5.7 The Challenge Language

This section contains the complete K semantic definition of CHALLENGE, a non-trivial experimental programming language that we propose here in order to challenge existing or future language definitional frameworks. All language constructs of CHALLENGE can be found in existing programming languages, possibly in a more general form and possibly using a syntax slightly different from ours. In short, none of the CHALLENGE features is artificially crafted and the language, as a whole, is plausible and powerful.

We chose features that already exist in popular programming languages to avoid criticisms, from proponents of frameworks that cannot handle the proposed features, that those features are useless and thus do not deserve attention. Our standpoint is that the very fact that a language feature exists in a mainstream programming language is sufficient evidence that the feature is practical and useful, so an ideal semantic framework should unarguably support it: while a language designer may subjectively chose to include such a feature in her language or not, a language-design framework designer aiming at an ideal framework, as we do, has no choice and should simply support it. The CHALLENGE features, and particularly their combination, have been carefully chosen to satisfy two criteria:

1. To capture *the essence* of several conventional language concepts; if a semantic framework cannot handle a certain CHALLENGE feature then it is clear that that framework cannot handle a potentially wide range of desirable language features;

2. To illustrate how one can define a complex language *modularly* in K: each new feature is added to the language without changing any of the already defined features. We here assume that one knows upfront the entire language and that at each step one takes the globally best design decisions. In Section 5.8 we discuss a more complex language design scenario.

```
Exp ::= Bool | Int | Real | VarId
      | Exp + Exp [strict]
      | Exp * Exp [strict]
      | Exp / Exp [strict]
      | Exp <= Exp [seqstrict]
      | not Exp [strict]
      | Exp and Exp [strict(1)]
      | ++ VarId
      | Stmt ; Exp
      | malloc Exp [strict]
      | & VarId
      | * Exp [strict]
      | λList(){VarId} . Exp
      | Exp List(){Exp} [strict]
      | μ VarId . Exp
      | callcc Exp [strict]
      | randomBool
      | new-agent Stmt
      | me
      | parent
      | receive
      | receive-from Exp [strict]
      | quote Exp
      | unquote Exp
      | eval Exp [strict]
Stmt ::=
      | Stmt ; Stmt
      | { vars List{VarId} ; Stmt}
      | if Exp then Stmt else Stmt [strict(1)]
      | while Exp do Stmt
      | output Exp [strict]
      | Exp := Exp [strict(2)]
      | aspect Stmt
      | spawn Stmt
      | acquire Exp [strict]
      | free Exp [strict]
      | release Exp [strict]
      | rv Exp [strict]
      | send-asynch Exp Exp [strict]
      | send-synch Exp Exp [strict]
      | halt-thread
      | halt-agent
      | halt-system
```

Figure 5.13: K annotated CHALLENGE syntax

5.7.1 Challenge Syntax

Figure 5.13 shows the syntax of CHALLENGE, which is split into two syntactic categories: expressions (*Exp*) and statements (*Stmt*). To save space and give the reader an early intuition on the evaluation strategies of the various constructs, the syntax of CHALLENGE in Figure 5.13 is already annotated with K strictness attributes (Section 5.5). The language starts with simple arithmetic expressions, then gradually grows in complexity by adding expressions with side effects, memory allocation and pointers, functions, recursion, call with current continuation, non-determinism, multi-threading with shared memory and thread synchronization, agents and both synchronous and asynchronous inter-agent communication, and in the end code generation. We next informally explain the CHALLENGE language constructs and briefly give our reasons for including them in CHALLENGE. Section 5.8 discusses in depth how the various semantic frameworks in Chapter 3 encounter difficulties in supporting some of the CHALLENGE features, and Chapters 10 and 12 elaborate on the meaning of these features. We here assume the reader familiar with them.

Expressions can be both arithmetic and boolean, and include both integer and real numbers. The arithmetic/boolean expression constructs and their evaluation strategies are similar to those of IMP, which are discussed in Section 5.2. However, we keep the integer and real numbers separate and all the arithmetic and relational operations are overloaded to work with both. Together with the top-level split of syntax into expressions and statements, we believe that the above make CHALLENGE stress relatively well the capabilities of the semantic framework to deal with multiple syntactic categories, both disjoint and included in each other, as well as with operator overloading.

Expressions can have side effects, both minimal (with variable increment, e.g., "`++`x") and arbitrarily complex ("s `;` e" means "evaluate statement s, then evaluate expression e in the resulting state and return its value"). Side effects are very common in most programming languages and their inclusion may require non-modular changes when one uses certain semantic frameworks.

CHALLENGE allows to allocate memory with a `malloc` expression construct and allows to deallocate memory with a `free` statement construct. Like in C, `malloc` evaluates to the location where the allocated block starts, and `free` can only be called on a location which has been previously allocated with a `malloc` (so that the number of locations to free is known). Like C, CHALLENGE also allows unrestricted pointer arithmetic, allows accessing the address at which a variable is allocated with the expression construct `&`x, allows dereferencing locations with the expression construct `*`p, and allows writing values to locations using statements of the form `*`p `:=` e. Unlike in C, we prefer to use `:=` instead of `=` for assignment, to avoid notational confusion with the equality symbol of equational logic that we use many places in this book. Not all programming languages allow as explicit and direct memory access as CHALLENGE. Nevertheless, a semantic framework able to naturally deal with the memory operations of CHALLENGE can likely support arbitrarily complex memory operations in programming language semantic definitions.

Any semantic framework worth its salt must support definitions of languages with functions. Since higher-order functions tend to be more complex than other functions and since there are many higher-order functional languages, it suffices to include only higher-order functions in CHALLENGE. We do it by means of λ-expressions (Section 4.4). To stress the framework's capability to syntactically and semantically deal with lists of terms, we allow λ-abstractions taking arbitrary lists of parameters ($\lambda\mathbf{List}^{()}\{\mathit{VarId}\}\,.\,\mathit{Exp}$) and λ-applications taking arbitrary lists of arguments ($\mathit{Exp}\,\mathbf{List}^{()}\{\mathit{Exp}\}$) in CHALLENGE, with $()$ the unit of these list constructs. For example, $(\lambda().\,e)()$, $(\lambda x.\,e)e'$, and $(\lambda(x_1,x_2).\,e)(e_1,e_2)$ are all well-formed CHALLENGE expressions when $x, x_1, x_2 \in \mathit{VarId}$ and e, e', e_1, e_2 are well-formed expressions. Note that we defined the λ-application to be strict both in its

first argument (expected to evaluate to a function) and in its second argument; its second argument is a list of expressions, so the application is implicitly strict in all these expressions in the list (i.e., each of the expressions in the list will be evaluated before the function is applied; how this works is explained in Section 5.5); therefore, CHALLENGE is call-by-value (see Chapter 12).

Since recursion is a fundamental design and semantic concept in programming languages, and since recursion can have many apparently different facets, CHALLENGE allows three different means to achieve recursion: (1) through implicit recursion obtained using conventional λ-calculus encodings of fixed-point combinators (see Section 4.4), which only work in untyped settings; (2) through iteration using the provided `while` loop statement construct (`while` *Exp* `do` *Stmt*); and (3) through explicit use of a provided fixed-point μ construct ($\mu\, VarId\,.\, Exp$). Even though in theory only one approach to recursion suffices, we strongly believe that a language definitional framework should naturally support all three approaches in practice, to avoid artificial encodings of one into another.

Many languages have control-intensive constructs, such as abrupt termination, exceptions, break/continue of loops, and even callcc (from "call with current continuation"). Therefore, any worthwhile language definitional framework must support language constructs that abruptly change the execution flow. Since callcc is as complex a control-intensive construct as it can be, we included it as part of the CHALLENGE language (`callcc` *Exp*). We also included several abrupt termination statements (for threads, agents and program), which we discuss shortly. Other control constructs, such as exceptions and break/continue of loops, are defined and discussed in Chapters 10 and 12.

Some semantic frameworks, particularly those of a denotational nature, cannot handle non-determinism and/or concurrency satisfactorily: while they still allow in some cases to define a collecting semantics (i.e., a semantics that collects all behaviors), such a semantics is inappropriate for obtaining an interpreter for the defined language when executing it. To illustrate a framework's capability to deal with non-determinism, CHALLENGE includes the simplest imaginable non-deterministic construct, namely a `randomBool` boolean constant which non-deterministically evaluates to *true* or *false* each time it is evaluated.

Concurrency and distribution of software in general and of programming languages in particular is not anymore the exception, but the norm. Thus, any practical language definitional framework should provide good support for concurrency. Therefore, CHALLENGE contains language constructs that stress the underlying framework in several ways with respect to concurrency. It allows to dynamically create and terminate both agents (`new-agent` *Stmt*) and threads (`spawn` *Stmt*). The threads live inside agents and the threads inside an agent communicate and synchronize with each other by means of shared memory, re-entrant acquisition (`acquire` *Exp*) and releasing (`release` *Exp*) of locks (any value can serve as a lock), and by rendez-vous barriers (`rv` *Exp*). Agents communicate with each other by means of synchronous (`send-synch` *Exp Exp*) and asynchronous (`send-asynch` *Exp Exp*) message-send commands (the first argument is the receiving agent's name and the second argument is value being passed), and by targeted (`receive-from` *Exp*, where the argument is the sending agent's name) and non-targeted (`receive`) expression constructs. If a language semantic framework supports the above common concurrency features, then it likely can support any degree of concurrency that a language designer may want to include in her language.

To keep a tight connection between the concurrency features and the rest of the language, CHALLENGE includes several other features that do not dictate the semantics of the concurrent ones but interact with them. For example, since there are three types of executing entities, namely threads, agents, and the entire system, it makes sense to have three abrupt termination statements, one for the current thread (`halt-thread`), one for the current agent (`halt-agent`) and one for

the entire system (`halt-system`); the former also releases the resources (i.e., locks) that the thread held. We also included in CHALLENGE an output that is shared by all threads and all agents (via the `output` statement construct). The previously discussed constructs `malloc` / `free` act on the memory of the current agent only (so the agent's threads compete on memory), and `callcc` acts on the computation/continuation of the current thread only.

Aspect-oriented programming is an important increasingly growing trend in programming languages. There are many different approaches to aspects (e.g., dynamic versus static aspects) and we cannot possibly include all of them in CHALLENGE. We only include one aspect-oriented feature, namely executing an aspect statement whenever a function is invoked. To make it more interesting, we combine static with dynamic semantics of aspects in that we statically weave the current aspect before the body of a function at function declaration time, but we allow one to dynamically change the aspect code. This can be done by executing a statement "`aspect` s", which sets the argument statement s as the aspect of the current thread. Many variations are possible, particularly in combination with concurrency, but we prefer to keep CHALLENGE minimal. We added the particular aspect feature above for two reasons: first, executing code when a function is called is one of the most common aspects; second, it may be inconvenient to give it semantics in purely syntactic approaches (one needs to distinguish between weaved and not-yet-weaved functions) or in approaches which reduce desired language features to their "builtin" similar features (e.g., desired functions to their builtin functions, desired recursion to their fixed-points, etc.).

Finally, there are several programming languages providing support for dynamic code generation and execution of it (e.g., Lisp, Scheme, Jumbo, etc.). We therefore add support for code generation to CHALLENGE. We borrow our terminology from Scheme. The expression "`quote` (e)" freezes expression e into a special code value, and "`eval` (e)" first evaluates e to a code value, say c, and then unfreezes c and evaluates it in the current environment. To allow for arbitrarily complex code generation and reuse, one can use expressions like "`unquote` (e)" inside a quoted expression, with the following semantics: evaluate e in the current environment to a code value, then plug that code value into the quoted expression. Quoting of code can be nested and an expression must be unquoted as many times as quoted in order to be evaluated. Generation of code can be challenging or impossible to define modularly in most frameworks, because they drive the semantics according to the structure of the syntax, where the individuality of each language construct is important. Thus, most likely one needs to define quoting/unquoting for each language construct, which is non-modular. In K, we can define the semantics of `quote` / `unquote` / `eval` with 13 language independent rules.

5.7.2 Challenge Configuration

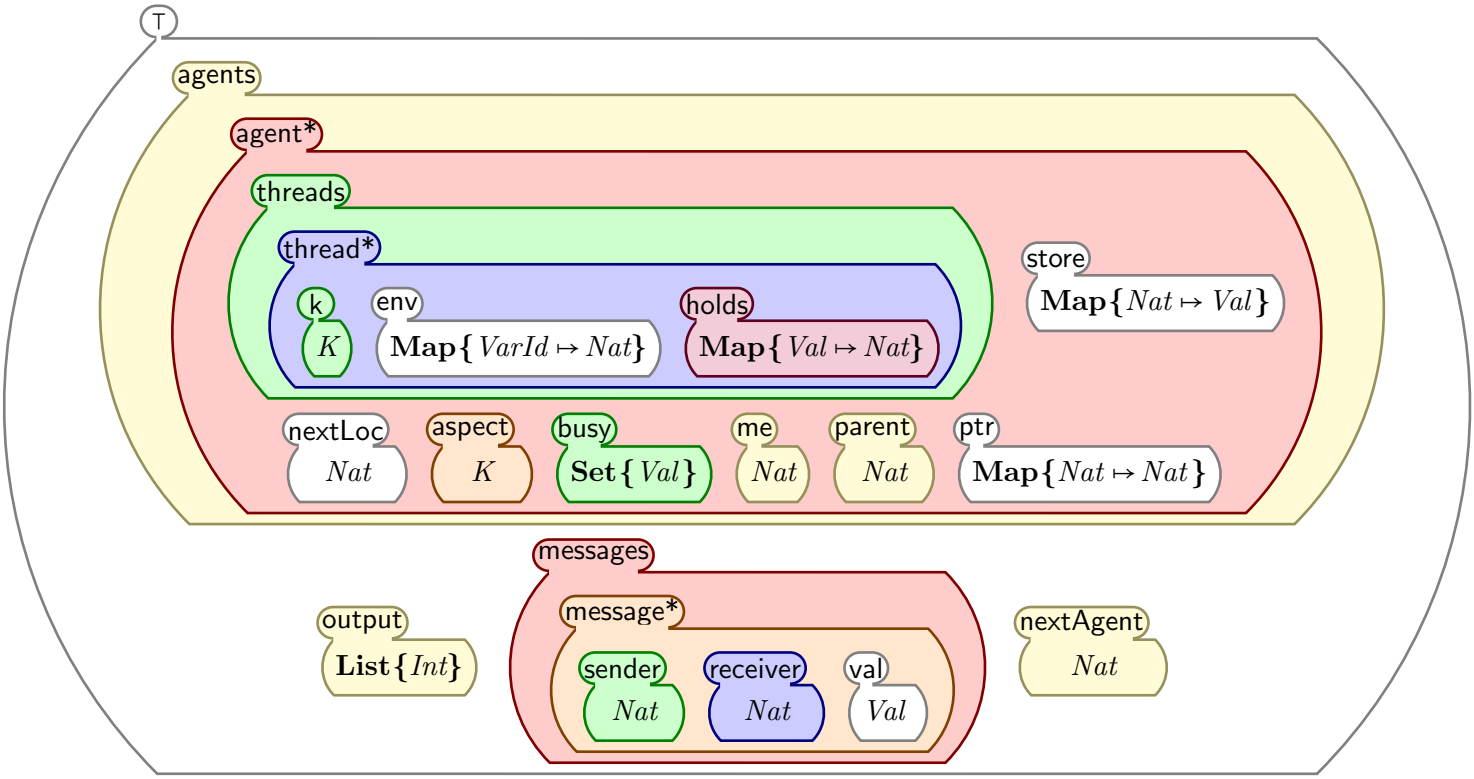
Figure 5.14 shows the complete K-configuration of CHALLENGE, both in term representation and in graphical representation. The cell contents is self-explanatory, but more detail will be given as we define the various language constructs in Section 5.7.3

5.7.3 Challenge Semantics

We will only give here rules for constructs having a different behavior than the one already presented for IMP++.

Reading from the 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

$$
\begin{array}{rcl}
\mathit{Configuration}_{\textsc{Challenge}} & \equiv & \langle \mathit{Agents}_{\textsc{Challenge}}\ \langle \mathbf{List}\{\mathit{Int}\}\rangle_{\mathsf{output}}\ \mathit{Messages}_{\textsc{Challenge}}\ \langle \mathit{Nat}\rangle_{\mathsf{nextAgent}}\rangle_{\top} \\
\mathit{Agents}_{\textsc{Challenge}} & \equiv & \left\langle \left\langle \begin{array}{c} \mathit{Threads}_{\textsc{Challenge}}\ \langle \mathbf{Map}\{\mathit{Nat} \mapsto \mathit{Val}\}\rangle_{\mathsf{store}} \\ \langle \mathit{Nat}\rangle_{\mathsf{nextLoc}}\ \langle K\rangle_{\mathsf{aspect}}\ \langle \mathbf{Set}\{\mathit{Val}\}\rangle_{\mathsf{busy}} \\ \langle \mathit{Nat}\rangle_{\mathsf{me}}\ \langle \mathit{Nat}\rangle_{\mathsf{parent}}\ \langle \mathbf{Map}\{\mathit{Nat} \mapsto \mathit{Nat}\}\rangle_{\mathsf{ptr}} \end{array} \right\rangle_{\mathsf{agents}*} \right\rangle_{\mathsf{agents}} \\
\mathit{Threads}_{\textsc{Challenge}} & \equiv & \langle\langle\langle K\rangle_{\mathsf{k}}\ \langle \mathbf{Map}\{\mathit{VarId} \mapsto \mathit{Nat}\}\rangle_{\mathsf{env}}\ \langle \mathbf{Map}\{\mathit{Val} \mapsto \mathit{Nat}\}\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}*}\rangle_{\mathsf{theread}} \\
\mathit{Messages}_{\textsc{Challenge}} & \equiv & \langle\langle\langle \mathit{Nat}\rangle_{\mathsf{sender}}\ \langle \mathit{Nat}\rangle_{\mathsf{receiver}}\ \langle \mathit{Val}\rangle_{\mathsf{val}}\rangle_{\mathsf{message}*}\rangle_{\mathsf{messages}}
\end{array}
$$



$\langle\rangle_{\top}$	: top cell, holding everything	$\langle\rangle_{\mathsf{busy}}$	: holds agent's busy locks
$\langle\rangle_{\mathsf{agents}}$	: holds all the agents	$\langle\rangle_{\mathsf{me}}$	: holds agent's id
$\langle\rangle_{\mathsf{agent}*}$	: holds one agent, can multiply	$\langle\rangle_{\mathsf{parent}}$	: holds agent's parent if
$\langle\rangle_{\mathsf{threads}}$	: holds all agent's threads	$\langle\rangle_{\mathsf{ptr}}$	: holds agent's store allocation map
$\langle\rangle_{\mathsf{thread}*}$	: holds one thread, can multiply	$\langle\rangle_{\mathsf{output}}$	: holds the system output
$\langle\rangle_{\mathsf{k}}$	: holds thread's computation	$\langle\rangle_{\mathsf{messages}}$	: holds all the pending messages
$\langle\rangle_{\mathsf{env}}$	: holds thread's environment	$\langle\rangle_{\mathsf{message}*}$	: holds one message, can multiply
$\langle\rangle_{\mathsf{holds}}$	: holds thread's locks	$\langle\rangle_{\mathsf{sender}}$	: holds the message sender's id
$\langle\rangle_{\mathsf{store}}$	: holds agent's store	$\langle\rangle_{\mathsf{receiver}}$	: holds the message receiver's id
$\langle\rangle_{\mathsf{nextLoc}}$	: holds the next available location	$\langle\rangle_{\mathsf{val}}$	: holds the message's value
$\langle\rangle_{\mathsf{aspect}}$	: holds agent's aspect	$\langle\rangle_{\mathsf{nextAgent}}$	: holds the next available agent id

Figure 5.14: The configuration of the Challenge language in both term (top) and graphical (middle) representation, with short explanation of cell contents (bottom)

the mapping of L to V can be matched in the store:

$$\langle \underline{x}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto l\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ l \mapsto v\ \cdots \rangle_{\mathsf{store}}$$
$$\underline{v}$$

Statement Block with variable declaration When variables are declared, new locations are allocated in the environment. We use the 'nextLoc' cell to hold the next available location; also, we use `Rho[X <- L]` to map X to L in Rho. Once the executions of the statements in the block is completed, the old environment must be resumed.

$$\langle \underline{\{\,\mathtt{vars}\ xl\ ;\ s\,\}} \cdots \rangle_{\mathsf{k}}\ \langle \underline{\qquad \rho \qquad} \rangle_{\mathsf{env}}\ \langle \underline{\qquad \sigma \qquad} \rangle_{\mathsf{store}}\ \langle \underline{\quad n \quad} \rangle_{\mathsf{nextLoc}}$$
$$s \curvearrowright \mathsf{env}(\rho) \qquad\qquad \rho[n\,..\,n +_{Int} |xl| -_{Int} 1/xl] \qquad\qquad \sigma[0/n\,..\,n +_{Int} |xl| -_{Int} 1] \qquad\qquad n +_{Int} |xl|$$

where

$$\langle \underline{\mathsf{env}(\rho)}\ \cdots \rangle_{\mathsf{k}}\ \langle \underline{\_} \rangle_{\mathsf{env}}$$
$$\cdot \qquad\qquad \rho$$

Side effects: increment

$$\langle \underline{\ \mathtt{++}\,x\ }\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto l\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ l \mapsto \underline{\quad i \quad}\ \cdots \rangle_{\mathsf{store}}$$
$$i +_{Int} 1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad i +_{Int} 1$$

References Obtaining the reference location of a name:

$$\langle \underline{\mathtt{\&}\,x}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto l\ \cdots \rangle_{\mathsf{env}}$$
$$l$$

Dereferencing a location:

$$\langle \underline{\mathtt{*}\,l}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ l \mapsto v\ \cdots \rangle_{\mathsf{store}}$$
$$v$$

Memory allocation and free:

$$\langle \underline{\mathtt{malloc}\,(n)}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \underline{\quad \cdot \quad}\ \cdots \rangle_{\mathsf{ptr}}\ \langle \underline{\qquad \sigma \qquad} \rangle_{\mathsf{store}}\ \langle \underline{\quad l \quad} \rangle_{\mathsf{nextLoc}}$$
$$l \qquad\qquad\qquad\qquad l \mapsto n \qquad\qquad\qquad\qquad \sigma[0/l\,..\,l +_{Int} n -_{Int} 1] \qquad\qquad l +_{Int} n$$

$$\langle \underline{\mathtt{free}\,(l)}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \underline{l \mapsto n}\ \cdots \rangle_{\mathsf{ptr}}\ \langle \underline{\qquad \sigma \qquad} \rangle_{\mathsf{store}}$$
$$\cdot \qquad\qquad\qquad\qquad \cdot \qquad\qquad\qquad\qquad \sigma[\bot/l\,..\,l +_{Int} n -_{Int} 1]$$

Exercise 176. *Evaluating an expression and returning a reference to it:*

$$\mathtt{ref} \equiv \lambda x.(\lambda p.\,\mathtt{*}\,p\ \mathtt{:=}\ x\ ;\ p)(\,\mathtt{malloc}\,(1))$$

$$\langle \underline{\mathtt{ref}\,v}\ \cdots \rangle_{\mathsf{k}}\ \langle \underline{\quad \sigma \quad} \rangle_{\mathsf{store}}\ \langle \underline{\quad l \quad} \rangle_{\mathsf{nextLoc}}$$
$$l \qquad\qquad\qquad\qquad \sigma[v/l] \qquad\qquad\qquad\qquad l +_{Int} 1$$

Assignment The right hand side of an assignment is evaluated to a value, specified 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).

$$\langle \underset{\cdot}{\underline{x := v}} \cdots \rangle_{\mathsf{k}} \ \langle \cdots \ x \mapsto l \ \cdots \rangle_{\mathsf{env}} \ \langle \cdots \ l \mapsto \underset{v}{\underline{\_}} \ \cdots \rangle_{\mathsf{store}}$$

context: $* e := e' \ [strict(e)]$

$$\langle \underset{\cdot}{\underline{* l := v}} \cdots \rangle_{\mathsf{k}} \ \langle \cdots \ l \mapsto \underset{v}{\underline{\_}} \ \cdots \rangle_{\mathsf{store}}$$

Function and aspects Since our language allows usage of pointer and references, it is more convenient to work with environments and to represent functions as closures. In addition, the Challenge language allows the specification of aspects, whose semantics is that they affect each function defined after the '`aspect`' declaration statement to execute the statement provided as its argument prior to execution their body, in the environment of the function. This is achieved by using a special cell to hold the current aspect, and by including the computation contained in that cell in the closure of each function being evaluated.

$$\langle \underset{\cdot}{\underline{\texttt{aspect}\ s}} \cdots \rangle_{\mathsf{k}} \ \langle \underset{s}{\underline{\_}} \rangle_{\mathsf{aspect}}$$

$$\langle \underset{closure(xl, s \curvearrowright e, \rho)}{\underline{\lambda xl \,.\, e}} \cdots \rangle_{\mathsf{k}} \ \langle \rho \rangle_{\mathsf{env}} \ \langle s \rangle_{\mathsf{aspect}}$$

Exercise 177. *Define a different semantics for aspects, or a new aspect construct, that would weave the aspect at function call time instead of at function declaration time.*

Function application Upon function call, the evaluated arguments are bound to the formal parameters, then the body of the function is executed in the environment saved by the closure, and finally the calling environment must restored.

$$\langle \underset{e \curvearrowright \mathsf{env}(\varrho)}{\underline{closure(xl, e, \rho) \ vl}} \cdots \rangle_{\mathsf{k}} \ \langle \underset{\rho[n \,..\, n +_{Int} |xl| -_{Int} 1/xl]}{\underline{\varrho}} \rangle_{\mathsf{env}} \ \langle \underset{\sigma[vl/n \,..\, n +_{Int} |xl| -_{Int} 1]}{\underline{\sigma}} \rangle_{\mathsf{store}} \ \langle \underset{n +_{Int} |xl|}{\underline{n}} \rangle_{\mathsf{nextLoc}}$$

Restoring the original environment, when the body of the function is completely evaluated:

$$\langle v \curvearrowright \underset{\cdot}{\underline{\mathsf{env}(\rho)}} \cdots \rangle_{\mathsf{k}} \ \langle \underset{\rho}{\underline{\_}} \rangle_{\mathsf{env}}$$

Tail call optimization:

$$\underset{\cdot}{\underline{\mathsf{env}(\_)}} \curvearrowright \mathsf{env}(\_)$$

Exercise 178. *Prove that the tail call optimization is sound.*

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

$$\langle \frac{\mu x\,.\,e}{e \curvearrowright \mathsf{env}(\rho)} \cdots \rangle_{\mathsf{k}}\ \langle \frac{\rho}{\rho[n/x]} \rangle_{\mathsf{env}}\ \langle \cdots \frac{\cdot}{n \mapsto \mu x\,.\,e} \cdots \rangle_{\mathsf{store}}\ \langle \frac{n}{n +_{Int} 1} \rangle_{\mathsf{nextLoc}}$$

Call with current continuation (call/cc) The semantics of call/cc is that of packaging the reminder of the computation as a value and passing it to the function passed as argument. When the thus packaged computation is applied on a value, the entire computation at tha time is replaced by the value being put on top of the packed computation. Note that the environment needs to be packed together with the computation, same as for function closures.

$$\langle \frac{\texttt{callcc}\ v \curvearrowright k}{v\ cc(k,\rho)} \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}$$

$$\langle \frac{cc(k,\rho)\ v \cdots}{v \curvearrowright k} \rangle_{\mathsf{k}}\ \langle \frac{\_}{\rho} \rangle_{\mathsf{env}}$$

Sequential non-determinism 'randomBool' non-deterministically evaluates to either 'true' or 'false'.

$$\langle \frac{\texttt{randomBool} \cdots}{true} \rangle_{\mathsf{k}}$$

$$\langle \frac{\texttt{randomBool} \cdots}{false} \rangle_{\mathsf{k}}$$

Threads Threads are characterized by independent control flow, but shared memory. In our definition, a 'thread' cell is used to group together all (computation, environment, acquired resources) cells related to a thread The '`spawn`' command creates a new thread which is initialized with the statement passed as argument as its computation, and the environment of the thread creating it.

$$\langle \cdots \langle \frac{\texttt{spawn}\ s \cdots}{\cdot} \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}} \cdots \rangle_{\mathsf{thread}}\ \frac{\cdot}{\langle \langle s \rangle_{\mathsf{k}}\ \langle \rho \rangle_{\mathsf{env}}\ \langle \cdot \rangle_{\mathsf{holds}} \rangle_{\mathsf{thread}}}$$

When the computation of a thread has completed, it can be dissolved and its held resources be released. 'busy' is a shared cell holding the names of the resources acquired by any of the threads.

$$\langle \cdots \langle \cdot \rangle_{\mathsf{k}}\ \langle h \rangle_{\mathsf{holds}} \cdots \rangle_{\mathsf{thread}}\ \langle \frac{busy}{busy -_{set} dom_{map}(h)} \rangle_{\mathsf{busy}}$$

Thread synchronization To attain mutual exclusion, threads can be synchronized by means of locks. In this language, 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 multiplicities for each lock are maintained in the 'holds' cell.

$$\langle \underset{\cdot}{\texttt{acquire}\ v} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ v \mapsto \underset{s_{Nat}(n)}{n} \cdots\rangle_{\mathsf{holds}}$$

$$\langle \underset{\cdot}{\texttt{acquire}\ v} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \underset{v \mapsto 0}{\cdot} \cdots\rangle_{\mathsf{holds}}\ \langle busy\ \underset{v}{\cdot}\rangle_{\mathsf{busy}} \quad \text{when } \neg_{Bool}(v \in_{set} busy)$$

$$\langle \underset{\cdot}{\texttt{release}\ v} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ v \mapsto \underset{n}{s_{Nat}(n)} \cdots\rangle_{\mathsf{holds}}$$

$$\langle \underset{\cdot}{\texttt{release}\ v} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \underset{\cdot}{v \mapsto 0} \cdots\rangle_{\mathsf{holds}}\ \langle \cdots\ \underset{\cdot}{v} \cdots\rangle_{\mathsf{busy}}$$

Rendez-vous synchronization Two threads can only pass together a barrier specified by a lock V. This is harder to achieve in all definitional frameworks we know of, because it requires the ability to access and reduce redexes of two computational units simultaneously.

$$\langle \underset{\cdot}{\texttt{rv}\ v} \cdots\rangle_{\mathsf{k}}\ \langle \underset{\cdot}{\texttt{rv}\ v} \cdots\rangle_{\mathsf{k}}$$

Agents 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.

$$\langle \underset{n}{\texttt{new-agent}\ s} \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \langle \underset{n +_{Int} 1}{n} \rangle_{\mathsf{nextAgent}}\ \underset{NewAgent}{\cdot}$$

where *NewAgent* stands for:

$$\langle\langle\langle s\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{env}}\ \langle\cdot\rangle_{\mathsf{holds}}\rangle_{\mathsf{thread}}\ \langle\cdot\rangle_{\mathsf{busy}}\ \langle n\rangle_{\mathsf{me}}\ \langle m\rangle_{\mathsf{parent}}\ \langle\cdot\rangle_{\mathsf{store}}\ \langle 0\rangle_{\mathsf{nextLoc}}\ \langle\cdot\rangle_{\mathsf{aspect}}\rangle_{\mathsf{agent}}$$

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

$$\langle \cdots\ \langle\cdot\rangle_{\mathsf{threads}} \cdots\rangle_{\mathsf{agent}} \rightharpoonup \cdot$$

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.

$$\langle \underset{m}{\texttt{me}} \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}$$

$$\langle \underset{n}{\texttt{parent}} \cdots\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{parent}}$$

$$\langle \underset{\cdot}{\texttt{send-asynch}\ n\ v} \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \underset{\langle\langle m\rangle_{\mathsf{sender}}\ \langle n\rangle_{\mathsf{receiver}}\ \langle v\rangle_{\mathsf{val}}\rangle_{\mathsf{message}}}{\cdot}$$

An agent can request to receive a message from a certain agent, or from any agent.

$$\langle \underset{v}{\texttt{receive-from}\ n}\ \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \underset{\cdot}{\langle\langle n\rangle_{\mathsf{sender}}\ \langle m\rangle_{\mathsf{receiver}}\ \langle v\rangle_{\mathsf{val}}\rangle_{\mathsf{message}}}$$

$$\langle \underset{v}{\texttt{receive}}\ \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \underset{\cdot}{\langle\cdots\ \langle m\rangle_{\mathsf{receiver}}\ \langle v\rangle_{\mathsf{val}}\ \cdots\rangle_{\mathsf{message}}}$$

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

$$\langle\cdots\ \langle \underset{\cdot}{\texttt{send-synch}\ n\ v}\ \cdots\rangle_{\mathsf{k}}\ \langle m\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \langle \underset{v}{\texttt{receive-from}\ m}\ \langle n\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}$$

$$\langle \underset{\cdot}{\texttt{send-synch}\ n\ v}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \langle \underset{v}{\texttt{receive}}\ \cdots\rangle_{\mathsf{k}}\ \langle n\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}$$

Exercise 179. *While the rules for asynchronous sending allow the sender and the receiver to be same agent, the two rules for synchronous sending assume that the sender and the receiver agents are different. Add two other rules for synchronous sending that extend the semantics above to allow agents to communicate synchronously with themselves (they would have to send from one thread and receive from another thread).*

Abrupt termination Execution of '`halt`' in one of the threads of an agent dissolves that agent.

$$\langle \texttt{halt-thread}\ \cdots\rangle_{\mathsf{k}} \to \langle\cdot\rangle_{\mathsf{k}}$$

$$\langle\cdots\ \langle \texttt{halt-agent}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{threads}} \to \langle\cdot\rangle_{\mathsf{threads}}$$

$$\langle\cdots\ \langle \texttt{halt-system}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agents}} \to \langle\cdot\rangle_{\mathsf{agents}}$$

Code Generation The equations below implement the quote/unquote mechanism of runtime code generation. First, the syntax of computations is extended to include "boxed" versions of the K arrow and K list constructors, as well as the helping operator quote, which tracks the level of nesting of quoted expressions. Similarly, boxed versions are introduced for all K labels (ad thus for all language constructs), and a new Label code is added to hold values of type code

$$\begin{array}{rcl} \mathit{KLabel} & ::= & \_\boxed{\rightsquigarrow}\_\ [\mathit{strict}]\ \mid\ \boxed{\mathit{KLabel}}\ [\mathit{strict}]\ \mid\ \_\boxed{,}\_\ [\mathit{strict}]\ \mid\ \mathit{quote}[\mathit{Nat}] \\ \mathit{KResultLabel} & ::= & \mathit{code} \end{array}$$

Initially, a quoted expression starts at the quoting level 0, and acts as a morphism on all K constructors, transforming them into their boxed version, except for the quote/unquote labels. Each of this boxed items is supposed to evaluate to a code fragment; once this happens, the fragments are glued together to a bigger code fragment.

$\langle \underset{quote[0]}{\texttt{quote}}\ (k)\ \cdots \rangle_{\mathsf{k}}$

$quote[n](\texttt{quote}(k)) \rightharpoonup \boxed{\texttt{quote}}(quote[s_{Nat}(n)](k))$
$quote[0](\texttt{unquote}(k)) \rightharpoonup k$
$quote[s_{Nat}(n)](\texttt{unquote}(k)) \rightharpoonup \boxed{\texttt{unquote}}(quote[n](k))$

$quote[n](\cdot) \rightharpoonup code(\cdot)$
$quote[n](k_1 \curvearrowright k_2) \rightharpoonup quote[n](k_1)\ \boxed{\curvearrowright}\ quote[n](k_2) \quad \text{when } k_1 \neq \cdot\ ,\ k_2 \neq \cdot$
$code(k_1)\ \boxed{\curvearrowright}\ code(k_2) \rightharpoonup code(k_1 \curvearrowright k_2)$

$quote[n](label(kl)) \rightharpoonup \boxed{label}(quote[n](kl)) \quad \text{when } label \neq \texttt{quote}\ ,\ label \neq \texttt{unquote}$
$\boxed{label}(code(kl)) \rightharpoonup code(label(kl))$

$quote[n](\_) \rightharpoonup code(\_)$
$quote[n](kl_1, kl_2) \rightharpoonup quote[n](kl_1)\ \boxed{,}\ quote[n](kl_2) \quad \text{when } kl_1 \neq \_\ ,\ kl_2 \neq \_$
$code(kl_1)\ \boxed{,}\ code(kl_2) \rightharpoonup code(kl_1, kl_2)$

$\texttt{eval}(code(k)) \rightharpoonup k$