls}\ (\mathit{Params})$ [function]

RULE
$$\frac{\texttt{mkDecls}\ (T\ \ X\ \texttt{,}\ \mathit{Ps})}{T\ \ X\ \texttt{;}\ \texttt{mkDecls}\ (\mathit{Ps})}$$

RULE
$$\frac{\texttt{mkDecls}\ (\bullet_{\mathit{Params}})}{\{\}}$$

The ltype context allows only expressions which can evaluate to an lvalue. To achieve this, we define a sort *LValue* to consist of program variables and array accesses and semantically constrain the hole of the ltype context to have the *LValue* sort.

SYNTAX $\mathit{LValue} ::= \mathit{Id}$
$\qquad\qquad\qquad\mid\ \mathit{Exp}[\mathit{Exp}]$

SYNTAX $\mathit{Exp} ::= \texttt{ltype}\ (\mathit{Exp})$

CONTEXT
$\texttt{ltype}\ (\square{:}\mathit{LValue})$

Note that there is no explicit rule for the ltype construct. One reason is that ltype's function as a filter is enough: once an expression is allowed to be evaluated, its corresponding type will be obtained using the other rules in the definition. The second reason is that, similarly to the lvalue construct from the dynamic semantics of SIMPLE, discussed in Sections 2.4 and 3.7, ltype is added as a wrapper by the heating rule generated by one of the context declarations introducing it, and it is removed by the corresponding cooling rule. For increment, those rules are:

RULE
$$\frac{\texttt{++}\ E}{\texttt{ltype}\ (E) \curvearrowright \texttt{++}\ \square}$$
[structural]

RULE
$$\frac{\texttt{ltype}\ (E) \curvearrowright \texttt{++}\ \square}{\texttt{++}\ E}$$
[structural]

The function types returns the list of types for a list of parameters.

SYNTAX $\mathit{Types} ::= \texttt{types}\ (\mathit{Params})$ [function]

RULE
$$\frac{\texttt{types}\ (T\ \_{:}\mathit{Id})}{T\ ,\ \bullet_{\mathit{Types}}}$$

RULE
$$\frac{\texttt{types}\ (T\ \_{:}\mathit{Id}\ ,\ P\ ,\ \mathit{Ps})}{T\ ,\ \texttt{types}\ (P\ ,\ \mathit{Ps})}$$

RULE
$$\frac{\texttt{types}\ (\bullet_{\mathit{Params}})}{\texttt{void}\ ,\ \bullet_{\mathit{Types}}}$$

This concludes the static definition of SIMPLE.

5 Language Definitions and Tools using $\mathbb{K}$

Besides didactic and prototypical languages (such as the lambda calculus, System F, and Agents), the $\mathbb{K}$ tool has been used to formalize several existing programming languages or paradigms and to design and develop (language-parametric) analysis and verification tools.

Programming languages research

$\mathbb{K}$ has been successfully used to formally and completely define the C programming language [17] and Scheme [29]. Additionally, $\mathbb{K}$ has been used in formalizing various aspects of features of languages like Haskell [27], Javascript, X10 [21], a RISC assembly language [7,6], and LLVM [16], as well as a framework for domain specific languages [50,51].

$\mathbb{K}$'s ability to easily express concurrent computations has been used in researching safe models for concurrency [24], synchronization of agent systems [15], models for P-Systems [56,11], and for the relaxed memory model of x86-TSO [52].

Analysis tools

Regarding analysis tools, $\mathbb{K}$ has been used for designing type checkers and type inferencers [18], for model checking executions with predicate abstraction [4,2] and heap awareness [49], for symbolic execution [5,3,1], computing worst case execution times [9,8], studying program equivalence [28], or researching runtime verification techniques [42,52]. Additionally, the C definition mentioned above has been used as a program undefinedness checker to analyze C programs [38].

Program Verification

$\mathbb{K}$ served as an inspiration for the design of Reachability Logic [48,45], a new logic for verification based on matching logic [41], unifying operational and axiomatic semantics [47], generalizing both Hoare logic and separation logic [46], which serves as basis for a new program verification tool for 𝕂 definitions using Hoare-like assertions [44].

All these definitions and analysis tools can be found on the 𝕂 tool website [26]. Other language definitions and analysis tools developed using the 𝕂 technique before the development of the 𝕂 tool include definitions of Java [19] and Verilog [30], as well as a static policy checker for C [25].

6 Conclusion

The 𝕂 semantic framework, consisting of a general-purpose concurrent rewriting approach together with a definitional technique specialized for concurrent programming languages and systems, brings together the advantages of existing language definitional frameworks while avoiding their limitations.

In spite of its youth, the 𝕂 framework has already proven to be practical, as it has been used with relatively little effort to define complex languages like Java, Scheme, Verilog, or C, and to use those definitions for analyzing programs written in those languages. 𝕂 is currently under heavy development, with bugs being fixed and new features and capabilities added on a regular basis. This is all possible due to the enthusiasm and strong belief of its designers and developers that 𝕂 can be not only an academic exercise but also a solid, practical and scalable tool for programming language design and analysis, as well as due to generous funding under the NSA contract H98230-10-C-0294, the NSF grant CCF-0916893, and the (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

References

[1] Arusoaie, A., D. Lucanu and V. Rusu, *A generic framework for symbolic execution*, in: *SLE*, Lecture Notes in Computer Science **8225**, 2013, pp. 281–301.

[2] Asavoae, I. M., *Systematic design of abstractions in K*, in: *WADT (preliminary proceedings)*, TR-08/12, Universidad Complutense de Madrid, 2012, p. 9.
URL http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf

[3] Asavoae, I. M., *Abstract semantics for alias analysis in K*, in: M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, 2013, in this issue.

[4] Asavoae, I. M. and M. Asavoae, *Collecting semantics under predicate abstraction in the K framework*, in: P. C. Ölveczky, editor, *WRLA*, Lecture Notes in Computer Science **6381** (2010), pp. 123–139.

[5] Asavoae, I. M., M. Asavoae and D. Lucanu, *Path directed symbolic execution in the K framework*, in: T. Ida, V. Negru, T. Jebelean, D. Petcu, S. M. Watt and D. Zaharie, editors, *SYNASC* (2010), pp. 133–141.

[6] Asavoae, M., *A K-based methodology for modular design of embedded systems*, in: *WADT (preliminary proceedings)*, TR-08/12, Universidad Complutense de Madrid, 2012, p. 16.
URL http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf

[7] Asavoae, M., *K semantics for assembly languages : A case study*, in: M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, 2013, in this issue.

[8] Asavoae, M., I. M. Asavoae and D. Lucanu, *On abstractions for timing analysis in the K framework*, in: R. Pena, M. Eekelen and O. Shkaravska, editors, *FOPARA*, Lecture Notes in Computer Science **7177** (2012), pp. 90–107.
URL http://dx.doi.org/10.1007/978-3-642-32495-6_6

[9] Asăvoae, M., D. Lucanu and G. Roşu, *Towards semantics-based WCET analysis*, in: C. Healy, editor, *WCET*, Austian Computer Society (OCG), 2011.

[10] Berry, G. and G. Boudol, *The chemical abstract machine*, Theoretical Computer Science **96** (1992), pp. 217–248.

[11] Chira, C., T.-F. Serbanuta and G. Stefanescu, *P systems with control nuclei: The concept*, Journal of Logic and Algebraic Programming **79** (2010), pp. 326–333.

[12] Clavel, M., F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet and C. Talcott, "All About Maude, A High-Performance Logical Framework," Lecture Notes in Computer Science **4350**, Springer, 2007.

[13] Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, *Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach*, in: G. Rozenberg, editor, *Handbook of Graph Grammars* (1997), pp. 163–246.

[14] Danvy, O. and L. R. Nielsen, *Refocusing in reduction semantics*, Technical Report BRICS RS-04-26, University of Aarhus (2004).

[15] Dinges, P. and G. Agha, *Scoped synchronization constraints for large scale actor systems*, in: M. Sirjani, editor, *COORDINATION*, Lecture Notes in Computer Science **7274** (2012), pp. 89–103.

[16] Ellison, C. and D. Lazar, *K definition of the LLVM assembly language* (2012). URL https://github.com/davidlazar/llvm-semantics

[17] Ellison, C. and G. Roșu, *An executable formal semantics of C with applications*, in: J. Field and M. Hicks, editors, *POPL* (2012), pp. 533–544.

[18] Ellison, C., T. F. Șerbănuță and G. Roșu, *A rewriting logic approach to type inference*, in: A. Corradini and U. Montanari, editors, *WADT*, Lecture Notes in Computer Science **5486** (2008), pp. 135–151.

[19] Farzan, A., F. Chen, J. Meseguer and G. Roșu, *Formal analysis of Java programs in JavaFAN*, in: R. Alur and D. Peled, editors, *CAV*, Lecture Notes in Computer Science **3114** (2004), pp. 501–505.

[20] Felleisen, M. and R. Hieb, *A revised report on the syntactic theories of sequential control and state*, Theoretical Computer Science **103** (1992), pp. 235–271.

[21] Gligoric, M., D. Marinov and S. Kamin, *CoDeSe: fast deserialization via code generation*, in: M. B. Dwyer and F. Tip, editors, *ISSTA* (2011), pp. 298–308.

[22] Goguen, J., T. Winkler, J. Meseguer, K. Futatsugi and J.-P. Jouannaud, *Introducing OBJ*, in: *Software Engineering with OBJ: algebraic specification in action*, Kluwer, 2000 .

[23] Goguen, J. A. and G. Malcolm, "Algebraic Semantics of Imperative Programs," Foundations of Computing, The MIT Press, 1996.

[24] Heumann, S., V. S. Adve and S. Wang, *The tasks with effects model for safe concurrency*, in: A. Nicolau, X. Shen, S. P. Amarasinghe and R. Vuduc, editors, *PPOPP* (2013), pp. 239–250.

[25] Hills, M., F. Chen and G. Roșu, *A rewriting logic approach to static checking of units of measurement in C*, in: G. Kniesel and J. S. Pinto, editors, *RULE'08*, Electronic Notes in Theoretical Computer Science **290** (2012), pp. 51–67.

[26] *K semantic framework website* (2013). URL https://k-framework.org

[27] Lazar, D., *K definition of Haskell'98* (2012). URL https://github.com/davidlazar/haskell-semantics

[28] Lucanu, D. and V. Rusu, *Program Equivalence by Circular Reasoning*, Technical Report RR-8116, INRIA (2012). URL http://hal.inria.fr/hal-00744374

[29] Meredith, P., M. Hills and G. Roșu, *An executable rewriting logic semantics of K-Scheme*, in: D. Dubé, editor, *SCHEME* (2007), pp. 91–103. URL http://www.schemeworkshop.org/2007/procFront.pdf

[30] Meredith, P. O., M. Katelman, J. Meseguer and G. Roșu, *A formal executable semantics of Verilog*, in: *MEMOCODE* (2010), pp. 179–188.

[31] Meseguer, J., *Rewriting as a unified model of concurrency*, in: J. C. M. Baeten and J. W. Klop, editors, *CONCUR*, Lecture Notes in Computer Science **458** (1990), pp. 384–400.

[32] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[33] Meseguer, J., M. Palomino and N. Martí-Oliet, *Equational abstractions*, Theoretical Computer Science **403** (2008), pp. 239–264.

[34] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

[35] Mosses, P. D., "CASL Reference Manual," Lecture Notes in Computer Science **2960**, Springer, 2004.

[36] Mosses, P. D., *Modular structural operational semantics*, Journal of Logic and Algebraic Programming **60-61** (2004), pp. 195–228.

[37] Plotkin, G. D., *A structural approach to operational semantics*, Journal of Logic and Algebraic Programming **60-61** (2004), pp. 17–139.

[38] Regehr, J., Y. Chen, P. Cuoq, E. Eide, C. Ellison and X. Yang, *Test-case reduction for C compiler bugs*, in: J. Vitek, H. Lin and F. Tip, editors, *PLDI* (2012), pp. 335–346.

[39] Reynolds, J. C., *The discoveries of continuations*, Lisp Symbolic Computation **6** (1993), pp. 233–248.

[40] Roşu, G., *CS322, Fall 2003 - Programming language design: Lecture notes*, Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science (2003), lecture notes of a course taught at UIUC.

[41] Roșu, G., C. Ellison and W. Schulte, *Matching logic: An alternative to Hoare/Floyd logic*, in: M. Johnson and D. Pavlovic, editors, *AMAST*, Lecture Notes in Computer Science **6486** (2010), pp. 142–162.

[42] Roșu, G., W. Schulte and T. F. Șerbănuță, *Runtime verification of C memory safety*, in: S. Bensalem and D. Peled, editors, *RV*, Lecture Notes in Computer Science **5779** (2009), pp. 132–151.

[43] Roşu, G. and T. F. Şerbănuţă, *An overview of the K semantic framework*, Journal of Logic and Algebraic Programming **79** (2010), pp. 397–434.

[44] Roșu, G. and A. Ștefănescu, *Matching logic: a new program verification approach*, in: R. N. Taylor, H. Gall and N. Medvidovic, editors, *ICSE* (2011), pp. 868–871.

[45] Rosu, G. and A. Stefanescu, *Checking reachability using matching logic*, in: G. T. Leavens and M. B. Dwyer, editors, *OOPSLA* (2012), pp. 555–574.

[46] Rosu, G. and A. Stefanescu, *From Hoare logic to matching logic reachability*, in: D. Giannakopoulou and D. Méry, editors, *FM*, Lecture Notes in Computer Science **7436** (2012), pp. 387–402.

[47] Rosu, G. and A. Stefanescu, *Towards a unified theory of operational and axiomatic semantics*, in: A. Czumaj, K. Mehlhorn, A. M. Pitts and R. Wattenhofer, editors, *ICALP (2)*, Lecture Notes in Computer Science **7392** (2012), pp. 351–363.

[48] Roșu, G., A. Ștefănescu, Ș. Ciobâcă and B. Moore, *One-path reachability logic*, in: *LICS*, 2013, pp. 358–367.

[49] Rot, J., I. M. Asavoae, F. S. de Boer, M. M. Bonsangue and D. Lucanu, *Interacting via the heap in the presence of recursion*, in: M. Carbone, I. Lanese, A. Silva and A. Sokolova, editors, *ICE*, Electronic Proceedings in Theoretical Computer Science **104**, 2012, pp. 99–113.

[50] Rusu, V. and D. Lucanu, *A K-based formal framework for domain-specific modelling languages*, in: B. Beckert, F. Damiani and D. Gurov, editors, *FoVeOOS*, Lecture Notes in Computer Science **7421** (2011), pp. 214–231.

[51] Rusu, V. and D. Lucanu, *K semantics for OCL—a proposal for a formal definition for OCL*, in: M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, 2013, in this issue.

[52] Şerbănuţă, T. F., "A Rewriting Approach to Concurrent Programming Language Design and Semantics," Ph.D. thesis, University of Illinois at Urbana-Champaign (2010), https://www.ideals.illinois.edu/handle/2142/18252.

[53] Șerbănuță, T. F., A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu and G. Roșu, *The K primer (version 2.5)*, in: M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, 2013, in this issue.

[54] Șerbănuță, T. F. and G. Roșu, *A truly concurrent semantics for the K framework based on graph transformations*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *ICGT*, Lecture Notes in Computer Science **7562** (2012), pp. 294–310.

[55] Şerbănuţă, T. F., G. Roşu and J. Meseguer, *A rewriting logic approach to operational semantics*, Information and Computation **207** (2009), pp. 305–340.

[56] Şerbănuţă, T. F., G. Ştefănescu and G. Roşu, *Defining and executing P systems with structured data in K*, in: D. W. Corne, P. Frisco, G. Păun, G. Rozenberg and A. Salomaa, editors, *WMC*, Lecture Notes in Computer Science **5391** (2008), pp. 374–393.

[57] Viry, P., *Equational rules for rewriting logic*, Theoretical Computer Science **285** (2002), pp. 487–517.