AGENT

MODULE BASIC-EXP-SYNTAX

SYNTAX $\mathit{Exp} ::= (\mathit{Exp})$ [bracket]

END MODULE

MODULE VAL

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

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

END MODULE

MODULE BOOL-EXP-SYNTAX

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

END MODULE

MODULE BOOL-EXP

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

END MODULE

MODULE INT-EXP-SYNTAX

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

END MODULE

MODULE INT-EXP

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

END MODULE

MODULE EXP-SYNTAX

SYNTAX $\mathit{Exp} ::= \mathit{Exp} * \mathit{Exp}$ [strict, mul]
$\mid \mathit{Exp}$ / Exp [strict, div]
$\mid \mathit{Exp} + \mathit{Exp}$ [strict, plus]
$\mid \mathit{Exp} \leq \mathit{Exp}$ [leq, seqstrict]
$\mid \mathit{Exp}$ == Exp [eq, strict]
$\mid$ not Exp [strict, not]
$\mid \mathit{Exp}$ and Exp [and, strict(1)]

END MODULE

MODULE EXP

RULE $\dfrac{I1{:}\mathit{Int} + I2{:}\mathit{Int}}{I1 +_{\mathit{Int}} I2}$

RULE $\dfrac{I1{:}\mathit{Int} * I2{:}\mathit{Int}}{I1 *_{\mathit{Int}} I2}$

RULE $\dfrac{I1{:}\mathit{Int} \;/\; I2{:}\mathit{Int}}{I1 \div_{\mathit{Int}} I2}$ requires $I2 =/=_{\mathit{Int}} 0$

RULE $\dfrac{I1{:}\mathit{Int} \leq I2{:}\mathit{Int}}{I1 \leq_{\mathit{Int}} I2}$

RULE $\dfrac{V1{:}\mathit{Val} \;\texttt{==}\; V2{:}\mathit{Val}}{V1 =_K V2}$

RULE $\dfrac{\texttt{not}\; T{:}\mathit{Bool}}{\neg_{\mathit{Bool}} T}$

RULE $\dfrac{\text{true and } E{:}\mathit{Exp}}{E}$

RULE $\dfrac{\text{false and } E}{\text{false}}$

END MODULE

MODULE IF-SYNTAX

SYNTAX $\mathit{Exp} ::=$ if Exp then Exp else Exp [if, strict(1)]

END MODULE

MODULE IF

RULE $\dfrac{\texttt{if true then}\; E \;\texttt{else}\; —}{E}$

RULE $\dfrac{\texttt{if false then}\; — \;\texttt{else}\; E}{E}$

END MODULE

MODULE ID-EXP-SYNTAX

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

END MODULE

MODULE LAMBDA-SYNTAX

SYNTAX $\mathit{Lambda} ::= \lambda \mathit{Id}.\mathit{Exp}$ [binder, lam]

SYNTAX $\mathit{Exp} ::= \mathit{Exp}\ \ \mathit{Exp}$ [app, strict]
$\mid \mathit{Lambda}$

END MODULE

MODULE LAMBDA

SYNTAX $\mathit{Val} ::= \mathit{Id}$
$\mid \mathit{Lambda}$

SYNTAX $\mathit{Variable} ::= \mathit{Id}$

RULE $\dfrac{(\lambda X{:}\mathit{Id}.E{:}K)\ \ V{:}\mathit{KResult}}{E[V \;/\; X]}$

END MODULE

MODULE MU-SYNTAX

SYNTAX $\mathit{Exp} ::= \mu \mathit{Id}.\mathit{Exp}$ [mu, binder]

END MODULE

MODULE MU

RULE $\dfrac{(\mu X{:}\mathit{Id}.E{:}K)}{E[(\mu X.E) \;/\; X]}$

END MODULE

MODULE CALLCC-SYNTAX

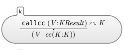
SYNTAX $\mathit{Exp} ::=$ callcc Exp [strict, callCC]

END MODULE

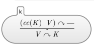
MODULE CALLCC

SYNTAX $\mathit{Val} ::= cc(K)$

RULE



RULE



END MODULE

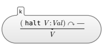
MODULE HALT-SYNTAX

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

END MODULE

MODULE HALT

RULE



END MODULE

MODULE SEQ-SYNTAX

SYNTAX $\mathit{Exp} ::=$ skip
$\mid \mathit{Exp}$; Exp [seq, strict(1)]

END MODULE

MODULE SEQ

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

RULE $\dfrac{V{:}\mathit{Val} \;;\; S{:}K}{S}$ [structural]

END MODULE

MODULE IO-SYNTAX

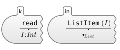
SYNTAX $\mathit{Exp} ::=$ read [read]
$\mid$ print Exp [strict, prnt]

END MODULE

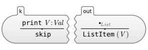
MODULE IO

CONFIGURATION:

RULE



RULE



END MODULE

MODULE REF-SYNTAX

SYNTAX $\mathit{Exp} ::=$ ref Exp [ref, strict]
$\mid * \mathit{Exp}$ [strict, dref]
$\mid \mathit{Exp}$:= Exp [strict(2), assgn]

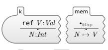
END MODULE

MODULE REF

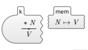
CONFIGURATION:

CONTEXT $* \square$:= —

RULE

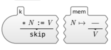


RULE



[transition]

RULE



[transition]

END MODULE

MODULE WHILE-SYNTAX

SYNTAX $\mathit{Exp} ::=$ while Exp do Exp [while]

END MODULE

MODULE WHILE

RULE $\dfrac{\texttt{while}\; E \;\texttt{do}\; S}{\texttt{if}\; E \;\texttt{then}\; (S \;;\; \texttt{while}\; E \;\texttt{do}\; S) \;\texttt{else skip}}$

END MODULE

MODULE THREADS-SYNTAX

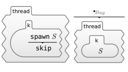
SYNTAX $\mathit{Exp} ::=$ acquire Exp [acq, strict]
$\mid$ release Exp [strict, rel]
$\mid$ rendezvous Exp [rndv, strict]
$\mid$ spawn Exp [spwn]

END MODULE

MODULE THREADS

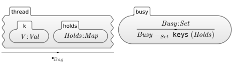
CONFIGURATION:

RULE



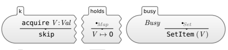
[transition]

RULE



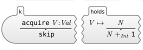
[transition]

RULE

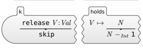


requires $\neg_{\mathit{Bool}}(V \;\texttt{in}\; \mathit{Busy}{:}\mathit{Set})$ [transition]

RULE

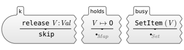


RULE

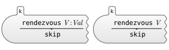


requires $N >_{\mathit{Int}} 0$

RULE



RULE



[transition]

END MODULE

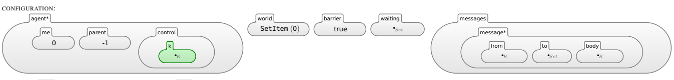
MODULE AGENTS-SYNTAX

SYNTAX $\mathit{Exp} ::=$ newAgent Exp [newAg]
$\mid$ me [me]
$\mid$ parent [parent]
$\mid$ receive [rcv]
$\mid$ receiveFrom Exp [rcvFr, strict]
$\mid$ send Exp to Exp [strict, sndTo]
$\mid$ sendSynch Exp to Exp [sndSyn, strict]
$\mid$ barrier [bar]
$\mid$ broadcast Exp [bcst, strict]
$\mid$ haltAgent [haltAg]

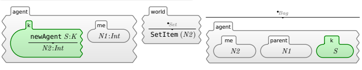
END MODULE

MODULE AGENTS

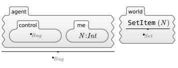
CONFIGURATION:



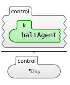
RULE



RULE

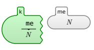


RULE

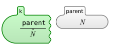


[transition]

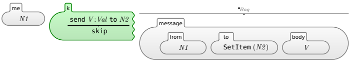
RULE



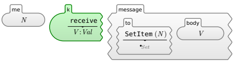
RULE



RULE

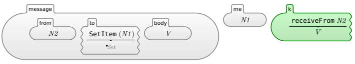


RULE



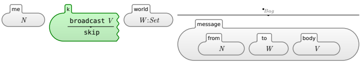
[transition]

RULE



[transition]

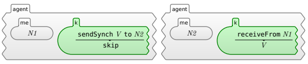
RULE



RULE

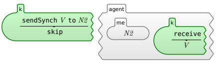
[structural]

RULE



[transition]

RULE



[transition]

RULE

requires $\neg_{Bool}(N \texttt{ in } W)$

RULE

requires $W \neq_K \bullet_{Set}$

[transition]

RULE

RULE

END MODULE

Generic Visitors

𝕂 allows users to define generic visitors, that is, visitors that take any K term and transform it according to given parameters. The particular user-defined syntax of the target language is irrelevant for the visitor, because that is all eventually transformed into K terms, and the visitors work with the latter. This is in sharp contrast with conventional visitor-like semantic approaches, such as those encountered in conventional semantics of `quote/unquote` constructs in languages with support for code generation, which are language-specific, that is, which have a case (semantic rule) for each language construct. Our generic visitor approach is possible thanks to 𝕂's meta-representation of syntax as `KLabel`s and applications of them to `KList`s.

Our current visitor has an API consisting of three `KLabel`s:

- `#visit`, which is expected to take two arguments, the K term to visit and the actual visitor.
- `#visitor`, which is a `KItem` stating what to do during the visit. Currently, `#visitor` expects four arguments: the first two describe the action to take, and the later two describe the condition under which to take the action. We need two arguments for each because both the action and the condition consist of a `KLabel` and a partial list of arguments for it. The complete list of arguments is obtained by appending the actual node being visited to the partial list of arguments.
- `#visited`, which is a wrapper for the visited K term.

One important aspect of 𝕂 visitors is that they need to allow for code (i.e., K terms) to be executed during the visiting process. For example, in an implementation of `quote/unquote` using visitors, code which is unquoted the right number of times (as many times as it has been quoted) has to be executed in exactly that context. This is achieved by simply allowing the action embedded in the visitor to do anything, including replacing the visited node with code to be executed, in particular with itself. In order for this to work, the temporary constructs used during the top-down traversal saying that the term is being visited need to be made strict. Once the argument subterms of these strict operators are visited, the larger term is also marked visited and the process continues until the entire term is marked as visited. The `#visited` label needs to yield `KResult`s in order for this to work, although in practice probably users of the visitor will subsort `#visited`-wrapped terms to their definition's particular `KResult`s.

Code Generation

Here we show the semantics of the code generation constructs, namely of `quote`, `unquote`, `lift`, `eval`. The interesting aspect of our 𝕂 definition of these constructs below is that it is *language independent*. That is, nothing needs to change in the semantics below if new syntactic constructs are added to or removed from Agent.

MODULE QUOTE-UNQUOTE-SYNTAX

Syntax. `lift` and `eval` are strict, where the former takes the resulting value and lifts it into a code value, and the later expects its argument to evaluate to a code value and turns it into its corresponding code, which is consequently evaluated in the current context. `quote` and `unquote` are not strict. The former freezes its argument code into a code value, without evaluating it, except for code appearing as arguments of `unquote`. In fact, `quote` and `unquote` can be nested; a counter keeps track of how many times `quote` appears nested, and only the code which is unquoted the same number of times gets evaluated in the current context. Please refer to languages like Scheme for more details on how these constructs work.

SYNTAX *Exp* ::= `quote` *Exp* [quote]
| `unquote` *Exp* [unquote]
| `lift` *Exp* [lift, strict]
| `eval` *Exp* [eval, strict]

END MODULE

MODULE QUOTE-UNQUOTE

Semantics. We here chose to use the generic visitor pre-defined in 𝕂. A direct definition would be clearer, but although still language-independent it would involve more rules. Additionally, this offers an opportunity to illustrate the power of 𝕂's generic visitors.

Define a visitor parametric in a natural number `N` that applies `quoteit` (defined below) with first argument `N` to `quote` and `unquote` nodes; these nodes are recognized with the predicate `isQuote` (also defined below). We define this visitor as a macro:

SYNTAX *KItem* ::= `qVisitor` (*Int*) [klabel('qVisitor)]

RULE $\dfrac{\texttt{qVisitor}\ (N{:}Int)}{\#visitor(\ \texttt{\#label}\ ('quoteit), \#klist(N),\ \texttt{\#label}\ ('isQuoted), \#klist())}$ [macro]

The `mQuote` macro defined below simply applies the visitor to a given K term. In this particular definition of Agent the K term will always be an expression, but we want our semantics to be as general as possible, so we want it to work also if we add other syntactic categories to our language (e.g., statements):

SYNTAX *Exp* ::= `mQuote` (*K*, *Int*) [klabel('mQuote)]

RULE $\dfrac{\texttt{mQuote}\ (E, N)}{\#visit(E,\ \texttt{qVisitor}\ (N))}$ [macro]

The semantics of `quote E` is defined as follows: visit `E`, starting with counter 0; whenever a nested `quote` construct is encountered, increment the counter and continue; whenever a nested `unquote` construct is encountered, if the counter is 0 then execute the unquoted code, otherwise decrement the counter and continue. The `unquote` construct is expected to produce a code value, otherwise a runtime error will occur if executed (because `unquote` can only occur inside the argument expression of a `quote`). Since the result of applying a visitor to a K term is the visited K term wrapped with label `#visited`, the semantics of the constructs `lift` and `eval` are defined using the `#visited` wrapper:

SYNTAX *KItem* ::= `quoteit` (*Int*, *K*) [klabel('quoteit)]

RULE $\dfrac{\texttt{quote}\ E}{\texttt{mQuote}\ (E, \texttt{0})}$

RULE $\dfrac{\texttt{quoteit}\ (N,\ \texttt{quote}\ E)}{\#visiting.kapp(\ \texttt{\#label}\ ('quote\_),\ \texttt{mQuote}\ (E, N +_{Int} \texttt{1}))}$

RULE $\dfrac{\texttt{quoteit}\ (\texttt{0},\ \texttt{unquote}\ E)}{E}$

RULE $\dfrac{\texttt{quoteit}\ (N,\ \texttt{unquote}\ E)}{\#visiting.kapp(\ \texttt{\#label}\ ('unquote\_),\ \texttt{mQuote}\ (E, N -_{Int} \texttt{1}))}$ requires $(N >_{Int} \texttt{0})$

RULE $\dfrac{\texttt{lift}\ V{:}Val}{\#visited(V)}$

RULE $\dfrac{\texttt{eval}\ \#visited(E{:}K)}{E}$

Since we want code values to become actual values in our language, we also need to explicitly state it that `#visited`-wrapped terms belong to `Val` (the generic visitor only ensures they are `KResult`s):

RULE $\dfrac{isVal(\#visited(\text{—}))}{\texttt{true}}$

Finally, we define the auxiliary predicate testing if a code fragment is a `quote` or `unquote`:

SYNTAX *Bool* ::= `isQuoted` (*Exp*) [function, klabel('isQuoted)]

RULE $\dfrac{\texttt{isQuoted}\ (\ \texttt{quote}\ E)}{\texttt{true}}$

RULE $\dfrac{\texttt{isQuoted}\ (\ \texttt{unquote}\ K)}{\texttt{true}}$

RULE $\dfrac{\texttt{isQuoted}\ (\text{—})}{\texttt{false}}$ [owise]

Conceptually, the above is the conventional definition of `quote/unquote`. However, the definitions that we encountered so far were all language specific; that is, rules propagating the transformations above through each particular language construct were given, ending up with a semantics of `quote/unquote` as large as the size of the language syntax. Note that our semantics is fixed and applies to any language.

END MODULE

MODULE AGENT-SYNTAX

END MODULE

MODULE AGENT

CONFIGURATION:

END MODULE