Core SAIL

February 3, 2022

1 Overview

The SAIL language belongs to the family of Synchronous Reactive Programming (or SRP for short) languages. It is a domain specific language aiming at increasing the reliability of reactive programming, especially in the field of IOT.

In SAIL, reactivity is expressed through parallel composition and signal broadcast. Parallel composition of commands c_1 and c_2 is noted $c_1 \parallel c_2$. It relies on a cooperative scheduling. The execution of a program consists in successive runs named *instants*. A signal named s is declared through the command `signal` s. Its lifetime is the innermost surrounding block of code. It is *emitted* through the command `emit` s. It can also be produced by the environment between instants. Once emitted, a signal is *present* for the duration of the current instant. The execution of a command may be subject to the presence of a signal. This is expressed by the statement `when` s c. For example, the following command prints a message if s is present. Otherwise it is suspended until the signal is emitted or a preemption occurs (see the `watching` construct below)

$$\texttt{pause} = \texttt{when}\ s\ \mathit{print\_string}(\text{"s was here"})$$

The end of an instant occurs when no further progress can be made. Either because the execution is terminated or because all components are waiting for an absent signal. Every component of a program has the opportunity to react to a present signal before the instant terminates, components have a consistent view of their environment. Finally, a command may be preempted at the end of the current instant if it is blocked and if a given signal is present. This is expressed by the statement s `watching` s c which behaves as c but preempts its residual at the end of the instant if s is present. If c terminates during the instant then `watching` s c also terminates. As an example, consider the following statement which will note `pause` in the rest of the document.

$$\texttt{signal}\ s; \texttt{watching}\ s\ \{\texttt{emit}\ s; \texttt{signal}\ s'; \texttt{when}\ s'\ \}$$

The `pause` statement suspends its execution until the end of the instant and resumes at the next instant. Ths signal s' is not emitted and thus the program is suspended because of the `when` statement. As the signal s' is local, it can't be emitted whatever is the context of the

program. At the end of the instant the signal s is present and thus the `watch` statement is preempted.

The following example shows a possible use of this construct.

```
signal s;
signal s';
{
    print_string("A");
    emit s;
    pause;
    emit s'
||
    when s' print_string("B")
||
    watching s
        when s' print_string("C")
}
```

The program prints "A" at the first instant and "B" at the second instant. The message "C" is never printed because the `watching` block is preempted.

2 Syntax

The SAIL language supports usual ground data types (boolean, integers, floating points numbers, characters and strings) as well as compound data types (arrays, structures and enumerations) and generic data types. Types in SAIL are defined by :

$$\tau ::= \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{string} \mid \mathit{array}\langle\tau\rangle \mid \mathit{ref}\langle\tau\rangle \mid \mathit{box}\langle\tau\rangle \mid \mathit{id}\langle\tau,\ldots,\tau\rangle \mid A$$

where $\mathit{array}\langle\tau\rangle$ denotes an array of values of type τ, $\mathit{box}\langle\tau\rangle$ denotes a pointer to a heap allocated value of type τ, $\mathit{ref}\langle\tau\rangle$ denotes a shared reference and $\mathit{id}\langle\tau,\ldots,\tau\rangle$ denotes a user type which is either a structure or an enumeration. A type variable A denotes a generic type. Closed types are types in which no type variable occurs. Structure and enumeration are respectively defined by :

$$\begin{array}{l}\texttt{struct}\ \mathit{id}\lceil\langle A,\ldots,A\rangle\rceil\ \{f:\tau,\ldots,f:\tau\}\\ \texttt{enum}\ \mathit{id}\lceil\langle A,\ldots A\rceil\rangle\ \{f\lceil:(\tau,\ldots,\tau)\rceil,\ldots,f\lceil:(\tau,\ldots,\tau)\rceil\}\end{array}$$

where $\lceil . \rceil$ denotes an optional element. In both cases, the type variables may occur in the types used in the definition. As an example, the type of generic list may be defined in SAIL

by

$$
\begin{array}{l}
\texttt{enum } option < A > \{ \\
\quad None, \\
\quad Some(A) \\
\} \\
\texttt{struct } list < A > \{ \\
\quad head : option\langle ref\langle node\langle A\rangle\rangle\rangle \\
\} \\
\texttt{struct } node\langle A\rangle\{ \\
\quad elem : A, \\
\quad next : option\langle ref\langle node\langle A\rangle\rangle\rangle \\
\}
\end{array}
$$

Expressions of the SAIL language are defined by the following grammar :

$$
\begin{array}{rcl}
e & ::= & \mid x \mid c \mid \ominus e \mid e \oplus e \\
& & \mid [e; \ldots ; e] \mid e[e] \mid \{f : e, \ldots, f : e\} \mid e.f \mid C(e, \ldots, e) \\
& & \mid \&e \mid *e
\end{array}
$$

Variables are names ranging over a finite set and are noted x, y, z and so on. A constant c is a literal denoting a boolean, an integer, a floating-point value, a character or a string. Given a literal c, we note $\widehat{c}$ the corresponding value. Unary and binary operators (respectively $\ominus$ and $\oplus$) are usual operators over integer and boolean value and are assumed to come with semantics functions (noted $\widehat{\ominus}$ and $\widehat{\oplus}$). An expression $[e_0, \ldots, e_{n-1}]$ denotes an array value filled with the values denoted by $e_0, \ldots, e_{n-1}$ and and $e_1[e_2]$ denotes the value at the position denoted by e_2 of such a value denoted by e_1. An expression $\{f_0 : e_0, \ldots, f_{n-1} : e_{n-1}\}$ denotes a structured value filled with values $e_0, \ldots, e_{n-1}$ and $e.f$ denotes the value at position f of such a value denoted by e. An expression $\&e$ denotes the memory location at which the value denoted by e is stored. An expression $*e$ denotes the value stored at the memory location denoted by e.

Commands of Core-SAIL contain usual commands such as variable declaration, sequential composition, conditional, loops and method calls. In Core sail, method calls return no values. Core-SAIL also contains reactive constructs for parallel composition and signal handling. The grammar of commands is given below, followed by an intuitive description for the more significative commands. In next section, we will see that the execution stack will reflect the parallel nature of terms. We choose to distribute the stack over commands (symbols ω). This will be explained latter. For now the symbols $\{c\}_\omega$ can be read as the command c.

$$
\begin{array}{rcl}
c & ::= & \mid \texttt{var x} : \tau \mid \texttt{signal s} \mid \texttt{skip} \mid \texttt{e}_1 = \texttt{e}_2 \mid \texttt{e}_1 = \texttt{ref}(\texttt{e}_2) \\
& & \mid \{c\}_\omega \mid c; c \mid \texttt{if } e\ c\ c \mid \texttt{while } e\ c \mid \texttt{case } e\ \{p : c, \ldots, p : c\} \mid m(e, \ldots, e) \mid \texttt{return} \\
& & \mid \texttt{emit } s \mid \texttt{when } s\ \{c\}_\omega \mid \texttt{watching } s\ \{c\}_\omega \mid \{c\}_\omega \parallel \{c\}_\omega
\end{array}
$$

- $\texttt{var x} : \tau$ and $\texttt{signal } s$ declare respectively a variable x of type τ and a signal s
- $e_1 = e_2$ stores the value of e_1 at the memory location denoted by e_1

- the sequence, conditional, loop commands and method calls behave as usual.
- `case` $e\ \{p : c, \ldots, p : c\}$ performs pattern matching on the value denoted by e. It takes the first pattern p of the list that match the value and behaves as the corresponding command c in an environment augmented by a mapping of the variable of p to the appropriate values.
- `when` $e\ \{c\}_\omega$ behaves as c when the signal s is present. When s is absent the command is suspended.
- `watching` $s\ \{c\}_\omega$ behaves as c but, if s is present at the end of the instant the whole block terminates
- $\{c_1\}_{\omega_1} \parallel \{c_2\}_{\omega_2}$ runs c_1 and c_2 in parallel. Parallel composition terminates when both branches terminate.

Methods and programs The syntax of method declaration is

$$\texttt{method id}\langle \texttt{A}, \ldots, \texttt{A}\rangle(\texttt{x} : \tau, \ldots, \texttt{x} : \tau)\ c$$

A program in Core-SAIL is given by a sequence of declarations of user-defined types and methods. It also provide an entry point which is a single command (which corresponds to the Main process in SAIL). Consider the following example which computes the factorial of 5.

```
method factorial(x : int, y : &int){
    if(x == 0 or x == 1){*y = 1; return}
    else{
        var z : int;
        factorial(x − 1, &z);
        y = x ∗ z;
        return
    }
}

var x : int;
factorial(5, &x);
```

Memory locations We assume a built-in method

$$box\langle A\rangle(x : A, y : \&box\langle A\rangle)$$

which allocates a new memory location, of type $box\langle A\rangle$ which receives the value of x. The new memory location is written in y. Core-SAIL distinguish two kind of memory locations. Those of type $box\langle\tau\rangle$, for some type τ, are memory locations explicitly allocated by the user as in $x = box(1, \&y)$. Memory locations of type $ref\langle\tau\rangle$ are shared references obtained by the & operator.

3 Dynamic semantics

Core SAIL commands execute in the context of an environment and a heap. The environment maps program variables and signal names to memory addresses (noted α possibly with subscripts). The heap maps memory addresses to values and signal states.

Values Values are defined by the following grammar

$$\begin{array}{rcl} \mathit{Value} \ni v & ::= & v_{\texttt{bool}} \mid v_{\texttt{int}} \mid v_{\texttt{float}} \mid v_{\texttt{char}} \mid v_{\texttt{string}} \\ & & \mid A(v,\ldots,v) \mid S(f:v,\ldots,f:v) \mid E_C(v,\ldots,v) \mid \ell \end{array}$$

where given a ground type τ, v_τ is a value of type τ. A value $A(v,\ldots,v)$ is an array, A value $S(f_0 : v_0,\ldots,f_{n-1} : v_{n-1})$ is a structured value with fields $f_0,\ldots v_1$ and a value $C(v_0,\ldots,v_{n-1})$ is an enum value with constructor C. A memory location ℓ is a pair (α, o) where α is a memory address and o is an abstract offset. An abstract offset is a sequence of field names and indices denoting a position in a value.

$$\begin{array}{rcccl} \mathit{Offset} & \ni & o & ::= & \epsilon \mid o.f \mid o.n \\ \mathit{Location} & \ni & \ell & ::= & (\alpha, o) \end{array}$$

We omit ϵ when the sequence is non empty. For exemple the value at location $g.1$ in the value $\{f : 1, g : [4, 5, 6]\}$ is 5.

The update of the value v at offset o with value w is given by $\mathit{update}_V(v, o, w)$ which is defined by:

$$\begin{array}{lcll} \mathit{update}_V(v, \epsilon, w) & = & w & \\ \mathit{update}_V(S(\ldots, f_i{:}v_i, \ldots), f.o, w) & = & S(\ldots, f_i{:}v, \ldots) & v = \mathit{update}_V(v_i, o, w) \\ \mathit{update}_V(A[\ldots, v_i, \ldots], n.o, w) & = & A[\ldots, v, \ldots] & v = \mathit{update}_V(v_i, o, w) \end{array}$$

Stack and heap An environment ρ is a non empty list of mappings from variable to values. The use of a list of mappings instead of a single mapping is motivated by the need for a precise view of the lifetime of variables. The elements of the list represents the nesting of code blocks.

$$\begin{array}{lcll} \omega & \in & \mathit{Variables} \rightharpoonup \mathit{Location} & \text{frame} \\ \rho & = & \overline{\omega} & \text{stack} \\ \sigma & \in & \mathit{Location} \rightharpoonup \mathit{Value}^{\perp} \cup \{\mathbf{0}, \mathbf{1}\} & \text{heap} \end{array}$$

where $\overline{\omega}$ denotes a list $\omega_0,\ldots,\omega_{n-1}$ and $\perp = \mathit{Value} \cup \{\perp\}$ and $\perp$ represents the value of an allocated but uninitialized location. The special values $\mathbf{0}$ and $\mathbf{1}$ represent the state of a signal, absent ($\mathbf{0}$) or present($\mathbf{1}$).

Notations Given a partial function $f : A \rightharpoonup B$, and a value $a \notin dom(f)$, we note $f_{a \rightsquigarrow b}$, the partial function defined by $f_{a \rightsquigarrow b}(a) = b$ and $f_{a \rightsquigarrow b}(x) = f(x)$ for all $x \neq a$. Given a partial function $f : A \rightharpoonup B$, and a value $a \notin dom(f)$, we note $f_{a \mapsto b}$, the partial function defined by $f_{a \mapsto b}(a) = b$ and $f_{a \mapsto b}(x) = f(x)$ for all $x \neq a$. We note $f_{a,c \mapsto b,d}$ for the parallel update of a and b if $a \neq b$. We extends the functional notations to list of functions.

$$
\begin{array}{lcll}
\epsilon(x) & = & \mathit{undef} & \\
(\overline{\omega} \cdot \omega(x)) & = & \begin{cases} \omega(x) & \text{if } x \in dom(\omega) \\ \overline{\omega}(x) & \text{otherwise} \end{cases} & \\
\rho_{x \rightsquigarrow \ell} & = & \overline{\omega} \cdot \omega_{x \rightsquigarrow \ell} & \text{if } \rho = \overline{\omega} \cdot \omega \\
\mathit{enter}(\rho) & = & \rho \cdot \varnothing & \\
\mathit{exit}(\rho \cdot \omega) & = & \mathit{Loc}(\omega) &
\end{array}
$$

where $\varnothing$ denotes the empty mapping and $Loc(\omega)$ is the set of memory location occurring in ω.

3.1 Semantics of expressions

The semantics of expressions is devised in two functions $eval_L \in \mathcal{X} \times \mathcal{E} \times \mathcal{H} \rightharpoonup \mathcal{L}$ and $eval_R \in \mathcal{X} \times \mathcal{E} \times \mathcal{H} \rightharpoonup \mathcal{V}$ which correspond respectively to the evaluation of left-values and right-values.

$$
\begin{array}{lcl}
eval_L(x, \rho, \sigma) & = & (\rho(x), \epsilon) \\
eval_L(*e, \rho, \sigma) & = & \ell \text{ if } eval_R(e, \rho, \sigma) = \ell \\
eval_L(e.f, \rho, \sigma) & = & (\ell, o.f) \text{ if } eval_L(e, \rho, \sigma) = (\ell, o) \\
eval_L(e_1[e_2], \rho, \sigma) & = & (\ell, o.n) \text{ if } eval_L(e_1, \rho, \sigma) = (\ell, o) \text{ and } eval_R(e_2, \rho, \sigma) = n
\end{array}
$$

$$
\begin{array}{lcl}
eval_R(x, \rho, \sigma) & = & \sigma(\rho(x)) \\
eval_R(c, \rho, \sigma) & = & \widehat{c} \\
eval_R(\ominus\, e, \rho, \sigma) & = & \widehat{\ominus}(eval_R(e, \rho, \sigma)) \\
eval_R(e_1 \oplus e_2, \rho, \sigma) & = & \widehat{\oplus}(eval_R(e_1, \rho, \sigma), eval_R(e_2, \rho, \sigma)) \\
eval_R(e.f, \rho, \sigma) & = & v_i \text{ if } eval_R(e, \rho, \sigma) = S(\ldots, f_i : v_i, \ldots), f = f_i \\
eval_R(e_1[e_2], \rho, \sigma) & = & v_i \text{ if } eval_R(e_1, \rho, \sigma) = A(\ldots, v_i, \ldots), eval_R(e_2, \rho, \sigma) = i \\
eval_R([e_0, \ldots, e_{n-1}], \rho, \sigma) & = & A(v_0, \ldots, v_{n-1}) \text{ if } \{eval_R(e_i, \rho, \sigma) = v_i\}_{i=0}^{n-1} \\
eval_R(\{f : e_1; \ldots; f : e_n\}, \rho, \sigma) & = & S(f_0 : v_0, \ldots, f_{n-1} : v_{n-1}) \text{ if } \{eval_R(e_i, \rho, \sigma) = v_i\}_{i=0}^{n-1} \\
eval_R(C(e_0, \ldots, e_n), \rho, \sigma) & = & E_C(v_0, \ldots, v_{n-1}) \text{ if } \{eval_R(e_i, \rho, \sigma) = v_i\}_{i=0}^{n-1} \\
eval_R(\&e, \rho, \sigma) & = & eval_L(e, \rho, \sigma) \\
eval_R(*e, \rho, \sigma) & = & \sigma(\ell) \text{ if } eval_R(e, \rho, \sigma) = \ell
\end{array}
$$

3.2 Semantics of commands

The semantics of commands is divided in several levels. First we have microsteps which are given by rules of the form $c, \rho, \sigma \rightarrow K, \omega, \sigma'$. In a microstep, each component is executed

one time until it is terminated or suspended. The status $K \in \{\mathcal{C}, \mathcal{S}\ c, \mathcal{R}\}$ tells us if the command is terminated ($\mathcal{C}$), suspended with the continuation c to resume ($\mathcal{S}\ c$) or returning control to the caller ($\mathcal{R}$). The frame ω denotes the part of the stack allocated during the micro-steps. Commands containing blocks of the form $\{.\}_\omega$ is a suspened command in which ω was recorded when the command suspends. As mentioned earlier, we use these annotations to distribute the stack over commands.

$$\frac{}{c \equiv c} \qquad \frac{}{\{c_1\}_{\omega_1} \parallel \{c_2\}_{\omega_2} \equiv \{c_2\}_{\omega_2} \parallel \{c_1\}_{\omega_1}}$$

$$\frac{}{K \equiv K} \qquad \frac{c \equiv c'}{\mathcal{S}\ c \equiv \mathcal{S}\ c'}$$

3.3 steps and instants

The execution of an instant is a succession of microsteps. We perform microsteps until the command is either terminated or suspended (predicate *suspended*).If the command is suspended, we perform preemption (function *unlock*) before to run a new instant. The computation of instants is given in figure 3.3.

$$\frac{\mathit{suspended}(c, \rho \cdot \omega, \sigma)}{\mathit{suspended}(\{c\}_\omega, \rho, \sigma)} \qquad \frac{\mathit{suspended}(c_1, \rho, \sigma)}{\mathit{suspended}(c_1; c_2, \rho, \sigma)}$$

$$\frac{\rho(s) = \alpha \qquad \sigma(\alpha) = \mathbf{0} \vee (\sigma(\alpha) = \mathbf{1} \wedge \mathit{suspended}(c, \rho \cdot \omega, \sigma))}{\mathit{suspended}(\texttt{when}\ s\ \{c\}_\omega, \rho, \sigma)}$$

$$\frac{\mathit{suspended}(c, \rho \cdot \omega, \sigma)}{\mathit{suspended}(\texttt{watching}\ s\ \{c\}_\omega, \rho, \sigma)}$$

$$\frac{\mathit{suspended}(c_1, \rho \cdot \omega_1, \sigma) \vee \mathit{suspended}(c_2, \rho \cdot \omega_2, \sigma)}{\mathit{suspended}(c_1\ {}_{\omega_1}\|_{\omega_2}\ c_2, \rho, \sigma)}$$

The function *unlock* terminates preemption blocks if the watched signal is present.

$\mathit{unlock}(\{c\}_\omega, \rho, \sigma)$	$=$	$\{c'\}_\omega$	where $c' = \mathit{unlock}(c, \rho \cdot \omega, \sigma)$
$\mathit{unlock}(c_1; c_2, \rho, \sigma)$	$=$	$c_1'; c_2$	where $c_1' = \mathit{unlock}(c_1, \rho, \sigma)$
$\mathit{unlock}(\texttt{when}\ s\ \{c\}_\omega)$	$=$	$\texttt{when}\ s\ \{c'\}_\omega$	where $c' = \mathit{unlock}(c, \rho \cdot \omega, \sigma)$
$\mathit{unlock}(\texttt{watching}\ s\ \{c\}_\omega)$	$=$	$\texttt{skip}$	if $\rho(s) = \alpha$ and $\sigma(\alpha) = \mathbf{1}$
$\mathit{unlock}(\texttt{watching}\ s\ \{c\}_\omega)$	$=$	$\texttt{watching}\ s\ \{c'\}_\omega$	if $\rho(s) = \alpha$, $\sigma(\alpha) = \mathbf{0}$ and $c' = \mathit{unlock}(c, \rho \cdot \omega, \sigma)$
$\mathit{unlock}(c_1\ {}_{\omega_1}\|_{\omega_2}\ c_2, \rho, \sigma)$	$=$	$c_1'\ {}_{\omega_1}\|_{\omega_2}\ c_2'$	where $c_i' = \mathit{unlock}(c_i, \rho \cdot \omega_i, \sigma),\ i = 1, 2$

We note $(c, \sigma) \to (c', \sigma)$ for $(c, \epsilon, \sigma) \to (c', \varnothing, \sigma')$, $\mathit{suspended}(c, \sigma)$ for $\mathit{suspended}(c, \varnothing, \sigma)$

$$\frac{x \notin dom(\rho) \qquad \alpha \notin dom(\sigma)}{\texttt{var x} : \tau, \rho, \sigma \to \mathcal{C}, [\texttt{x} \rightsquigarrow \alpha], \sigma_{\alpha \rightsquigarrow undef}} \qquad \frac{s \notin dom(\rho) \qquad \alpha \notin dom(\sigma)}{\texttt{signal } s, \rho, \sigma \to \mathcal{C}, [s \rightsquigarrow \alpha], \sigma_{\alpha \rightsquigarrow \mathbf{0}}}$$

$$\frac{}{\texttt{skip}, \rho, \sigma \to \mathcal{C}, \varnothing, \sigma}$$

$$\frac{\begin{array}{c} eval_L(e_1, \rho, \sigma) = (\alpha, o) \qquad eval_R(e_2, \rho, \sigma) = v \\ \sigma(\alpha) = v_\alpha \qquad v' = update_V(v_\alpha, o, v) \end{array}}{e_1 := e_2, \rho, \sigma \to \mathcal{C}, \varnothing, \sigma_{\alpha \mapsto v'}}$$

$$\frac{\begin{array}{c} eval_L(e_1, \rho, \sigma) = (\alpha, o) \qquad eval_R(e_2, \rho, \sigma) = v \\ \sigma(\alpha) = v_\alpha \qquad \alpha' \notin dom(\sigma) \qquad v' = update_V(v_\alpha, p, \alpha') \end{array}}{e_1 := \texttt{box}(e_2), \rho, \sigma \to \mathcal{C}, \varnothing, \sigma_{\alpha' \rightsquigarrow v, \alpha \mapsto v'}}$$

$$\frac{c_1, \rho \cdot \omega, \sigma \to \mathcal{C}, \omega_1, \sigma'' \qquad c_2, \rho \cdot \omega\omega_1, \sigma'' \to \mathcal{K}, \omega_2, \sigma'}{c_1; c_2, \rho \cdot \omega, \sigma \to \mathcal{K}, \omega_1\omega_2, \sigma'}$$

$$\frac{c_1, \rho, \sigma \to \mathcal{S}\ c_1', \omega, \sigma'}{c_1; c_2, \rho, \sigma \to \mathcal{S}\ c_1'; c_2, \omega, \sigma'} \qquad \frac{c_1, \rho, \sigma \to \mathcal{R}, \omega, \sigma'}{c_1; c_2, \rho, \sigma \to \mathcal{R}, \omega, \sigma'}$$

$$\frac{c, \rho \cdot \omega, \sigma \to \mathcal{S}\ c', \omega', \sigma'}{\{c\}_\omega, \rho, \sigma \to \mathcal{S}\ \{c'\}_{\omega\omega'}, [\ ], \sigma'}$$

$$\frac{c, \rho \cdot \omega, \sigma \to \mathcal{K}, \omega', \sigma' \qquad \omega\omega' \not\downarrow \overline{\alpha} \qquad \mathcal{K} \in \mathcal{C}, \mathcal{R} \qquad \sigma'' = drop(\sigma', \overline{\alpha})}{\{c\}_\omega, \rho, \sigma \to \mathcal{K}, \varnothing, \sigma''}$$

$$\frac{eval_R(e, \rho, \sigma) = \texttt{true} \qquad \{c_1\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}{\texttt{if } e\ \{c_1\}\ \{c_2\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}$$

$$\frac{eval_R(e, \rho, \sigma) = \texttt{false} \qquad \{c_2\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}{\texttt{if } e\ \{c_1\}\ \{c_2\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}$$

$$\frac{eval_R(e, \rho, \sigma) = \texttt{true} \qquad \{c\}_\varnothing; \texttt{while } e\ c, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}{\texttt{while } e\ c, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}$$

$$\frac{eval_R(e, \rho, \sigma) = \texttt{false}}{\texttt{while } e\ c, \rho, \sigma \to \mathcal{C}, \varnothing, \sigma}$$

Figure 1: Microsteps

$$\frac{\begin{array}{c} eval_R(e_i, \rho, \sigma) \Downarrow v_i \qquad 0 \le i \le n-1 \\ \texttt{method}\ m\langle A_0, \ldots, A_{m-1}\rangle(x_0 : \tau_0, \ldots, x_{n-1} : \tau_{n-1})\lceil : \tau \rceil\ c \in \mathit{Methods} \\ \{c\}_{[x_i \mapsto v_i]_{i=0}^{n-1}}, \epsilon, \sigma \to \mathcal{R}(v), \omega, \sigma' \end{array}}{m(e_0, \ldots, e_{n-1}), \rho, \sigma \to \mathcal{C}, \omega, \sigma'}$$

$$\frac{\begin{array}{c} eval_R(e, \rho, \sigma) = v \qquad filter(p, v) = \mathit{undef} \\ \texttt{case}\ e\ \{\overline{p : c}\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma' \end{array}}{\texttt{case}\ e\ \{p : c, \overline{p : c}\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}$$

$$\frac{\begin{array}{c} eval_R(e, \rho, \sigma) = v \qquad filter(p, v) = [(x_i, v_i)]_{i=0}^{n-1} \\ \{\ell_i\}_{i=0}^{n-1} \cap \mathit{dom}(\sigma) = \emptyset \\ \{c\}_{[x_i \mapsto \ell_i]_{i=0}^{n-1}}, \rho, \sigma_{\{\ell_i, \mapsto v_i\}_{i=0}^{n-1}} \to \mathcal{K}, \omega, \sigma' \end{array}}{\texttt{case}\ e\ \{p : c, \overline{p : c}\}, \rho, \sigma \to \mathcal{K}, \omega, \sigma'}$$

Figure 2: Microsteps

and $\mathit{unlock}(c, \sigma)$ for $\mathit{unlock}(c, \varnothing, \sigma)$.

$$\frac{(c, \sigma) \to (\mathcal{C}, \sigma')}{(c, \sigma) \Rightarrow \sigma'} \qquad \frac{(c, \sigma) \to (\mathcal{S}\ c', \sigma') \qquad \mathit{suspended}(c', \sigma')}{(c, \sigma) \Rightarrow (c', \sigma')}$$

$$\frac{(c, \sigma) \to (\mathcal{S}\ c', \sigma') \qquad \neg \mathit{suspended}(c', \sigma') \qquad (c', \sigma') \Rightarrow (c'', \sigma'')}{(c, \sigma) \Rightarrow (c'', \sigma'')}$$

Execution of a program Given a command c, the initial state is $\{c\}_{\varnothing}, \epsilon, h_{\emptyset}$

$$\frac{\rho(s) = \ell \qquad \sigma(\ell) = \mathbf{1} \qquad c, \rho \cdot \varnothing, \sigma, \to \mathcal{K}, \omega', \sigma' \qquad \mathcal{K} \in \mathcal{C}, \mathcal{R} \qquad \omega\omega' \not\downarrow \overline{\alpha} \qquad \sigma'' = \mathit{drop}(\omega, \overline{\alpha})}{\texttt{when}\ s\ \{c\}_\omega, \rho, \sigma \to \mathcal{K}, \varnothing, \sigma''}$$

$$\frac{\rho(s) = \ell \qquad \sigma(\ell) = \mathbf{1} \qquad c, \rho \cdot \omega, \sigma \to \mathcal{S}\ c', \omega', \sigma'}{\texttt{when}\ s\ \{c\}_\omega, \rho, \sigma \to \mathcal{S}\ \texttt{when}\ s\ \{c'\}_{\omega\omega'}, \varnothing, \sigma'}$$

$$\frac{\rho(s) = \ell \qquad \sigma(\ell) = \mathbf{0}}{\texttt{when}\ s\ \{c\}_\omega, \rho, \sigma \to \mathcal{S}\ \texttt{when}\ s\ \{c\}_\omega, \varnothing, \sigma}$$

$$\frac{c, \rho \cdot \omega, \sigma \to \mathcal{K}, \omega', \sigma' \qquad \mathcal{K} \in \mathcal{C}, \mathcal{R} \qquad \omega\omega' \not\downarrow \overline{\alpha} \qquad \sigma'' = \mathit{drop}(\omega, \overline{\alpha})}{\texttt{watching}\ s\ \{c\}_\omega, \rho, \sigma \to \mathcal{K}, \varnothing, \sigma''}$$

$$\frac{c, \rho \cdot \omega, \sigma \to \mathcal{S}\ c', \omega', \sigma'}{\texttt{watching}\ s\ \{c\}_\omega, \rho, \sigma \to \mathcal{S}\ \texttt{watching}\ s\ \{c'\}_{\omega\omega'}, \varnothing, \sigma'}$$

$$\frac{c_1, \rho \cdot \omega_1, \sigma \to \mathcal{C}, \omega_1', \sigma'' \qquad c_2, \rho \cdot \omega_2, \sigma'' \to \mathcal{C}, \rho \cdot \omega_2', \sigma''' \qquad \omega_1\omega_2\omega_1'\omega_2' \not\downarrow \overline{\alpha} \qquad \sigma'' = \mathit{drop}(\sigma, \overline{\alpha})}{\{c_1\}_{\omega_1} \parallel \{c_2\}_{\omega_2}, \rho, \sigma \to \mathcal{C}, \varnothing, \sigma''}$$

$$\frac{c_1, \rho \cdot \omega_1, \sigma \to \mathcal{S}(c_1'), \omega_1', \sigma'' \qquad c_2, \rho \cdot \omega_2, \sigma'' \to \mathcal{S}(c_2'), \omega_2', \sigma'}{\{c_1\}_{\omega_1} \parallel \{c_2\}_{\omega_2}, \rho, \sigma \to \mathcal{S}\ \{c_1\}_{\omega_1\omega_1'} \parallel \{c_2\}_{\omega_2\omega_2'}, \varnothing, \sigma'}$$

$$\frac{\{c_1\}, \rho \cdot \omega_1, \sigma \to \mathcal{S}(c_1'), \omega_1', \sigma'' \qquad \{c_2\}, \rho \cdot \omega_2, \sigma'' \to \mathcal{C}, \omega_2', \sigma'''}{\{c_1\}_{\omega_1} \parallel \{c_2\}_{\omega_2}, \rho, \sigma \to \mathcal{S}\ \{c_1'\}_{\omega_1\omega_1'} \parallel \{\texttt{skip}\}_{\omega_2\omega_2'}, \varnothing, \sigma'}$$

$$\frac{c \equiv c' \qquad c', \rho, \sigma \to K', \omega, \sigma' \qquad K' \equiv K}{c, \rho, \sigma \to K, \omega, \sigma'}$$

Figure 3: Microsteps

$$\frac{(c, \sigma) \Rightarrow \sigma'}{(c, \sigma) \Longmapsto \sigma'} \qquad \frac{(c, \sigma) \Rightarrow (c', \sigma')}{(c, \sigma) \Longmapsto (c', \sigma')}$$

$$\frac{(c, \sigma) \Longmapsto (c', \sigma') \qquad \mathit{unlock}(c', \sigma) = c'' \qquad (c'', \sigma') \Longmapsto (c''', \sigma')}{(c, \sigma) \Longmapsto (c''', \sigma')}$$

Figure 4: Instants