WRLA'10, LNCS 2010, pp 104-122
Springer (to appear) 2010

# K-Maude: A Rewriting Based Tool for Semantics of Programming Languages

Traian Florin Șerbănuță and Grigore Roșu
University of Illinois at Urbana-Champaign

**Abstract.** K is a rewriting-based framework for defining programming languages. K-Maude is a tool implementing K on top of Maude. K-Maude provides an interface accepting K modules along with regular Maude modules and a collection of tools for transforming K language definitions into Maude rewrite theories for execution or analysis, or into LaTeX for documentation purposes. The current K-Maude prototype was successfully used in defining several languages and language analysis tools, both for research and for teaching purposes. This paper describes the K-Maude tool, both from a user and from an implementer perspective.

## 1 Introduction

There is overwhelming evidence by now that rewriting logic [4] is a powerful framework for programming language design, semantics and analysis (there are too many papers on these topics to cite; we recommend the interested reader to consult the rewriting logic semantics project [5, 9] and the references there). There are two major reasons for that: (1) on the one hand, existing language definitional approaches such as structural operational semantics (with [11] or without [7] evaluation contexts, modular [6] or not) and natural semantics [3] can be faithfully captured by rewriting logic, so one can use rewriting logic and Maude [2] to define and analyze languages using these formalisms, and (2) on the other hand, rewriting logic, thanks to its generality and powerful tool support, encourages the development of new language definitional approaches.

The K framework [8] is a semantic definitional framework inspired from rewriting logic but specialized and optimized for programming languages. It consists of three components: a concurrent rewrite abstract machine, a language definitional technique, and a specialized notation. The aim of the concurrent rewrite abstract machine is to increase the potential for concurrency of a rewrite theory by allowing rules which overlap but do not change the overlapped sub-term (e.g., two threads writing in different locations in the store) to apply concurrently; the concurrency aspect of K is beyond the scope of this paper, so we do not discuss it here. We will briefly recall the K language definitional technique in Section 2, but this paper is essentially related to the K specialized notation.

The K technique has been manually (without automated tool support) used in the context of rewriting logic and Maude for more than five years, for teaching programming language and program verification courses as well as for several research projects. Such manual uses of K in Maude turned our to be verbose and error prone, because Maude is a general purpose rewrite engine not specifically optimized for programming languages. Thus, the idea of developing a K specialized layer on top of Maude came naturally. The resulting integrated toolkit is called

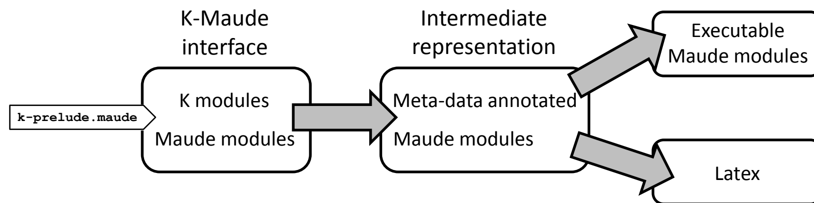



**Fig. 1.** K-Maude overview. Grayed arrows correspond to translating tools.

K-Maude and is the subject of this paper. Figure 1 shows the architecture of K-Maude. The gray arrows represent translators implemented as part of the K-Maude toolkit. A file `k-prelude.maude` contains several Maude modules that a handy in most language definitions, such as ones for defining computations, configurations, environments, stores, etc. It is highly recommended that this file is included in all K definitions, with or without using K-Maude. The K-Maude interface is what the user typically sees: besides usual Maude modules (K-Maude fully extends Maude), one can also include K modules comprising syntax, semantics or configuration definitions using the K specialized notation.

A first component of K-Maude translates K modules to Maude modules. The resulting Maude modules serve as an intermediate representation of K-Maude definitions and are not intended to be executable. However, they are heavily annotated with meta-data attributes. Skilled Maude users may write K language definitions directly in this Maude intermediate representation. This intermediate representation can be further translated to different back-ends. We provide two such translators, one to executable/analyzable Maude and one to Latex. The former yields actual executable language definitions in Maude which can serve as interpreters for the defined languages or as a basis for formal analysis, while the latter is useful for documentation purposes. Indeed, we believe that K can be used by ordinary language designers as a formal notation for rigorously specifying the semantics of their languages, the same way context-free grammars are used for formally specifying syntax, so a user-friendly latex notation may be preferred.

Section 2 briefly discusses the K definitional framework. Section 3 gives a user perspective of K-Maude, both wrt its builtin features and how it can be used. Section 4 briefly discusses the intermediate representation of K modules as Maude modules. Section 5 describes how K-Maude is translated to Maude, so that language designers can execute and formally analyze their K language definitions using Maude, and Section 6 describes how K-Maude is translated to Latex, so that language designers can visualize their language definitions.

## 2 K: A Rewriting-Based Framework for Computations

K [**?**] is a rewriting-based language definitional framework based on intuitions from the chemical abstract machine (CHAM) [1], evaluation contexts [11] and continuations [10]. The idea underlying language semantics in K is to represent the program configuration as a "nested soup" structure, which contains the context needed for the computation, with elements of the context represented as multisets or lists each stored inside a corresponding *cell*; a cell may also contain other cells.

Cells generally include standard items such as environments, stores, etc, as well as items specific to the given semantics, and are written using the notation $\langle ... \rangle_{env}$, etc.; when written in ascii, such as in K-Maude, we prefer to use the XML-like notation `<env> ... </env>`. One regularly used cell, referred to as k, represents the current *computation structure* of sort $K$, or simply the *computation*, which is a $\curvearrowright$-separated list of tasks, such as $t_1 \curvearrowright t_2 \curvearrowright ... \curvearrowright t_n$. Another, $\top$, represents the entire configuration structure.

A K definition consists of two types of sentences: structural rules (often reversible, like equations) and computational rules (typically non-reversible). Structural rules carry no computational meaning; instead, borrowing a concept from CHAMs, structural rules can *heat* and *cool* computations. When a computation is heated, it breaks into smaller pieces, exposing subexpressions of more complex expressions for evaluation. Cooling reverses this process, reassembling the (potentially modified) pieces into a computation with the same "shape". The following are examples of structural rules:

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$
$$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightleftharpoons b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2$$

Language syntax is completely abstract in K, in the sense that each language construct is a *KLabel* which is applied to other computations, i.e., terms of sort $K$; for convenience, we continue to use the mix-fix notation for syntax, like above. Unlike in evaluation contexts, $\square$ is not a "hole," but rather part of a *KLabel*, carrying the obvious "plug" intuition; e.g., the *KLabel*s involving $\square$ above are $\square + _$ (in the first equation) and $\mathsf{if}\ \square\ \mathsf{then_\ else_}$ (in the second).

Many structural rules can be automatically generated by annotating constructs in the language syntax with *strict* attributes: a *strict* attribute generates the appropriate structural rules for each strict argument. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute; for example, the two equations directly above correspond to the attributes *strict* for $_ + _$ (i.e., strict in all arguments, with the heating/cooling rules for the second operand not shown) and *strict*(1) for $\mathsf{if_\ then_\ else_}$. Once can also define evaluation contexts in K, by stating that they are strict in certain variables; for example, assuming a C-like language, an attribute *strict*($L$) associated to the term $*L = E$ says that (the l-value) $L$ needs to be first evaluated (before the assignment can be defined).

Computational rules represent actual steps of computation. The following are examples of computational rewrite rules:

$$i_1 + i_2 \rightarrow i, \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2$$
$$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1$$
$$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2$$

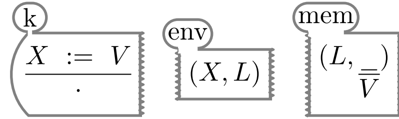
In addition to rules like the above, of the form "$l \rightarrow r$", K allows one to also write rules using the following contextual notation ($C$ is a multi-context):

$$C[\underset{t'_1}{\underline{t_1}}, \underset{t'_2}{\underline{t_2}}, ..., \underset{t'_n}{\underline{t_n}}]$$

which says that in context $C$, $t_i$ rewrites to $t'_i$ for each $i \in \{1, ..., n\}$. The context $C$ can be concurrently shared by various rules, which can apply concurrently

provided that none of them changes $C$ (they can "read only" $C$). If one ignores this concurrency aspect, then one can translate each K contextual rule into a rule $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$; this is precisely what K-Maude does.

When cells contains list or multi-set soups we take the liberty to draw them as round-ended boxes tagged at the top with their labels. Moreover, when more cell items are allowed to the left or to the right of the items of interest, the cell is allowed to be "ragged" in that direction. For example, the following is the K rule for variable assignment in a language with both an environment mapping variable names to locations and a store mapping locations to values:




The above rules says that if the assignment $X := V$ is the first computational task, and if $X$ is at location $L$ in the environment, then replace whatever is at location $L$ in the store (_ is a generic variable) by $V$ and discard the assignment ($\cdot$ is the unit of the list construct $_ \curvearrowright _$ for sort $K$). When using ascii notation, for example in K-Maude, we write the above as follows: `<k>[[X := V ==> .]] ...</k> <env>... (X,L) ...</env> <mem>... (L,[[_ ==> V]]) ...</mem>` .

## 3 K-Maude Interface

For the purpose of this paper, K can be regarded as a notational layer on top of rewriting logic, specialized and optimized for writing definitions of complex programming languages and models. Since our aim for K-Maude is to fully support rewriting logic and Maude, we implemented it as an extension of Maude. Full Maude is a reflective definition of Maude allowing for experimentation with new features and interface extensions. Since K-Maude introduces specific notation for rules, which require adding specific syntax to Maude modules, it is implemented as an extension of Full Maude. Consequently, one is free to use or not the K notation when writing language definitions. An extreme approach, which could be convenient for existing Maude users who want to gradually get exposed to K, is to only include the provided `k-prelude.maude` and then follow the K language definitional technique but use plain Maude, the same way one can give SOS or other semantic definitions using plain Maude.

### 3.1 K-Maude Core

The core syntax of K-Maude can be found in `k-prelude.maude`. K is a language of lists and sets, so the K-Maude core needs to provide them. Moreover, the idea of *computation* is at the core of the K framework; therefore, the K-Maude core provides the basic bricks for building computations. Finally, the core provides minimal support for describing configurations as "nested soups" of cells.
*Lists, (multi-)sets, maps* are key constructs in K. To support them, K-Maude provides specific parametric modules for lists (`KLIST`), multi-sets (`KBAG`), and maps (`KMAP`), which resemble the generic ones defined in the Maude prelude, but define a common subsort (containing the empty structure) and a common supersort, to ease the use the parametric modules for sorts in a subsorting

relation. For the same reason, the `SUBSORT` module takes two `TRIV` parameters, and includes lists and sets for these parameters, ensuring that the corresponding `Elt` sorts are subsorted, as well as all corresponding `KLIST` and `KBAG` sorts.

**mod** SUBSORT{*X* :: TRIV, Y :: TRIV} **is including** KBAG{*X*} + KBAG{Y} .
 **subsort** X$Elt < Y$Elt .
 **subsort** NeList{*X*} < NeList{Y} . **subsort** List{*X*} < List{Y} .
 **subsort** NeSet{*X*} < NeSet{Y} . **subsort** Set{*X*} < Set{Y} .
**endm**

*Basic computation syntax* is introduced in module `K`. A computation is a term of a specific sort $K$, and is defined as a list of tasks with identity '$.K$' and constructor '$_ \curvearrowright _$' (in ascii '`_~>_`'), as well as a way of building structured computations by applying labels on top of lists of computations:

 **op** $_\curvearrowright_$ : $K$ $K \to K$ [prec 100 assoc id: $.K$] .
 **op** _(_) : KLabel List{$K$} $\to K$ [prec 0 gather(& &)] .

*Result and proper computations* are distinguished to allow for a computational treatment of strictness rules. We distinguish two subsorts of $K$, namely: (1) *KResult*, meant to describe *results*, or computations which need no further evaluation; and (2) *KProper*, for computations incompletely evaluated. We additionally introduce *KResultLabel* and *KProperLabel*, as subsorts of *KLabel*, together with their corresponding application constructors:

 **op** _(_) : KProperLabel List{$K$} $\to$KProper [ditto] .
 **op** _(_) : KResultLabel List{$K$} $\to$KResult [ditto] .

*The program configuration* is a structured "soup" of cells. The basic configuration components are of sort *ConfigItem*, which is predefined as well and is expected to be constructed from cells. For example, if one prefers to define cells manually (as discussed in Section 3.4, K-Maude provides special support for defining configuration cells) then one may define the computation cell $k$ as follows, using an XML-like convention for grouping data (the XML-like notation is only required when one uses the builtin cell support of K-Maude — see Section 3.4): '**op** <k>_</k> : K $\to$ConfigItem [prec 0] .'. Having a unique configuration item sort makes cell nesting easy; for example, this is the definition of the top configuration cell: '**op** <T>_</T> : Set{ConfigItem} $\to$ConfigItem [prec 0] .'

### 3.2 K-Maude specific modules

To distinguish K-Maude modules from the usual Maude modules, and thus allow them to be transformed into their corresponding Maude modules, we introduce them by specific keywords. The kinds of K-modules currently supported by K-Maude are syntax, configuration, semantics, and full definition, and are introduced through constructs (k ⟨*keyword*⟩ **for** ⟨*Name*⟩ **is** ... k), where ⟨*keyword*⟩ is repectively **syntax**, **configuration**, **semantics**, or **definition**, and ⟨*Name*⟩ is an identifier for the language, or language feature being defined. The convention is that the syntax/configuration/semantics modules define only syntax/configuration/semantics features, while a definition module contains the entire definition (syntax/configuration/semantics) of a language or a feature in a single module.

WRLA'10, ENTCS (to appear) 2010 / LNCS, Springer 2010, pp 104-122

### 3.3 Language syntax and annotations

*Syntax sorts and operations* When using the syntax modules, all sorts and operations defined there are assumed to be part of the language syntax. However, for definition modules, which mix syntax and semantics, a distinction needs to be made between the signature corresponding to the language syntax and that used for giving semantics. For that, K-Maude introduces two new declarations, **xsort** and **xop**, to identify syntax sorts and operations; all operations of result xsort are assumed language syntax constructs (i.e., xops). K-introduces several new attributes (which can be used in addition to those of Maude):
— *strict* specifies what arguments need to be evaluated before evaluating the termed topped in that language construct. For example, in operation definition '**op** _:=_ : Exp Exp →Exp [**strict**(2)]', the strictness attribute states that the semantic rule for assignment can assume that the second argument is evaluated.
— *seqstrict* is similar, but also states the evaluation *sequence* of the arguments.
— *renameTo* is used to rename an operation, either for avoiding parsing problems, or to desugar some constructs into other K constructs. For example, in the definition '**op** _ _ : Exp Exp →Exp [**strict renameTo** apply]', the rename attribute specifies that _ _ would be renamed to apply for semantics purposes. Similarly, in the definition '**op** {_} : StmtList → Stmt [**renameTo** _]', the rename attribute specifies that the {_} wrapper is no longer needed once the parsing is done, since it was there only for grouping reasons.
— *aux* denotes that a certain language construct would be expressed (through equations) in terms of other language constructs and will not appear in the semantics definition. For example, '**op** if(_)_ : Exp Stmt →Stmt [**aux** prec 47]' uses `aux` to say that although we accept conditionals without the else branch, they would not be handled specifically in the semantics, being reduced to the if-then-else construct through an equation like '**eq** if(E) St = if (E) St else {}'.

### 3.4 Defining the program configuration

*Cell labels* specify the labels used to wrap configuration items. All declared labels must be constants of builtin sort *CellLabel* and must specify what sort of items they wrap by using the K specific **wrapping** attribute. The following declaration of the *env* label, '**op** env : → CellLabel [**wrapping** Map{K,K}]', specifies that the semantics would use a cell identified by *env* holding elements of sort Map{K,K}, i.e., maps from $K$ to $K$; the cell itself has an XML-like syntax, e.g., the cell corresponding to the definition above is <env>_</env>.

The $k$ CellLabel, wrapping computations (of sort $K$), and the $T$ CellLabel, wrapping a soup of configuration items, are very common and thus predefined.
*Configuration structure.* Although not required, specifying the structure of the running configuration of the program is very useful for complex configurations such as the one presented below. Specifying the structure of the configuration serves not only for documenting purposes, but has consequences in both modularity and compactness of definitions, since the semantics rules need to mention only the context required for them to apply, as detailed in next section. The following K declaration, introduced by the K keyword `kconf`, specifies the structure of the configuration for a complex language (shown in Appendix A).

```
kconf <T> <agent*> <thread*> <k> _ </k> <env> _ </env> </thread*>
                   <store> _ </store> <nextLoc> _ </nextLoc>
                   <me> _ </me> <parent> _ </parent>
          </agent*> <nextAgent> _ </nextAgent>
          <messages> <message*> _ </message*> </messages>
      </T>
```

Besides the already mentioned syntax for cells, e.g., <k> _ </k>, the symbol '`*`' appended to the cell label (e.g., for thread) is used to denote that the cell defined by that label can appear multiple times in a program configuration. The configuration term, together with the following cell labels declarations,

```
ops env store : → CellLabel [wrapping Map`{K`,K`}] .
op nextLoc : → CellLabel [wrapping Nat] .
ops agent messages thread : → CellLabel [wrapping Set`{ConfigItem`}] .
op message : → CellLabel [wrapping Tuple] .
ops me parent nextAgent : → CellLabel [wrapping AgentName] .
```

completely define the program configuration to contain agents, a message, and a counter for generating new agent names. Each agent can contain a number of threads, a store map and a counter used to generate new free location names for the store, as well as two cells, `me` and `parent`, containing the name of the current agent, and that of its creator, respectively.

### 3.5 Defining language semantics

K-specific semantics constructs which are supported by K-Maude are K-contexts, K-equations, K-rules, and K-nondeterministic-rules, each being introduced by **kcxt**, **keq**, **krl**, and **knd**, respectively. Additionally, K-equations and K-rules, provide notational shortcuts to make the equations/definitions more compact.

*Context strictness.* K-contexts are usually used for specifying strictness constraints which depend on a context rather than on a single construct. For example, '**kcxt** ∗ K1 := K2 [**strict**(K1)] .' specifies the evaluation to an L-value of a pointer in the assignment construct, allowing the rule for pointer-assignment to assume it has a value in place of K1; the fact that it would have a value instead of K2 as well was specified by the strictness annotation for `:=`.

### 3.6 K equations, rules

K-equations, K-rules, and K nondeterministic rules share the same form, except the first would be "desugared" into equations, while the latter into rules. They basically describe a special term enriched with syntax for expressing K-specific features as: in-place rewriting, cell comprehension, and anonymous variables.

*In-place rewriting.* A K-equation/rule is a term which should contain at least one occurrence of the ⟦ T1 ⟹ T2 ⟧ construct, which is used as a textual representation for the K visual replacement pattern: $\frac{T1}{T2}$. For example, the following K-rule defines how the computation and the environment are encapsulated as a value during a call/cc invocation:

**krl** <k> ⟦`callcc` *V* ⟹`apply`(*V*, cc(*K*, env(*Rho*)))⟧ ↷ *K* </k> <env> *Rho* </env>

*Anonymous variables,* specified by '_' (underscore) can be used to replace all variables whose name is not needed in the match-and-replace process. For example, the rule for applying the current continuation value obtained above,

$$\textbf{krl} <\mathsf{k}> [\![\texttt{apply}(\mathrm{cc}(K,\mathsf{env}(Rho)),V) \Longrightarrow V \curvearrowright K]\!] \curvearrowright _ </\mathsf{k}> <\mathsf{env}> [\![_ \Longrightarrow Rho]\!] </\mathsf{env}>$$

uses anonymous variables to abstract away the remainder of the computation and the old environment, since they would be overwritten.
*Cell list and set comprehension.* Similarly to the visual K notation, Maude-K allows partial specification of the contents of a cell list/set, by attaching elipses '...' to the side of the cell which should be abstracted away. For example, the rule for retrieving the address of a variable from the environment is:

$$\textbf{krl} <\mathsf{k}> [\![\& \ X \Longrightarrow L]\!] \ldots </\mathsf{k}> <\mathsf{env}> \ldots X \mid\rightarrow L \ldots </\mathsf{env}>$$

This rule specifies that if the first task in the computation cell is the request for dereferencing a name, and somewhere in the environment the mapping of the name to a location can be found, then that location should replace the dereferencing expression at the top of the computation, while the rest of the computation and of the environment can be abstracted away by '...'.

*Context abstraction.* The main reason for specifying the structure of the configuration is that one does not need to mention the full context required for the application of a rule, but only the parts which are relevant. A simple instance of using context abstraction is the rule for assigning a value to a name:

$$\textbf{krl} <\mathsf{k}> [\![X := V \Longrightarrow .K]\!] \ldots </\mathsf{k}> <\mathsf{env}> \ldots X \mid\rightarrow L \ldots </\mathsf{env}>$$
$$<\mathsf{store}> \ldots L \mid\rightarrow [\![_ \Longrightarrow V]\!] \ldots </\mathsf{store}> .$$

The rule for assignment should be the same in any definition containing an environment and a store. Although in our definition the store is not at the same level with the computation and the environment, we can still use this rule in the specification, because it can be easily inferred which store the rule refers to.

The following rule expresses randez-vous synchronization:

$$\textbf{krl} <\mathsf{k}> [\![\texttt{rv} \ V \Longrightarrow .K]\!] \ldots </\mathsf{k}> <\mathsf{k}> [\![\texttt{rv} \ V \Longrightarrow .K]\!] \ldots </\mathsf{k}> .$$

Note that, although the two computation cells need to be in two different threads, there is no danger of confusion, since the multiplicity of the `k` cell is one, so the only way to make sense of this rule is to have each computation in its own thread, since the multiplicity of the thread may vary.

The benefits of using context abstraction can be better understood from the following rule, which specifies the asynchronous sending of a value $V$ to an agent identified by name $A$:

$$\textbf{krl} <\mathsf{k}> [\![\mathrm{send{-}asynch} \ \mathrm{A} \ V \Longrightarrow .K]\!] \ldots </\mathsf{k}> <\mathsf{me}> \mathrm{Me} </\mathsf{me}>$$
$$[\![.\mathrm{empty} \Longrightarrow <\mathsf{message}> [\mathrm{Me},\mathrm{A},V] </\mathsf{message}>]\!] .$$

Here all three involved cells are at quite different positions in the configuration structure: the computation cell $k$ is inside a *thread*, which is at the same level as the *me* cell inside an *agent* cell, while the *message* cell is inside a *messages* cell, which is at the same level with the *agent* cell inside the top cell.

WRLA'10, LNCS 2010, pp. 104–122
WRLA'10, Springer (to appear) 2010

*Rewrite rules* One can additionally use rewrite-rules and equations when giving semantics to the language constructs. For example, the following two rewrite rules define the semantics for the conditional; note that, by declaring the conditional strict in the condition, it can be assumed that it is evaluated when giving its semantics: **rl** if (# true) then S1 else _ => S1 .
**rl** if (# false) then _ else S2 => S2 .

## 4 K-Maude Intermediate Representation

The K-Maude interface is written on top of the Full Maude interface to be able to take advantage of the existing work on parsing and interpreting Maude modules. Therefore, we are using the same database of modules defined by Full Maude, and we encode our specific constructs in such a way that they would be recognized as a Maude module. This has two benefits: first, it is easier to work with K-modules as Maude meta-modules; second, having an internal representation which is recognizable as a Maude module makes it possible to write K specifications directly at the core level, albeit using a more unfriendly interface.
*Syntax sorts, operations, and K extra attributes* We encode all attributes which are not native to Maude into the **metadata** string attribute. For example, the core representation of the function application operator defined above is:

**op** __ : Exp Exp →Exp [metadata "**renameTo** `apply` **strict syntax**"] .

Besides the existing **renameTo** apply and **strict** attributes, a new attribute **syntax** appeared in the metadata string. The reason for this is because `Exp` was declared as a syntax sort, introduced with the keyword **xsort**. All operations having the result sort a syntax sort, or those which are introduced by **xop** would be represented in the core representation as normal operations, but with the **syntax** keyword added to the list of attributes in the **metadata**.

A similar encoding is performed for attributes specific to configuration or semantics constructs, e.g., **wrapping**.
*Configurations, K-contexts, K-equations, and K-rules* are all encoded as membership axioms. The following special module is used to facilitate parsing and encoding of the remaining K-specific constructs:

**mod** *K*−RULES **is including** CONFIG .
  **op** ⟦_⟹_⟧ : Universal Universal → Universal [poly (0 1 2) prec 0] .
  **op** AnyVar : → Universal [poly (0)] .
  **ops** keq_ krl_ knd_ kcxt_ kconf_ : Universal →[*K*] [prec 127 poly(1)] .
**endm**

The first declaration defines the in-place replacement as a fully polymorphic operation, thus allowing it to be placed anywhere and to take any arguments, which is crucial for an in-place replacement operator. The second declaration is that for the anonymous variable. Since '_' is reserved by Maude, we have opted to replace it in the internal representation with 'AnyVar'; being anonymous must also be polymorphic. Finally, the last declaration allows to encode each specific K declaration as a membership axiom, while maintaining its identity. For example, the rule for applying a continuation in the core representation is:

WRLA'10, LNCS 2010, pp 104-122
WRLA'10, Springer (to appear) 2010

**mb krl** <k> ⟦apply(cc($K$, env($Rho$)), $V$) $\Longrightarrow V \curvearrowright K$⟧ $\curvearrowright$ AnyVar </k>
<env> ⟦AnyVar $\Longrightarrow Rho$⟧ </env> : $K$ .

What basically happens is that the rule/equation/context is wrapped in a mb_: K . context, which is meaningless from a theoretical point of view, but convenient as a means to represent K constructs as Maude terms.

# 5 From K-Maude to Maude

This section describes the technical part of the K-Maude tool. As the semantics of the K framework itself is given using rewriting logic, it comes natural that the executable semantics of K, as given by the K-Maude tool, is given by reduction to pure Maude (executable) rewriting theories. These transformations are completely defined within Maude, taking advantage of the Maude's reflective capabilities and the predefined Maude (or Full Maude) modules used to represent and transform meta-terms and meta-modules.

## 5.1 From syntax to K syntax and K representation

When specifying the syntax we want to take advantage of the full power of specification, to obtain a syntax as close as possible to the desired language syntax. To do that, and to reduce parsing conflicts, we allow specification features as multiple sorts, mixfix operators, and so on. However, we want to keep computations to a minimal structure to facilitate easy and generic traversal functions, which are crucial for advanced reflective features such as code generation. Nevertheless, while the running structure of the semantics should be as simple as possible, it is still quite desirable to define the semantic rules using constructs as close as possible to the specified language syntax. To achieve this, the K-Maude tool automatically generates two additional syntaxes, an intermediate one, and a running one, from the input (user) syntax.
*The K intermediate syntax* is only used for parsing the semantic rules. Therefore, it bears a close resemblance to the original one by redefining the same operator symbols, but collapsing all their sorts to the computation sort K. For example, the conditional operation described above would now become '**op** if_then_else_ : K K K →K .'. In this step, the **renameTo** attributes are also considered. For example, the generated operation declaration for the function application '__' specified above would be: '**op** apply : K K → K .'.
*The K running syntax* only consists of K labels, as defined in the core syntax of computations presented above. It is automatically derived from the intermediate syntax by associating to each operation symbol a constant of the *KProperLabel* sort. For the two constructs above, their corresponding K label declarations could be '**ops** if~then~else~ apply~~ : → KProperLabel .'.

The *KProperLabel* sort is used because we assume that all of the syntax constructs need evaluation when giving semantics; this plays an important role in addressing strictness, as presented in next section. The K label symbols are generated by replacing the '_' symbols with '~' for the mixfix operators, and by appending as many '~' as the length of the arity for the prefix ones; this seems like a good choice for avoiding label symbol conflicts.

*Handling data types* There are certain sorts, such as integers, booleans, and names, which need to be handled in a special way, to be able to identify them when giving the semantics. Therefore, for all these special categories, identified by prefixing their sort name with the '#' symbol in the syntax, the K syntax is not generated automatically. K-Maude provides a built-in translations for basic data types and for names; for example the integers are embedded in *KResult* (they are already in evaluated form) through the '#' wrapper: '**op** #_ : Int → KResult .', while the names receive their own sort *Name* which is subsorted to *KProper* (names need to be evaluated), and are wrapped by 'name': **op** name : #Name →Name .

*Translating syntax terms to K running terms* is achieved through a polymorphic translation function, '**op** mkK : Universal → [K] [poly (1)] .', whose equational definition is also automatically generated. For example, the equations generated for conditional and function application constructs are:

```
eq mkK(if_then_else_(X0:Exp,X1:Stmt,X2:Stmt))
   = _`(_`)(if~then~else~,_`,_(mkK(X0:Exp),mkK(X1:Stmt),mkK(X2:Stmt))) .
eq mkK(__(X0:Exp,X1:Exp))
   = _`(_`)(apply~~,_`,_(mkK(X0:Exp),mkK(X1:Exp))) .
```

Since these rules are automatically generated, they are generated in prefix form, to avoid potential parsing problems. However, to improve readability, we will use mixfix notation for the remainder of the paper.

*K intermediate syntax to K running syntax* translation is realized by equating K intermediate syntax constructs to their corresponding label application constructs of the K running syntax. The following equations "desugar" the two constructs defined above:

```
eq if X0:K then X1:K else X2:K = if~then~else~(X0:K,X1:K,X2:K) .
eq apply(X0:K,X1:K) = apply~~(X0:K,X1:K) .
```

### 5.2 Strictness

Strictness declarations, either those provided as attributes to operator declaration, or those introduced through specific K contexts constructs, are transformed into pairs of equations which pull the strict arguments out of their contexts for evaluation, and, once they became values, plug them into their original context.

*Strict operator attributes* are transformed as follows: for each position which is declared as strict, a special KLabel declaration is generated and two equations, one for pulling out the *proper* computation for evaluation and the other for plugging in the *result* computation. For the assignment operation declared strict in the first argument, the operation declaration and equations generated are:

```
op ~:=@ : → KLabel .
eq <k> ~:=~(k1:K,k:KProper) ↷KCxt:K</k>
 = <k> k:KProper ↷~:=@(k1:K) ↷KCxt:K</k> .
eq <k> k:KResult ↷~:=@(k1:K) ↷KCxt:K</k> .
 = <k> ~:=~(k1:K,k:KResult) ↷KCxt:K</k>
```

These equations apply only at the top of the continuation, because they should only affect the current evaluation redex. Again, as a way to generate unique and meaningful identifiers, we have chosen to generate the K label "freezing" the remainder arguments by replacing the '~' corresponding to the strict argument with '@', identifying the position of the "hole".

WRLA'10, LNCS 2010 (to appear) / Springer, pp 104-122, 2010

*Strict contexts* receive a similar treatment, but now the generated label used to freeze the remaining computations and to represent the hole would be more complex. For example, the generated K label and equations for the K context defining the evaluation to L-value of a pointer in an assignment, are:

**op** ∗@:=˜ : → KLabel .
**eq** <k> ˜:=˜(∗˜(k:KProper),$K$':$K$) ↷KCxt:$K$ </k>
 = <k> k:KProper ↷∗@:=˜($K$':$K$) ↷KCxt:$K$ </k> .
**eq** <k> k:KResult ↷∗@:=˜($K$':$K$) ↷KCxt:$K$ </k> .
 = <k> ˜:=˜(∗˜(k:KResult),$K$':$K$) ↷KCxt:$K$ </k>

### 5.3 K Semantics

This section describes and exemplifies the process of translating the K semantic constructs to Maude constructs, obtaining an executable definition as a result.

*Applying Context Transformers* Although K-Maude allows the specification to omit the configuration context (for modularity and compactness purposes), this context needs to be filled in by the tool as a first step towards obtaining a runnable definition. To do that, we use the tree associated to the **kconf** declaration and iteratively match the cells having the maximal level in the tree, and to wrap them (if not already wrapped) by their corresponding parent cell in the configuration tree, and then continue. Let us present how the context transformers algorithm works on the examples discussed in Section 3.5.

**krl** <k> ⟦$X$ := $V$ ⟹ .$K$⟧ ...</k> <env>... $X$ |→ $L$ ...</env>
 <store>... $L$ |→ ⟦_ ⟹ $V$⟧ ...</store> .

For the assigment rule, the deepest in the configuration tree are the k and the env cell, which both are subcells of the thread cell. Since the `store` cell, although declared at the same level, corresponds to a higher level in the configuration tree, the first two cells are wrapped by a thread cell:

**krl** <thread>... <k> ⟦$X$ := $V$ ⟹.$K$⟧ ...</k> <env>... $X$ |→ $L$ ...</env>
 ...</thread> <store>... $L$ |→ ⟦_ ⟹ $V$⟧ ...</store> .

The levels of the cells in the new term correspond to their levels in the configuration term; therefore the algorithm concludes succesfully.

**krl** <k> ⟦`rv` $V$ ⟹.$K$⟧ ...</k> <k> ⟦`rv` $V$ ⟹.$K$⟧ ...</k> .

Although the two computations are here at the same level, their multiplicity does not correspond to the one declared in the configuration term. Therefore the context transformers will wrap each of them in their container thread cell:

**krl** <thread>... <k> ⟦`rv` $V$ ⟹.$K$⟧ ...</k> ...</thread>
 <thread>... <k> ⟦`rv` $V$ ⟹.$K$⟧ ...</k> ...</thread> .

Since this cell allows multiple instances in the same cell, the process is complete.

Finally, let us apply the context transformers on the asynchronous send rule:

**krl** <k> ⟦send−asynch A $V$ ⟹.$K$⟧ ...</k> <me> Me </me>
 ⟦.empty ⟹ <message> [Me,A,$V$] </message>⟧ .

First, the k cell is at the lowest level, so it must be wrapped by the thread cell:

<thread>... <k> ⟦send−asynch A $V$ ⟹.$K$⟧ ...</k> ...</thread>
 <me> Me </me> ⟦.empty ⟹<message> [Me,A,$V$] </message>⟧ .

In the next iteration, since all top cells are at the same level, but not having the same parent, they are split up and wrapped accordingly:

```
<agent>... <me> Me </me>
   <thread>... <k> ⟦send−asynch A V ⟹.K⟧ ...</k> ...</thread>
...</agent>
<messages>... ⟦.empty ⟹<message> [Me,A,V] </message>⟧
...</messages>
```

This term is correct according to the configuration, thus the process is complete.
*Resolving cell comprehension and anonymous variables.* Once the context transformers have been applied (taking advantage of the cell comprehension feature), the next step towards obtaining a standard rewriting theory is to resolve cell comprehension and anonymous variables by replacing them with variables of the right sort. To do that, the K definition is traversed, and each term is recursively visited. The visitor uses the information specified as the **wrapping** attribute in the declaration of the cell labels to infer the constructor and the variables needed to replace the ellipses, and it uses the full signature to resolve the anonymous variables. For example, the context-transformed version of the assignment rule above will look as follows after this step:

```
krl <k> ⟦X := V ⟹ .K⟧ ↷ Rest:K </k>
    <env> X |→ L &' Rest:Map{K,K} </env>
    <store> L |→ ⟦...:K ⟹ V⟧ &' Rest':Map{K,K} </store> .
```

Note that although set comprehension uses ellipses on both sides of the cell, we only need one variable, since the constructor is associative and commutative. The ellipses are replaced by the corresponding constructor and a variable whose name starts with '`Rest`' and which can be "primed" several times for disambiguation. Anonymous variables use '...' as their variable name, again primed if necessary. At the completion of this step, the asynchronous send rule would look as follows:

```
krl <agent> Rest:Set{ConfigItem} <me> Me </me>
      <thread> Rest':Set{ConfigItem}
         <k> ⟦send−asynch A V ⟹.K⟧ ↷Rest:K </k> </thread>
    </agent>
    <messages> Rest'':Set{ConfigItem}
      ⟦.empty ⟹ <message> [Me,A,V] </message>⟧ </messages>
```

*Transforming K-equations and rules into rewrite equations and rules* becomes relatively simple upon the completion of the steps above. Mathematically speaking, from each K rule/equation term C[ ⟦l1 ⟹ r1⟧, ..., ⟦ln ⟹ rn⟧ ], two terms of the corresponding rewrite rule ($l \Rightarrow r$) or equation ($l = r$), can be inferred as being $l = C[l1, \ldots, ln]$, and $r = C[r1, \ldots, rn]$. This inference process is defined by building the two terms $l$ and $r$ together while traversing the K rule/equations. At the completion of this step, the asynchronous send rule is:

```
rl <agent> Rest:Set{ConfigItem} <me> Me </me>
      <thread> Rest':Set{ConfigItem}
         <k> send−asynch A V ↷Rest:K </k> </thread>
   </agent>
   <messages> Rest'':Set{ConfigItem} .empty </messages>
=> <agent> Rest:Set{ConfigItem} <me> Me </me>
      <thread> Rest':Set{ConfigItem}
```

```
      <k> .K ↷ Rest:K </k> </thread>
  </agent>
  <messages> Rest'':Set{ConfigItem} <message> [Me,A,V] </message>
  </messages>
```

*Reduction to the K running syntax.* Although the previous step produces rules/equations which would be accepted by Maude with the K intermediate and running syntaxes combined, for executability reasons we need to make sure all rules are in the K running syntax. This is achieved by (meta-)reducing each semantic rule/equation to its normal form w.r.t. the equations transforming K intermediate to K running syntax, as well as to the strictness equations. Doing so ensures that all rules/equations would act on the running syntax of K, and thus the consistency of their interaction among themselves. Additionally, this step reduces the compositions of constructors with their identities (due to the use of · in rules) which were introduced at the previous step. The final running version of the asynchronous send rule would thus be:

```
rl <agent> Rest:Set{ConfigItem} <me> Me </me>
     <thread> Rest':Set{ConfigItem}
        <k> send−asynch~~(A, V) ↷Rest:K </k> </thread>
   </agent>
   <messages> Rest'':Set{ConfigItem} </messages>
=> <agent> Rest:Set{ConfigItem} <me> Me </me>
     <thread> Rest':Set{ConfigItem} <k> Rest:K </k> </thread>
   </agent>
   <messages> Rest'':Set{ConfigItem} <message> [Me,A,V] </message>
   </messages>
```

## 6 From K-Maude to LaTeX

Although the K-Maude textual interface is itself a good communication medium, K was primarily introduced through a very intuitive and visual notation, more appropriate for programming languages design. Therefore, to facilitate the inclusion of (parts of) the K definitions in research papers and presentations, K-Maude allows for annotations (as special attributes) specifying how various constructs should be represented in LaTeX, and provides a tool (written in Maude, as well) which automatically generates a LaTeXdocument from a provided K-Maude definition. The LaTeX-specific attributes (currently only **renameTo** and **color** are wrapped in an attribute wrapper, **latex**. For example, the following definition of the lambda abstraction renames '`lambda`' to '$\lambda$' for LaTeXpurposes:

```
op lambda_._ : #Name Exp →Exp [latex(renameTo \ensuremath`{\lambda`}_._)].
```

Besides renaming constructs, one can additionally associate color codes to the K cells, to make them more easily distinguishable in presentations, as in the following example, used for coloring the agent cell in red:

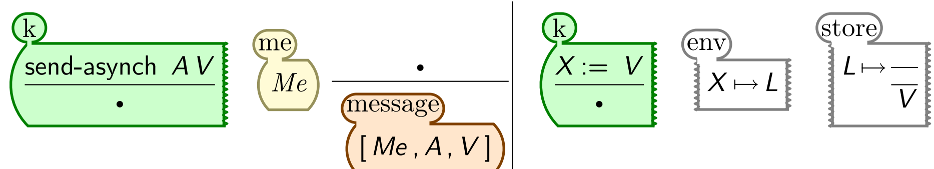
```
ops agent : → CellLabel [wrapping Set`{ConfigItem`} latex(color: red)] .
```

*Formatted output.* Sort, subsort, and operation declarations are converted to their equivalent BNF notation, since this notation is prevalent in programming languages definitions. For example, the following declarations

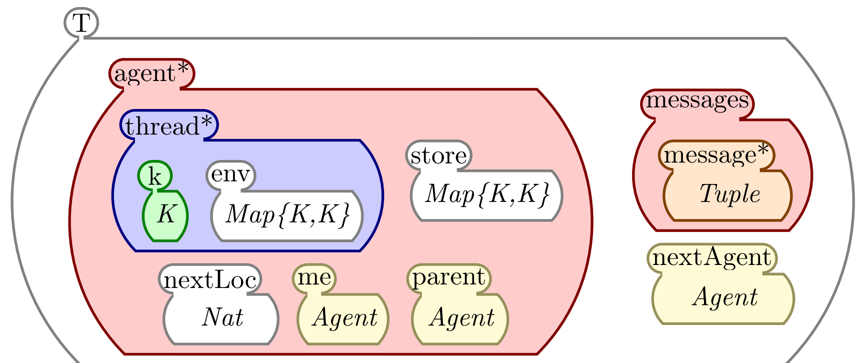
**xsort** Exp .
**subsort** #Int #Name < Exp .
**op** _+_ : Exp Exp →Exp [gather(E *e*) prec 33 **strict**] .
**op** lambda_._ : #Name Exp →Exp [**latex**(**renameTo** \ensuremath‘{\lambda‘}_._)].

are automatically typesetted to: $\mathit{Exp} ::= \#\mathit{Int} | \#\mathit{Name}$
$| \ \mathit{Exp} * \ \mathit{Exp}$ [strict]
$| \ \lambda \, \#\mathit{Name} \, . \ \mathit{Exp}$

K cells are represented using the `tikz` package as rectangles with rounded sides and with the cell label attached to the top. Completely specified cells have both sides rounded. Cells specified with ellipses on either side have the corresponding side “ripped”. For example, the rules for asynchronous message sending and variable assignment discussed above would be typesetted as, respectively:




The configuration is represented by combining the configuration term with the **wrapping** attributes of cell labels declarations, which are used to fill the _ places. For example, the configuration specified above can be typesetted to:




## 7 Conclusions

We described K-Maude, an implementation of the K language definitional framework in Maude. K-Maude consists of an interface, which allows one to define a language using both special K modules and Maude modules. The K modules are translated into intermediate representation Maude modules, which can be further translated into either executable Maude modules to obtain interpreters or into Latex to obtain formal language semantics documentation.

## References

1. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
2. M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
3. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
4. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
5. José Meseguer and Grigore Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
6. Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.
7. Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.
8. Grigore Roşu. K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign, 2006. A previous version of this work has been published as technical report UIUCDCS-R-2005-2672 in 2005. K was first introduced in 2003, in the technical report UIUCDCS-R-2003-2897: lecture notes of CS322 (programming language design).
9. Traian Florin Serbanuta, Grigore Rosu, and Jose Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 2007. to appear.
10. Christopher Strachey and Christopher P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symb. Computation*, 13(1/2):135–152, 2000.
11. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

The appendix is only provided for reviewer's convenience.

## A Annotated definition of the K-CHALLENGE language

The definition presented below was used to compare the features of the K framework with those of other language definitional frameworks. It is completely written using the K-Maude tool and is effectively an interpreter on top of Maude for the given language. The definition was developed iteratively, starting from a minimalist language, and iteratively adding new features. All new features were appended, together with the induced replacements for existing rules (if any).

We present below the full definition, fragmented with explanations of the various constructs used. To make this appendix self-contained, parts of the explanations regarding the K-Maude interface might be repeated.

```
(k definition for KCHALLENGE is
 including INT−K−SYNTAX + BOOL−K−SYNTAX + FLOAT−K−SYNTAX .
 including NAME−K−SYNTAX .
 including COMMON−ENV−STORE .
 including BINDER−K−SYNTAX .
 including KMAP{K,K} + KSET{K} .
 including TUPLE .

 var VL : List{KResult} . var N : Nat .
 vars I I1 I2 : Int . var B : Bool . var X : Name . var V V' : KResult .
 vars F1 F2 : Float . var L L' : Loc .
 var E BE S S1 S2 K K' : K . var Sigma : Store . var Rho Rho' : Env{Loc} .

 xsort Exp .
```

'xsort' is used to specify a syntax sort – all operations having it as a result sort would be considered part of syntax

```
 subsort #Int #Bool #Float < Exp .
```

By convention, '#' sorts are used for input literals, while their corresponding semantics sorts are named by removing the '#' . Int, Bool, and Float, are imported from built-in modules INT-K-SYNTAX, BOOL-K-SYNTAX, and FLOAT-K-SYNTAX, respectively.

```
 subsort #Name < Exp .
 krl <k> [[X ==> V]] ...</k> <env>... X |-> L ...</env> <store>... L |-> V ...</store> .
```

Names are imported from predefined module NAME-K-SYNTAX. 'env' and 'store' cells as well as basic operations on them are imported from COMMON-ENV-STORE . When a name is matched at the top of computation, it can replaced by value V, provided that the mapping of X to location L can be matched in the environment and the mapping of L to V can be matched in the store.

```
 op _+_ : Exp Exp ->Exp [gather(E e) prec 33 strict ] .
 rl # I1 + # I2 => # (I1 + I2) .
 rl # F1 + # F2 => # (F1 + F2) .
```

'gather' and 'prec' are standard attributes in Maude, used for parsing. Additionally 'strict' specifies that both operands of '+' should be evaluated (order not specified) before '+' being evaluated itself. This allows us to only give semantics of '+' on valued terms. Operation '#' is used to distinguish values when giving semantics.

```
op _*_ : Exp Exp → Exp [gather(E e) prec 31 strict ] .
rl _*_(# I1, # I2) => # (I1 * I2) .
rl _*_(# F1, # F2) => # (F1 * F2) .

op _<=_ : Exp Exp →Exp [prec 37 seqstrict ] .
rl # I1 <= # I2 => # (I1 <= I2) .
rl # F1 <= # F2 => # (F1 <= F2) .
```

'seqstrict' is specified when the order of evaluation matters. The default order is left-to-right.

```
op not_ : Exp → Exp [prec 53 strict ] .
rl not (# B) => # (not B) .

op _and_ : Exp Exp →Exp [gather(E e) prec 55 strict (1)] .
rl (# true) and BE => BE .
rl (# false) and _ => # false .
```

The semantics of 'and' given above is "shortcut". For this, we only require the first operand to be evaluated—'strict(1)'—, and evaluate the expression according to that value. Note that one can use anonymous variable '_' if the name of a variable is only required for matching.

```
xsort Stmt .
op _;_ : Stmt Stmt → Stmt [prec 100 gather(e E) renameTo _↷_] .
```

By renaming the statement sequencing operator to '_↷_', the K sequencing operator, we don't need to specify any additional semantic rules for it.

```
op var_ : #Name →Stmt .
krl <k> ⟦var X ⟹.K⟧ ...</k>
    <env> ⟦Rho ⟹Rho[X ←L]⟧ </env>
    <nextLoc> ⟦L ⟹L + 1⟧ </nextLoc> .
```

Variable declaration: we use the 'nextLoc' cell imported from COMMON-ENV-STORE. also, we use `Rho[X <- L]` to map X to L in Rho.

```
op if_then_else_ : Exp Stmt Stmt → Stmt [ strict (1)] .
rl if (# true) then S1 else _ => S1 .
rl if (# false) then _ else S2 => S2 .
```

Before giving semantics for conditional, we only need to evaluate the condition, specified by the 'strict(1)' annotation.

```
op while_do_ : Exp Stmt → Stmt .
keq <k> ⟦while BE do S ⟹if BE then (S → while BE do S) else .K⟧ ...</k> .
```

The semantics of 'while' is given by unrolling, but only at the top of the computation, to avoid cycling.

```
op _;_ : Stmt Exp → Exp [prec 110 renameTo _→_] .

op output_ : Exp → Stmt [ strict ] .
op output : → CellLabel [wrapping List‘{KResult‘}] .
krl <k> ⟦output V ⟹ . K⟧ ...</k> <output>... ⟦.nil ⟹ V⟧ </output> .
```

We declare a new cell—output—to hold the list of output values.

```
op ++_ : #Name →Exp [prec 0 renameTo inc] .
krl <k> ⟦inc(X) ⟹# (I + 1)⟧ ...</k> <env>... X |→ L ...</env>
    <store>... L |→ # ⟦I ⟹ I + 1⟧ ...</store> .
```

Sometime we need to rename constructs when going from syntax to semantics to avoid name clashing. In this case, we rename '++' to 'inc'.

```
op #_ : Loc → KResult .
op &_ : #Name →Exp [prec 0] .
krl <k> ⟦& X ⟹# L⟧ ...</k>
    <env>... X |→ L ...</env> .
```

Obtaining the reference location of a name.

```
op *_ : Exp → Exp [ strict prec 25] .
krl <k> ⟦* # L ⟹V⟧ ...</k> <store>... L |→ V ...</store> .
```

Dereferencing a location.

```
op ref_ : Exp → Exp [ strict prec 0] .
krl <k> ⟦ref V ⟹L⟧ ...</k>
    <store> ⟦Sigma ⟹ Sigma[L ← V]⟧ </store>
    <nextLoc> ⟦L ⟹L + 1⟧ </nextLoc> .
```

Evaluating an expression and returning a reference to it.

```
op _:=_ : Exp Exp →Stmt [ strict (2)] .
krl <k> ⟦X := V ⟹.K⟧ ...</k>
    <env>... X |→ L ...</env>
    <store> ⟦Sigma ⟹ Sigma[L ← V]⟧ </store> .
kcxt * K := K' [ strict (K)] .
krl <k> ⟦* # L := V ⟹.K⟧ ...</k>
    <store> ⟦Sigma ⟹ Sigma[L ← V]⟧ </store> .
```

The right hand side is evaluated to a value, specified here by 'strict(2)' in the operator declaration, while the left-hand-side is evaluated to a l-value, here either a Name, or the dereferencing of a location (the 'strict(K)' in the K-context declaration).

```
op aspect_ : Stmt → Stmt .
op aspect : → CellLabel [wrapping K] .
krl <k> ⟦aspect S ⟹ . K⟧ ...</k> <aspect> ⟦_ ⟹S⟧ </aspect> .
keq <k> ⟦lambda X . E ⟹ closure (X, S → E, Rho)⟧ ...</k>
    <env> Rho </env> <aspect> S </aspect> .
```

Aspects are defined to be executed before the start of each function evaluation, in the environment in which that function was defined. Note that lambda abstractions are evaluated to closures.

```
op __ : Exp Exp → Exp [ strict  renameTo apply] .
op closure : Name K Env{Loc} →KResult .
krl <k> ⟦apply(closure(X, E, Rho), V) ⟹ E ↷ restore (Rho')⟧ ...</k>
    <env> ⟦Rho' ⟹Rho[X ← L]⟧ </env>
    <store> ⟦Sigma ⟹ Sigma[L ← V]⟧ </store>
    <nextLoc> ⟦L ⟹L + 1⟧ </nextLoc> .
```

Application of a function is declared ‘strict’ (call-by-value), renamed to ‘apply’ to avoid name clashes. Note that we use ‘restore’ (imported from COMMON-ENV-STORE) to restore the environment once the function was evaluated.

```
krl <k> ⟦mu X . E ⟹ E ↷ restore (Rho)⟧ ...</k>
    <env> ⟦Rho ⟹Rho[X ← L]⟧ </env>
    <store> ⟦Sigma ⟹ Sigma[L ← mu X . E]⟧ </store>
    <nextLoc> ⟦L ⟹L + 1⟧ </nextLoc> .
```

The semantics of $\mu X.E$ is given by evaluating the expression $E$ in the environment/-store where $X$ is bound to a $\mu X.E$.

```
op callcc_ : Exp → Exp [ strict ] .
op cc : K Env{Loc} → KResult .
 krl <k> ⟦callcc V ⟹apply(V, cc(K, Rho))⟧ ↷ K </k>
      <env> Rho </env> .
 krl <k> ⟦apply(cc(K, Rho), V) ↷ _ ⟹ V ↷ K⟧ </k>
       <env> ⟦_ ⟹Rho⟧ </env> .

op randomBool : → Exp .
knd <k> ⟦randomBool ⟹# true⟧ ...</k> .
knd <k> ⟦randomBool ⟹# false⟧ ...</k> .
```

‘randomBool’ non−deterministically evaluates to either ‘true’ or ‘false’.

```
op spawn_ : Stmt → Stmt .
op holds : → CellLabel [wrapping Map‘{K‘,K‘}] .
op thread : → CellLabel [wrapping Set‘{ConfigItem‘}] .
var Holds : Map‘{K‘,K‘} .
krl <thread>... <k> ⟦spawn S ⟹.K⟧ ...</k> <env> Rho </env> ...</thread>
    ⟦.empty ⟹ .empty <thread> <k> S </k> <env> Rho </env>
                     <holds> .empty </holds> </thread>⟧ .
```

Spawning a new thread: the ‘thread’ cell is used to group together all cell related to a thread; ‘holds’ holds the locks acquired by the thread (and their multiplicity).

```
op busy : → CellLabel [wrapping Set‘{K‘}] .
var Busy : Set‘{K‘} .
krl ⟦<thread>... <k> .K </k> <holds> Holds </holds> ...</thread>
   ⟹ .empty⟧ <busy> ⟦Busy ⟹Busy −' keys(Holds)⟧ </busy> .
```

When the computation of a thread has completed, it can be dissolved and its resources released. ‘busy’ holds the names of the locks acquired by any of the threads.

```
op acquire_ : Exp → Stmt [ strict ] .
krl <k>⟦acquire V ⟹.K⟧ ...</k>
    <holds>... V |→ # ⟦N ⟹ s(N)⟧ ...</holds> .
```

```
kcrl <k>⟦acquire V ⟹.K⟧ ...</k>
     <holds>... ⟦.empty ⟹ V |→ # 0⟧ ...</holds>
     <busy> Busy ⟦.empty ⟹V⟧</busy> if not(V in' Busy) .
```

One can lock on any value. A lock can only be acquired it it is not busy. Same thread can acquire same lock multiple times, therefore we keep multiplicities for each lock in the 'holds' cell.

```
op release_ : Exp → Stmt [ strict ] .
krl <k>⟦release V ⟹.K⟧ ...</k>
    <holds>... V |→ # ⟦s(N) ⟹ N⟧ ...</holds> .
krl <k>⟦release V ⟹.K⟧ ...</k>
    <holds>... ⟦V |→ # 0 ⟹ .empty⟧ ...</holds>
    <busy>... ⟦V ⟹ .empty⟧ ...</busy> .

op rv_ : Exp → Stmt [ strict ] .
krl <k> ⟦rv V ⟹.K⟧ ...</k> <k> ⟦rv V ⟹.K⟧ ...</k> .
```

Rendez-vous synchronization. Two threads can only pass together a barrier specified by a lock V.

```
sort Agent .
op agent : Nat → Agent .
subsort Agent < KResult .
op new−agent_ : Stmt →Exp .
op agent : → CellLabel [wrapping Set`{ConfigItem`}] .
ops me parent nextAgent : → CellLabel [wrapping Agent] .
krl <k> ⟦new−agent S ⟹agent(N)⟧ ...</k> <me> Me </me>
    <nextAgent> agent(⟦N ⟹N + 1⟧) </nextAgent>
    ⟦.empty ⟹ <agent>
                 <thread>
                  <k> S </k> <env> .empty </env> <holds> .empty </holds>
                 </thread>
                 <store> .empty </store> <nextLoc> loc(0) </nextLoc>
                 <aspect> .K </aspect> <busy> .empty </busy>
                 <me> agent(N) </me> <parent> Me </parent>
               </agent>⟧ .
```

An agent is here a collection of threads, grouped in an 'agent' cell identified by an id held by the 'me' cell, and holding in the 'parent' cell a reference id to its creating agent. 'nextAgent' is used for providing fresh ids for agents.

```
rl <agent>
      <store> _ </store>
      <nextLoc> _ </nextLoc>
      <aspect> _ </aspect>
      <busy> _ </busy>
      <me> _ </me>
      <parent> _ </parent>
   </agent> => .empty .
```

When all threads inside an agent have completed, the agent can be dissolved.

```
op me : → Exp .
krl <k>⟦me ⟹A⟧ ...</k> <me> A </me> .

op parent : → Exp .
krl <k>⟦parent ⟹ A⟧ ...</k> <parent> A </parent> .

op message : → CellLabel [wrapping Tuple] .
vars Me Parent A : Agent .
op send−asynch_ _ : Exp Exp →Stmt [strict] .
krl <k> ⟦send−asynch A V ⟹.K⟧ ...</k> <me> Me </me>
    ⟦.empty ⟹ <message> [Me,A,V] </message>⟧ .
```

An agent can send any value (including agents ids) to other agents (provided it knows their id). To model asynchronous communication, each value sent is wrapped in a 'message' cell identifying both the sender and the intended receiver.

```
op receive−from_ : Exp →Exp [strict] .
 krl <k> ⟦receive−from A ⟹V⟧ ...</k> <me> Me </me>
     ⟦<message> [A,Me,V] </message> ⟹.empty⟧ .
```

An agent can request to receive a message from a certain agent.

```
ops receive : → Exp .
 krl <k> ⟦receive ⟹V⟧ ...</k> <me> Me </me>
     ⟦<message> [_,Me,V] </message> ⟹.empty⟧ .
```

An agent can request to receive a message from any agent.

```
op send−synch_ _ : Exp Exp →Stmt [strict] .
 krl <agent>... <k> ⟦send−synch A V ⟹.K⟧ ...</k> <me> Me </me> ...</agent>
  <agent>... <k> ⟦receive−from Me ⟹V⟧ ...</k> <me> A </me> ...</agent> .
 krl <agent>... <k> ⟦send−synch A V ⟹.K⟧ ...</k> ...</agent>
  <agent>... <k> ⟦receive ⟹V⟧ ...</k> <me> A </me> ...</agent> .
```

The message can be sent synchronously, in which case, two agents need to matched together for the exchange to occur.

```
op halt : → Stmt .
krl ⟦<agent>... <k> halt ...</k> ...</agent>
⟹ .empty⟧ .
```

The semantics of halt in one of the thread of an agent is that is dissolves the agent.

```
op messages : → CellLabel [wrapping Set‘{ConfigItem‘}] .
op result : → CellLabel [wrapping List‘{KResult‘}] .
kconf
  <T>
    <agent∗>
       <thread∗>
          <k> _ </k>
          <env> _ </env>
          <holds> _ </holds>
       </thread∗>
       <store> _ </store>
```

```
        <nextLoc> _ </nextLoc>
        <aspect> _ </aspect>
        <busy> _ </busy>
        <me> _ </me>
        <parent> _ </parent>
    </agent*>
    <output> _ </output>
    <messages> <message*> _ </message*> </messages>
    <nextAgent> _ </nextAgent>
  </T> <result> _ </result> .
```

For K to know where each cell is located, one needs to specify (if there are cells at different levels) the structure of the configuration, together whit an indication of which of the cells have multiplicities. Notice here that we have a wrapper for messages which was not specified anywhere, as well as a wrapper for the results which is at the top level.

```
op '['[ _ ']'] : Stmt → Config .
eq ⟦P:Stmt⟧
 = <T>
     <agent>
       <thread>
          <k> mkK(P:Stmt) </k>
          <env> .empty </env>
          <holds> .empty </holds>
       </thread>
       <store> .empty </store>
       <nextLoc> loc(0) </nextLoc>
       <aspect> .K </aspect>
       <busy> .empty </busy>
       <me> agent(0) </me>
       <parent> agent(0) </parent>
     </agent>
     <output> .nil </output>
     <messages> .empty </messages>
     <nextAgent> agent(1) </nextAgent>
   </T> .
```

This is how a program is initialized to be executed using the above definition.

```
krl ⟦<T> <output> VL </output> <nextAgent> _ </nextAgent>
      <messages> _ </messages> </T>
  ⟹ <result> VL </result> ⟧.
```

When there are no more agents executing, we can collect the output and transfer it into the result cell.

Advanced features: code generation

```
op quote_ : Exp → Exp .
op unquote_ : Exp → Exp .
op eval_ : Exp → Exp [ strict ] .
```

```
op quote : Nat List{K} → KProper .
op code : List{K} → KResult .
op _box‘(→‘)_ : K K → KProper [strict] .
op _box‘(‘,‘)_ : K K → KProper [strict] .

op box : K → KSynLabel .
op kl : KLabel → K .
var KL : KLabel . var KLK : K . var Ks : NeList‘{K‘} .
var K1 K2 : NeK .
kcxt box(KLK)(Ks) [strict(Ks)] .

keq <k> ⟦quote K ⟹quote(0,K)⟧ ...</k> .
eq quote(N, K1 → K2) = quote(N, K1) box(→) quote(N,K2) .
eq code(K1) box(→) code(K2) = code(K1 →K2) .
ceq quote(N, KL(Ks)) = box(kl(KL))(quote(N, Ks))
 if KL =/= quote~ /\ KL =/= unquote~ .
eq box(kl(KL))(code(Ks)) = code(KL(Ks)) .
eq quote(N, quote(K)) = box(kl(quote~))(quote(s(N), K)) .
eq quote(0, unquote(K)) = K .
eq quote(s(N), unquote(K)) = box(kl(unquote~))(quote(N,K)) .

eq quote(N, (K, Ks)) = quote(N, K) box(,) quote(N, Ks) .
eq code(K) box(,) code(Ks) = code((K, Ks)) .

eq quote(N, V) = code(V) .
eq quote(N,X) = code(X) .

eq eval code(K) = K .

k)
```

The end of our K definition module is specified by ‘k)’.

## B KCHALLENGE in LaTeX

MODULE KCHALLENGE

IMPORTS BOOL-K-SYNTAX+FLOAT-K-SYNTAX+INT-K-SYNTAX
IMPORTS NAME-K-SYNTAX
IMPORTS KMAP{K,K}
IMPORTS TUPLE
IMPORTS KCHALLENGE-CONFIGURATION
SYNTACTIC CONSTRUCTS:

*Exp* ::= *#Bool*|*#Float*|*#Int*|*#Name*
| me
| parent
| randomBool
| receive
| * *Exp* [strict]
| ++ *#Name* [renameTo inc]
| callcc *Exp* [strict]
| eval *Exp* [strict]
| new-agent *Stmt*
| not *Exp* [strict]
| quote *Exp*
| receive-from *Exp* [strict]
| ref *Exp* [strict]
| unquote *Exp*
| &( *#Name* )
| *Exp* *Exp* [renameTo apply strict]
| *Exp* * *Exp* [strict]
| *Exp* + *Exp* [strict]
| *Exp* and *Exp* [strict(1)]
| *Stmt* ; *Exp* [renameTo _ $\curvearrowright$ _]
| *Exp* $\leq$ *Exp* [seqstrict]
| $\lambda$ *#Name* . *Exp*
| $\mu$ *#Name* . *Exp*

*Stmt* ::=
| halt
| acquire *Exp* [strict]
| aspect *Stmt*
| output *Exp* [strict]
| release *Exp* [strict]
| rv *Exp* [strict]
| spawn *Stmt*
| var *#Name*
| *Exp* := *Exp* [strict(2)]
| *Stmt* ; *Stmt* [renameTo _ $\curvearrowright$ _]
| send-asynch *Exp* *Exp* [strict]
| send-synch *Exp* *Exp* [strict]

| while *Exp* do *Stmt*
| if *Exp* then *Stmt* else *Stmt* [strict(1)]

SEMANTIC CONSTRUCTS:

*Agent* ::=
| agent( *Nat* )

*Config* ::=
| ⟦ *Stmt* ⟧

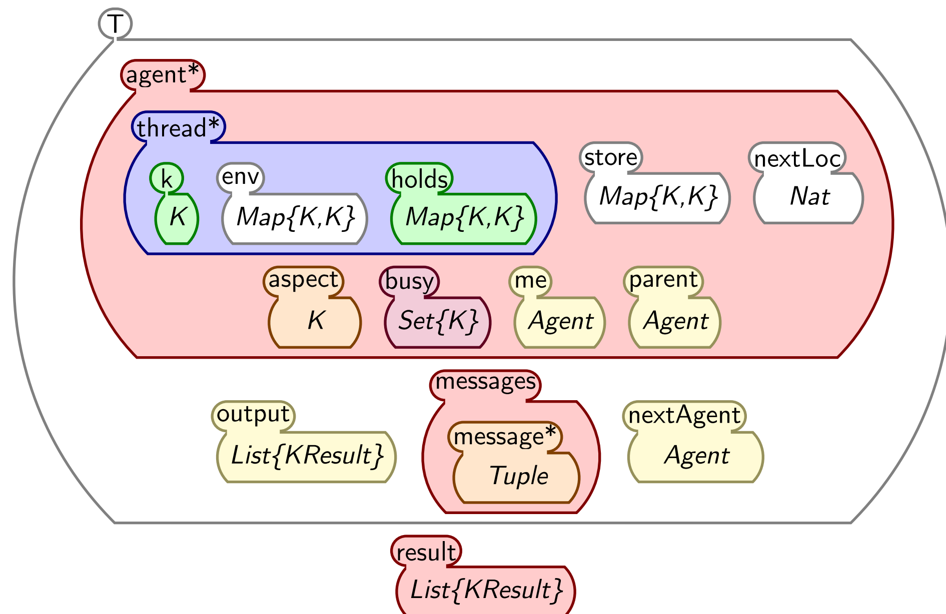
*KProper* ::=
| *K* [ , ] *K* [strict]
| *K* [ ⌒ ] *K* [strict]
| quote( *Nat* , *List{K}* )

*KProperLabel* ::=
| [ ~ ]
| [ *K* ]

*KResultLabel* ::=
| cc
| closure
| code

*NeK* ::=
| *KLabel*
| *Map{K,K}*

CONFIGURATION:




K EQUATIONS AND RULES:

CONTEXT: * *K* := *K′* [strict(K)]

CONTEXT: [ ~ ]( *Kl* ) [strict(Kl)]

CONTEXT: *KLabel* ( *Ks* ) [strict(Ks)]

EQUATION:

k: $\dfrac{\text{quote } K}{\text{quote}(\,0\,,K\,)}$

EQUATION:

k: $\dfrac{\text{while } BE \text{ do } S}{\text{if } BE \text{ then } S \curvearrowright \text{while } BE \text{ do } S \text{ else } \bullet}$

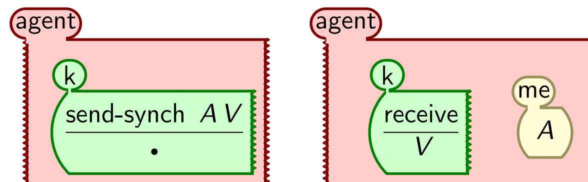
EQUATION:

k: $V \curvearrowright \dfrac{\rho}{\bullet}$ env: $\dfrac{\varrho}{\rho}$

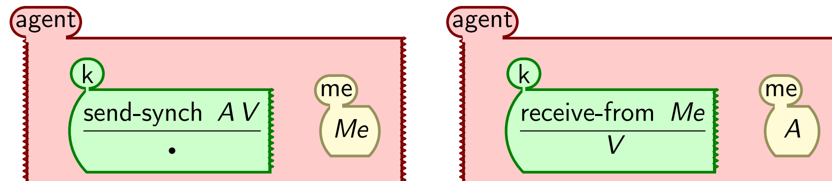
EQUATION:

k: $\dfrac{\lambda X\,.\,E}{\text{closure}\,(\,X\,,S \curvearrowright E\,,\,\rho\,)}$ env: $\rho$ aspect: $S$

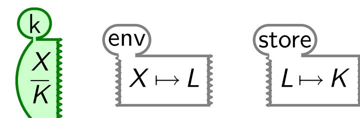
RULE:



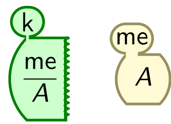

RULE:



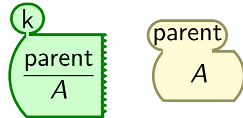

RULE:



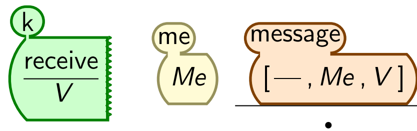

RULE:



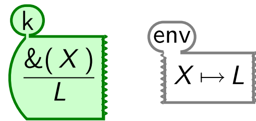

RULE:



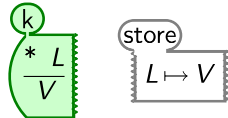

RULE:



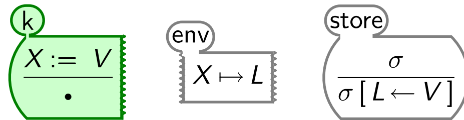

RULE:



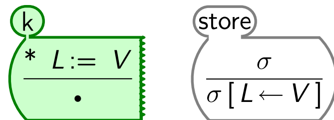

RULE:



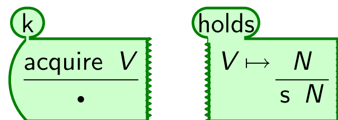

RULE:



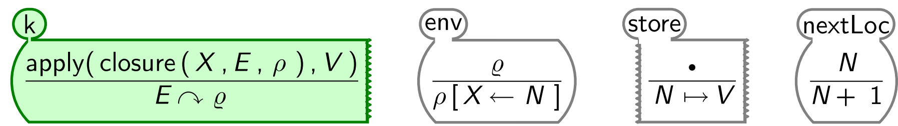

RULE:



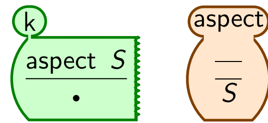

RULE:



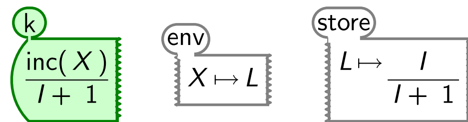

RULE:



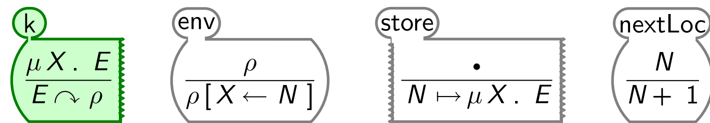

RULE:



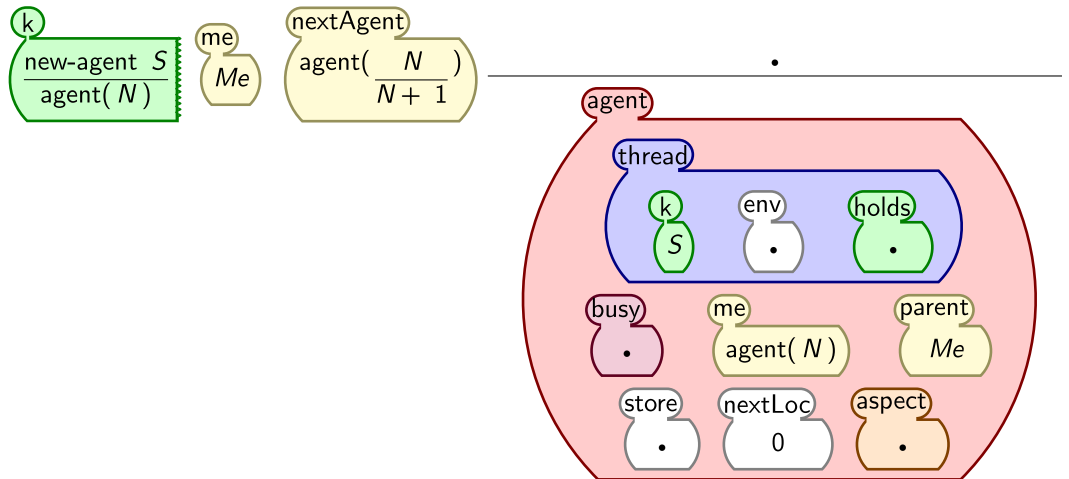

RULE:



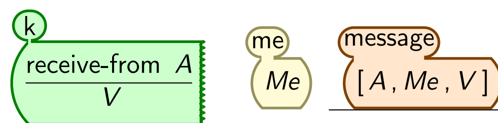

RULE:



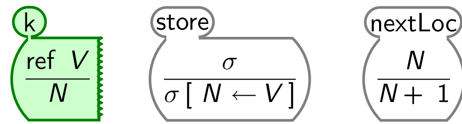

RULE:



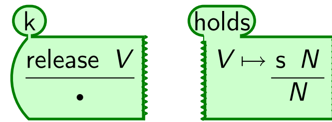

RULE:



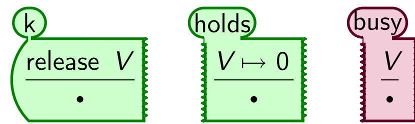

RULE:



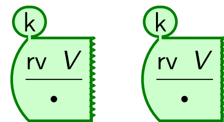

RULE:



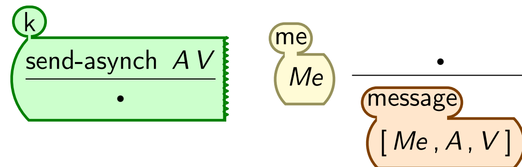

RULE:



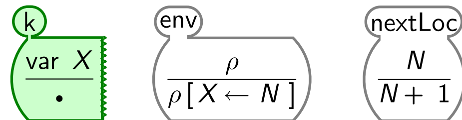

RULE:



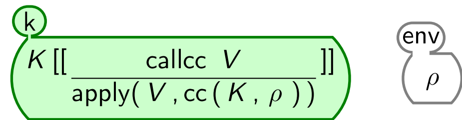

RULE:



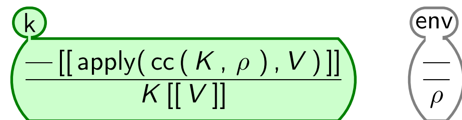

RULE:



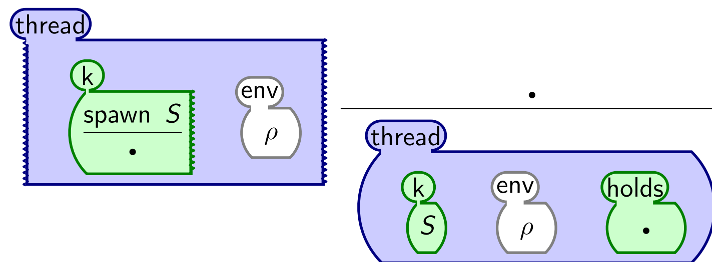

RULE:



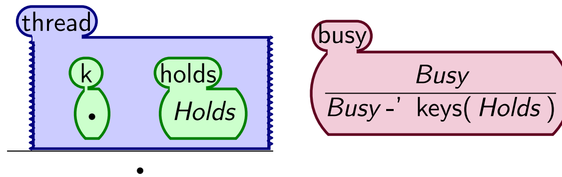

RULE:



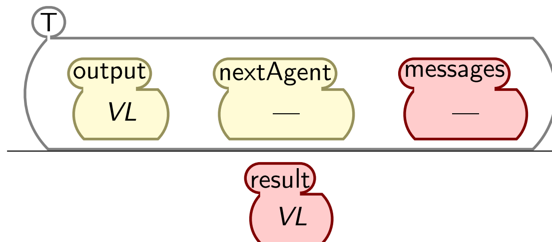

RULE:



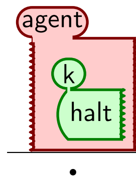

RULE:



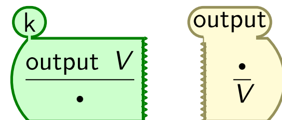

RULE:



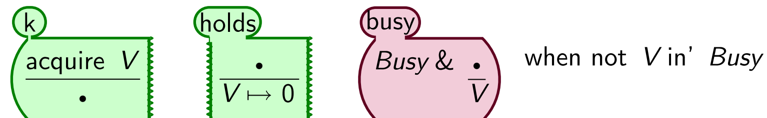

RULE:



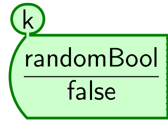

OUTPUT RULE:



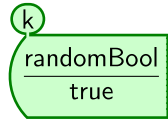

RULE:




NON-DET RULE:




NON-DET RULE:




EQUATION: quote( $N$ , $Label$ ( $Ks$ ) ) = $\boxed{Label}$ ( quote( $N$ , $Ks$ ) ) when $Label$ =/= quote and $Label$ =/= unquote

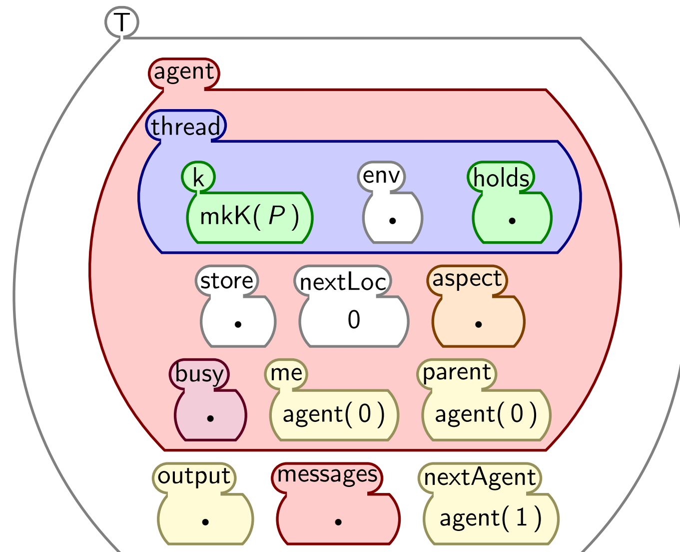
EQUATION: $\boxed{\sim}$ ( code ( $Kl$ ) ) = code ( ~( $Kl$ ) )

EQUATION: $\boxed{Label}$ ( code ( $Ks$ ) ) = code ( $Label$ ( $Ks$ ) )

EQUATION: code ( $K_1$ ) $\boxed{\curvearrowright}$ code ( $K_2$ ) = code ( $K_1 \curvearrowright K_2$ )

EQUATION: code ( $K$ ) $\boxed{,}$ code ( $Ks$ ) = code ( $K$ , $Ks$ )

EQUATION: $[\![ P ]\!]$ =




EQUATION: eval code ( $K$ ) = $K$

EQUATION: quote( 0 , unquote $K$ ) = $K$

EQUATION: quote( $N$ , • ) = code ( • )

EQUATION: quote( $N$ , $V$ ) = code ( $V$ )

EQUATION: quote( $N$ , $X$ ) = code ( $X$ )

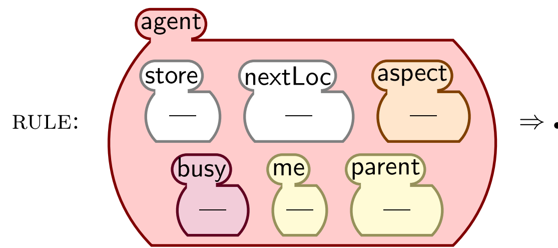
EQUATION: quote( $N$ , $K_1 \curvearrowright K_2$ ) = quote( $N$ , $K_1$ ) $\boxed{\curvearrowright}$ quote( $N$ , $K_2$ )

EQUATION: quote( $N$ , $K$ , $Ks$ ) = quote( $N$ , $K$ ) $\boxed{,}$ quote( $N$ , $Ks$ )

EQUATION: quote( $N$ , quote $K$ ) = $\boxed{\text{quote}}$ ( quote( s $N$ , $K$ ) )

EQUATION: quote( $N$ , ~( $Kl$ ) ) = $\boxed{\sim}$ ( quote( $N$ , $Kl$ ) )

EQUATION: quote( s $N$ , unquote $K$ ) = $\boxed{\text{unquote}}$ ( quote( $N$ , $K$ ) )

RULE: 

$\Rightarrow$ •

RULE: $F_1$ * $F_2$ $\Rightarrow$ $F_1$ * $F_2$

RULE: $I_1$ * $I_2$ $\Rightarrow$ $I_1$ * $I_2$

RULE: $F_1 + F_2 \Rightarrow F_1 + F_2$

RULE: $I_1 + I_2 \Rightarrow I_1 + I_2$

RULE: $F_1 \leq F_2 \Rightarrow F_1 \leq F_2$

RULE: $I_1 \leq I_2 \Rightarrow I_1 \leq I_2$

RULE: false and — $\Rightarrow$ false

RULE: true and $BE \Rightarrow BE$

RULE: if false then — else $S_2 \Rightarrow S_2$

RULE: if true then $S_1$ else — $\Rightarrow S_1$

RULE: not $B \Rightarrow$ not $B$

END MODULE