Appendix of DimSum: A Decentralized Approach to Multi-language Semantics and Verification

MICHAEL SAMMLER, MPI-SWS, Germany
SIMON SPIES, MPI-SWS, Germany
YOUNGJU SONG, MPI-SWS, Germany
EMANUELE D'OSUALDO, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEEPAK GARG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

A COMBINATORS

Filter. The filter combinator $M \mid_\sigma M' \triangleq (S_{\mathsf{filter}} \times S_M \times S_{M'}, \rightarrow_{\mathsf{filter}}, (\sigma, \sigma^0_{M'}, \sigma^0_{M'}))$ takes in a module $M \in \mathsf{Module}(E_1)$ and a filter $M' \in \mathsf{Module}(\mathsf{FilterEvents}(E_1, E_2))$ and then produces a module with events drawn from E_2. The states of the filter combinator are given by $S_{\mathsf{filter}} \triangleq \{\mathsf{P}, \mathsf{F}\} \cup \{\mathsf{P}(e) \mid e \in E_1\} \cup \{\mathsf{F}(e) \mid e \in E_1\}$ and the transitions are depicted in Fig. 1. The events of the filter module are drawn from the set

$$\mathsf{FilterEvents}(E_1, E_2) \triangleq \{\mathsf{Receive}(e_1) \mid e_1 \in E_1\} \cup \{\mathsf{Emit}(e_2) \mid e_2 \in E_2\} \cup \{\mathsf{Return}(e_1) \mid e_1 \in \mathsf{option}(E_1)\}$$

where $\mathsf{Receive}(e_1)$ is for accepting an incoming event, $\mathsf{Emit}(e_2)$ is for emitting an outgoing event, and $\mathsf{Return}(e_1)$ is for returning control to the inner module. In the last case, the event e_1 is optional to allow us to force the inner module to emit the event e_1 next (*i.e.*, all visible transitions of the inner module except ones emitting event e_1 are blocked).

Linking. The linking operator $M_1 \oplus_X M_2$ is defined on modules $M_1, M_1 \in \mathsf{Module}(E_{?!})$ where $E_{?!}$ is (an event type that is isomorphic to) $E \times \{?, !\}$. The parameter $X = (S, \rightsquigarrow, s^0)$ determines how the events are linked. It consists of a set of linking-interal states S, an initial state $s^0 \in S$, and a relation $\rightsquigarrow\ \subseteq (\mathsf{D} \times S \times E) \times ((\mathsf{D} \times S \times E) \cup \{\lightning\})$ describing how events should be translated. Formally, linking can be defined as $M_1 \oplus_X M_2 \triangleq M_1 \times M_2 \mid_{\mathsf{P}} \mathsf{link}_X$.[1] The module link_X is defined as $\mathsf{link}_X \triangleq (S_{\mathsf{link}} \times S_X, \rightarrow_{\mathsf{link}}, (\mathsf{Wait}, s^0_X))$ where $S_{\mathsf{link}} \triangleq (\{\mathsf{Wait}, \mathsf{Ub}\} \cup \{\mathsf{Emit}(e, \sigma) \mid e \in E_{?!}, \sigma \in S_{\mathsf{link}}\} \cup \{\mathsf{Return}(e) \mid e \in \mathsf{option}(E_{?!})\})$ and $\rightarrow_{\mathsf{link}}$ is defined in Fig. 2.

(Kripke) wrappers. The combinator $\lceil M \rceil_X$ translates a modules with events E_1 to a module with events E_2. This combinator is parametrized by $X = (S, R, \leftharpoondown, \rightharpoonup, s^0, F^0)$ where S is a set of states and s^0 is an initial state. These states were omitted in the main paper for simplicity. They don't give additional expressive power but make writing the wrapper $\lceil \cdot \rceil_{\mathsf{r} \rightleftharpoons \mathsf{a}}$ more pleasant. $\leftharpoondown$ and $\rightharpoonup$ are relations that describe how the wrapper transforms the incoming and outgoing events. Concretely, $\leftharpoondown$ describes how to translate an event $e_2 \in E_2$ to an event $e_1 \in E_1$ and $\rightharpoonup$ describes the translation from $e'_1 \in E_1$ to $e'_2 \in E_2$.

[1] The Coq development defines linking via more low-level combinators that we omit from the presentation here. Also the Coq development allows undefined behavior via a Boolean on the right side of $\rightsquigarrow$ instead of a separate $\lightning$ result.

Authors' addresses: Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Youngju Song, MPI-SWS, Saarland Informatics Campus, Germany, youngju@mpi-sws.org; Emanuele D'Osualdo, MPI-SWS, Saarland Informatics Campus, Germany, dosualdo@mpi-sws.org; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.

FILTER-STEP-PROG-NONE
$$\frac{\sigma = \mathsf{P} \vee \sigma = \mathsf{P}(e) \qquad \sigma_1 \xrightarrow{\tau} \Sigma}{(\sigma, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\sigma, \sigma_1', \sigma_2) \,\middle|\, \sigma_1' \in \Sigma\right\}}$$

FILTER-STEP-FILTER-NONE
$$\frac{\sigma = \mathsf{F} \vee \sigma = \mathsf{F}(e) \qquad \sigma_2 \xrightarrow{\tau} \Sigma}{(\sigma, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\sigma, \sigma_1, \sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}}$$

FILTER-STEP-PROG-RECV
$$\frac{\sigma_1 \xrightarrow{e} \Sigma}{(\mathsf{P}(e), \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\mathsf{P}, \sigma_1', \sigma_2) \,\middle|\, \sigma_1' \in \Sigma\right\}}$$

FILTER-STEP-PROG
$$\frac{\sigma_1 \xrightarrow{e} \Sigma}{(\mathsf{P}, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\mathsf{F}(e), \sigma_1', \sigma_2) \,\middle|\, \sigma_1' \in \Sigma\right\}}$$

FILTER-STEP-FILTER-RECV
$$\frac{\sigma_2 \xrightarrow{\mathsf{Receive}(e)} \Sigma}{(\mathsf{F}(e), \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\mathsf{F}, \sigma_1, \sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}}$$

FILTER-STEP-FILTER-EMIT
$$\frac{\sigma_2 \xrightarrow{\mathsf{Emit}(e)} \Sigma}{(\mathsf{F}, \sigma_1, \sigma_2) \xrightarrow{e}_{\mathsf{filter}} \left\{(\mathsf{F}, \sigma_1, \sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}}$$

FILTER-STEP-FILTER-RETURN
$$\frac{\sigma_2 \xrightarrow{\mathsf{Return}(e)} \Sigma}{(\mathsf{F}, \sigma_1, \sigma_2) \xrightarrow{\tau}_{\mathsf{filter}} \left\{(\text{if } e = \mathsf{Some}(e') \text{ then } \mathsf{P}(e') \text{ else } \mathsf{P}, \sigma_1, \sigma_2') \,\middle|\, \sigma_2' \in \Sigma\right\}}$$

Fig. 1. Definition of $\rightarrow_{\mathsf{filter}}$.

$$\mathsf{to}(d, e) = \begin{cases} \mathsf{Return}(\mathsf{left}(e?, \mathsf{L})) & \text{if } d = \mathsf{L} \\ \mathsf{Return}(\mathsf{right}(e?, \mathsf{R})) & \text{else if } d = \mathsf{R} \\ \mathsf{Emit}(e!, \mathsf{Return}(\mathsf{None})) & \text{else if } d = \mathsf{E} \end{cases}$$

LINK-STEP-WAIT-L
$$\frac{(\mathsf{L}, s, e) \rightsquigarrow (d, s', e')}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{left}(e!, d))}_{\mathsf{link}} \{(\mathsf{to}(d, e'), s')\}}$$

LINK-STEP-WAIT-L-UB
$$\frac{(\mathsf{L}, s, e) \rightsquigarrow \lightning}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{left}(e!, d))}_{\mathsf{link}} \{(\mathsf{Ub}, s)\}}$$

LINK-STEP-WAIT-R
$$\frac{(\mathsf{R}, s, e) \rightsquigarrow (d, s', e')}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{right}(e!, d))}_{\mathsf{link}} \{(\mathsf{to}(d, e'), s')\}}$$

LINK-STEP-WAIT-R-UB
$$\frac{(\mathsf{R}, s, e) \rightsquigarrow \lightning}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{right}(e!, d))}_{\mathsf{link}} \{(\mathsf{Ub}, s)\}}$$

LINK-STEP-WAIT-N
$$\frac{(\mathsf{E}, s, e) \rightsquigarrow (d, s', e')}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{env}(d))}_{\mathsf{link}} \{(\mathsf{Emit}(e'?, \mathsf{to}(d, e')), s')\}}$$

LINK-STEP-WAIT-N-UB
$$\frac{(\mathsf{E}, s, e) \rightsquigarrow \lightning}{(\mathsf{Wait}, s) \xrightarrow{\mathsf{Receive}(\mathsf{env}(d))}_{\mathsf{link}} \{(\mathsf{Ub}, s)\}}$$

LINK-STEP-EMIT
$$(\mathsf{Emit}(e, \sigma), s) \xrightarrow{\mathsf{Emit}(e)}_{\mathsf{link}} \{(\sigma, s)\}$$

LINK-STEP-RETURN
$$(\mathsf{Return}(e), s) \xrightarrow{\mathsf{Return}(e)}_{\mathsf{link}} \{(\mathsf{Wait}, s)\}$$

LINK-STEP-UB
$$(\mathsf{Ub}, s) \xrightarrow{\tau}_{\mathsf{link}} \emptyset$$

Fig. 2. Definition of $\rightarrow_{\mathsf{link}}$.

As mentioned in the paper, these relations are separation logic relations. As such, they are of type $E_1 \times S \times E_2 \times S \longrightarrow \mathit{UPred}(R)$. The second component of X is a resource algebra R [Jung et al. 2018] that determines a separation logic of uniform predicates over the resource algebra $\mathit{UPred}(R)$. F^0 denotes the initial set of resources owned by the wrapper. We define $\lceil M \rceil_X \triangleq M \mathrel{|\!\!\models_{\mathsf{F}}} [\![\mathsf{state}(s^0, F^0)]\!]_{\mathsf{S}}$

$$\mathbf{Instr} \ni \mathbf{c} \triangleq \mathsf{syscall}; \mathbf{c} \mid \mathsf{upd}(\mathbf{x}, \mathbf{r}.\ \mathbf{v}); \mathbf{c} \mid \mathsf{ldr}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{v}.\ \mathbf{v}'); \mathbf{c} \mid \mathsf{str}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{v}.\ \mathbf{v}'); \mathbf{c} \mid \mathsf{jump}$$

Fig. 3. Micro-Instructions of Asm

where the filter module is given by the following Spec program:[2]

$$\begin{aligned}
&\mathsf{state}(s_2, F_2) \triangleq_{\mathsf{coind}} \\
&\quad \exists e_2; \mathsf{vis}(\mathsf{Emit}(e_2)); \forall e_1, s_1, F_1; \mathsf{assume}(\mathsf{sat}(F_1 * F_2 * (e_1, s_1) \leftharpoondown (e_2, s_2))); \mathsf{vis}(\mathsf{Return}(e_1)); \\
&\quad \exists e_1'; \mathsf{vis}(\mathsf{Receive}(e_1')); \exists e_2', s_2', F_2'; \mathsf{assert}(\mathsf{sat}(F_1 * F_2' * (e_1', s_1) \rightharpoonup (e_2', s_2'))); \mathsf{vis}(\mathsf{Emit}(e_2')); \\
&\quad \mathsf{state}(s_2', F_2')
\end{aligned}$$

Intuitively, $\mathsf{state}(s_2, F_2)$ works as follows: Given an initial state s_2 and a proposition describing resource ownership of the translation F_2, state synchronizes with the environment on an event e_2. Then it angelically chooses an event e_1 for the inner module, a new state s_1, ownership of the environment F_1, and a proof that the ownership of the translation together with the ownership of the environment and the precondition $(e_1, s_1) \leftharpoondown (e_2, s_2)$ is satisfiable. Then state sends e_1 to the inner module M. Next, it receives an event e_1' from M and (demonically) chooses an event e_2' to emit to the environment, a new state s_2', new ownership of the translation F_2', and a proof that the ownership of the translation together with the ownership of the environment and the postcondition $(e_1', s_1) \rightharpoonup (e_2', s_2')$ is satisfiable. After emitting e_2', the process repeats with state s_2' and F_2'.

B MICRO-INSTRUCTIONS OF Asm

Inspired by Sammler et al. [2022a], instructions $\mathbf{c}$ in Asm are sequences of *micro instructions* (*i.e.*, simple instructions that, when composed together, form an actual instruction), depicted in Fig. 3. The instruction $\mathsf{syscall}; \mathbf{c}$ does a syscall and then executes $\mathbf{c}$. The instruction $\mathsf{upd}(\mathbf{x}, \mathbf{r}.\ \mathbf{v}); \mathbf{c}$ updates the register $\mathbf{x}$ according to the map $\mathbf{r} \mapsto \mathbf{v}$ applied to the current register values $\mathbf{r}$ and then executes $\mathbf{c}$. The instruction $\mathsf{ldr}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{v}.\ \mathbf{v}'); \mathbf{c}$ takes the value stored in $\mathbf{x}_2$, applies the transformation $\mathbf{v} \mapsto \mathbf{v}'$ to it to obtain an address, loads from the memory at that address, stores the result in $\mathbf{x}_1$, and then executes $\mathbf{c}$. The instruction $\mathsf{ldr}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{v}.\ \mathbf{v}'); \mathbf{c}$ takes the value stored in $\mathbf{x}_2$, applies the transformation $\mathbf{v} \mapsto \mathbf{v}'$ to it to obtain an address, stores in the memory at that address the value in $\mathbf{x}_1$, and then executes $\mathbf{c}$. The instruction jump reads the $\mathbf{pc}$ register and then jumps to the address stored there.

The reason for the micro instruction representation is that we can represent a large instruction set by chaining few primitives. For example, the instructions used in print are derived as follows:

$$\mathsf{ret} \triangleq \mathsf{upd}(\mathbf{pc}, \mathbf{r}.\ \mathbf{r}(\mathbf{x30})); \mathsf{jump} \qquad \mathsf{syscall} \triangleq \mathsf{syscall}; \mathsf{next} \qquad \mathsf{mov}\ \mathbf{x},\ \mathbf{v} \triangleq \mathsf{upd}(\mathbf{x}, \mathbf{r}.\ \mathbf{v}); \mathsf{next}$$

$$\mathsf{sle}\ \mathbf{x}_1,\ \mathbf{x}_2,\ \mathbf{x}_3 \triangleq \mathsf{upd}(\mathbf{x}_1, \mathbf{r}.\ \mathsf{if}\ \mathbf{r}(\mathbf{x}_2) \le \mathbf{r}(\mathbf{x}_3)\ \mathsf{then}\ 1\ \mathsf{else}\ 0); \mathsf{next}$$

where we abbreviate $\mathsf{next} \triangleq \mathsf{upd}(\mathbf{pc}, \mathbf{r}.\ \mathbf{r}(\mathbf{pc}) + 1); \mathsf{jump}$.

C SEMANTIC LINKING FOR Asm

The full definition of the semantic linking relation $\rightsquigarrow$ for Asm can be found in Fig. 4. Compared to the excerpt shown in the paper, it contains two additional cases, ASM-LINK-SYSCALL and ASM-LINK-SYSCALL-RETURN. The rule ASM-LINK-SYSCALL makes sure syscalls are passed on to the environment (and never come from the environment). When a syscall is triggered, we store the current turn d in

[2]The Coq development defines an equivalent module directly using a step relation, but we give the definition here using Spec for readability.

$$\textsc{asm-link-jump}\quad \frac{(d' = \mathsf{L} \wedge \mathbf{r}(\mathbf{pc}) \in \mathbf{d}_1) \vee (d' = \mathsf{R} \wedge \mathbf{r}(\mathbf{pc}) \in \mathbf{d}_2) \vee (d' = \mathsf{E} \wedge \mathbf{r}(\mathbf{pc}) \notin \mathbf{d}_1 \cup \mathbf{d}_2) \qquad d \neq d'}{(d, \mathsf{None}, \mathbf{Jump}(\mathbf{r}, \mathbf{m})) \rightsquigarrow_{\mathbf{d}_1,\mathbf{d}_2} (d', \mathsf{None}, \mathbf{Jump}(\mathbf{r}, \mathbf{m}))}$$

$$\textsc{asm-link-syscall}\quad \frac{d \neq \mathsf{E}}{(d, \mathsf{None}, \mathbf{Syscall}(\mathbf{v}_1, \mathbf{v}_2, \mathbf{m})) \rightsquigarrow_{\mathbf{d}_1,\mathbf{d}_2} (\mathsf{E}, \mathsf{Some}(d), \mathbf{Syscall}(\mathbf{v}_1, \mathbf{v}_2, \mathbf{m}))}$$

$$\textsc{asm-link-syscall-return}\quad \frac{d' \neq \mathsf{E}}{(\mathsf{E}, \mathsf{Some}(d'), \mathbf{SyscallRet}(\mathbf{v}, \mathbf{m})) \rightsquigarrow_{\mathbf{d}_1,\mathbf{d}_2} (d', \mathsf{None}, \mathbf{SyscallRet}(\mathbf{v}, \mathbf{m}))}$$

Fig. 4. Definition of semantic linking relation $\rightsquigarrow$ for Asm.

$$\begin{aligned}
\mathsf{Library} \ni \mathsf{R} &\triangleq (\mathsf{fn}\ \mathsf{f}(\overline{x}) \triangleq \overline{\mathsf{local}\ y[n]}; \mathsf{e}), \mathsf{R} \mid \emptyset \\
\mathsf{Expr} \ni \mathsf{e} &\triangleq \mathsf{v} \mid x \mid \mathsf{e}_1 \oplus \mathsf{e}_2 \mid \mathsf{let}\ x := \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 \mid \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ \mathsf{e}_2\ \mathsf{else}\ \mathsf{e}_3 \mid \mathsf{e}_1(\overline{\mathsf{e}_2}) \mid !\mathsf{e} \mid \mathsf{e}_1 \leftarrow \mathsf{e}_2 \\
\mathsf{BinOp} \ni \oplus &\triangleq + \mid < \mid == \mid \leq \\
\mathsf{Runtime\ Expr} \ni \mathsf{E} &\triangleq \cdots \mid \mathsf{alloc_frame}\ \overline{(x, n)}\ \mathsf{E} \mid \mathsf{free_frame}\ (\overline{\ell, n})\ \mathsf{E} \mid \mathsf{Ret}(b, \mathsf{E}) \mid \mathsf{Wait}(b)
\end{aligned}$$

Fig. 5. Grammar of Rec.

the private state of the linking operator. This way, we can make sure that when we return from a syscall (asm-link-syscall-return), the execution continues with the module that triggered the syscall.

D Rec

The language Rec is a simple, high-level language with arithmetic operations, let bindings, memory operations, conditionals, and (potentially recursive) function calls (depicted in Fig. 5). The libraries R of Rec are lists of function declarations. Each function declaration contains the name of the function f, the argument names $\overline{x}$, local variables $\overline{y}$ which are allocated in the memory, and a function body e. The set of function names $|\mathsf{R}|$ of a library R is defined as the names of the functions in the list R.

Module semantics. The semantics of a Rec library R is the module $[\![\mathsf{R}]\!]_{\mathsf{r}}$. The states of the module are of the from $\sigma = (\mathsf{E}, \mathsf{m}, \mathsf{R})$ where E is the current *runtime expression* (explained below). We write $(\rightarrow_{\mathsf{r}})$ for the transition system (shown in Fig. 6) and the initial state is $(\mathsf{Wait}(\mathsf{false}), \emptyset, \mathsf{R})$.

To define the transition relation $\rightarrow_{\mathsf{r}}$, we extend the static expressions e to runtime expressions E, which have operations for allocating and deallocating stack frames as well as two distinguished expressions $\mathsf{Ret}(b, \mathsf{E})$ and $\mathsf{Wait}(b)$. These expressions are used to control when the module emits call and return events: Initially, the module is waiting and willing to accept any incoming call to the functions of the library (see rec-start). Once it starts, the function call is wrapped in the $\mathsf{Ret}(b, \cdot)$ expression to ensure an event is emitted after the function finishes executing (see rec-ret-return). A call to functions of the library (see rec-call-internal), will trigger the allocation of the local variables and, subsequently, the execution of the function body. A call to an external function (see rec-call-external) will emit a $\mathsf{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})$ and proceed to the waiting state. The flag for the

REC-BINOP
$$(\mathsf{v}_1 \oplus \mathsf{v}_2, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \{(\mathsf{v}, \mathsf{m}, \mathsf{R}) \mid \mathsf{eval}_{\oplus}(\mathsf{v}_1, \mathsf{v}_2, \mathsf{v})\}$$

REC-LOAD
$$(!\mathsf{v}_1, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \{(\mathsf{v}_2, \mathsf{m}, \mathsf{R}) \mid \exists \ell.\ \mathsf{v}_1 = \ell \wedge \mathsf{m}(\ell) = \mathsf{v}_2\}$$

REC-STORE
$$(\mathsf{v}_1 \leftarrow \mathsf{v}_2, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \{(\mathsf{v}_2, \mathsf{m}[\ell \mapsto \mathsf{v}_2], \mathsf{R}) \mid \exists \ell.\ \mathsf{v}_1 = \ell \wedge \mathsf{heap\_alive}(\mathsf{m}, \ell)\}$$

REC-IF
$$(\mathsf{if}\ \mathsf{v}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \{(\mathsf{e}, \mathsf{m}, \mathsf{R}) \mid \exists b.\ \mathsf{v} = b \wedge \mathsf{if}\ b\ \mathsf{then}\ \mathsf{e} = \mathsf{e}_1\ \mathsf{else}\ \mathsf{e} = \mathsf{e}_2\}$$

REC-LET
$$(\mathsf{let}\ x := \mathsf{v}\ \mathsf{ine}, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \{(\mathsf{e}[\mathsf{v}/x], \mathsf{m}, \mathsf{R})\}$$

REC-VAR
$$(x, \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \emptyset$$

REC-ALLOC
$$\frac{\mathsf{heap\_alloc\_list}(\overline{n}, \overline{\ell}, \mathsf{m}_1, \mathsf{m}_2)}{(\mathsf{alloc}\ \overline{(y, n)}\ \mathsf{e}, \mathsf{m}_1, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \left\{(\mathsf{free\_frame}\ \overline{(\ell, n)}\ (\mathsf{e}[\overline{\ell}/\overline{y}]), \mathsf{m}_2, \mathsf{R}) \,\middle|\, \forall m \in \overline{n}.\ m > 0\right\}}$$

REC-FREE
$$(\mathsf{free\_frame}\ \overline{(\ell, n)}\ \mathsf{v}, \mathsf{m}_1, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \left\{(\mathsf{v}, \mathsf{m}_2, \mathsf{R}) \,\middle|\, \mathsf{heap\_free\_list}(\overline{(\ell, n)}, \mathsf{m}_1, \mathsf{m}_2)\right\}$$

REC-START
$$\frac{\mathsf{f} \in \mathsf{R}}{(\mathsf{Wait}(b), \mathsf{m}, \mathsf{R}) \xrightarrow{\mathsf{Call?}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}')}_{\mathsf{r}} \{(\mathsf{Ret}(b, \mathsf{f}(\overline{\mathsf{v}})), \mathsf{m}', \mathsf{R})\}}$$

REC-CALL-INTERNAL
$$\frac{(\mathsf{fn}\ \mathsf{f}(\overline{x}) \triangleq \overline{\mathsf{local}\ y[n]};\ \mathsf{e}) \in \mathsf{R}}{(\mathsf{f}(\overline{\mathsf{v}}), \mathsf{m}, \mathsf{R}) \xrightarrow{\tau}_{\mathsf{r}} \left\{(\mathsf{alloc}\ \overline{(y, n)}\ (\mathsf{e}[\overline{\mathsf{v}}/\overline{x}]), \mathsf{m}, \mathsf{R}) \,\middle|\, |\overline{x}| = |\overline{\mathsf{v}}|\right\}}$$

REC-CALL-EXTERNAL
$$\frac{\mathsf{f} \notin \mathsf{R}}{(\mathsf{f}(\overline{\mathsf{v}}), \mathsf{m}, \mathsf{R}) \xrightarrow{\mathsf{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})}_{\mathsf{r}} \{(\mathsf{Wait}(\mathsf{true}), \mathsf{m}, \mathsf{R})\}}$$

REC-RET-INCOMING
$$(\mathsf{Wait}(\mathsf{true}), \mathsf{m}, \mathsf{R}) \xrightarrow{\mathsf{Return?}(\mathsf{v}, \mathsf{m}')}_{\mathsf{r}} \{(\mathsf{v}, \mathsf{m}', \mathsf{R})\}$$

REC-RET-RETURN
$$(\mathsf{Ret}(b, \mathsf{v}), \mathsf{m}, \mathsf{R}) \xrightarrow{\mathsf{Return!}(\mathsf{v}, \mathsf{m})}_{\mathsf{r}} \{(\mathsf{Wait}(b), \mathsf{m}, \mathsf{R})\}$$

REC-EVAL-CTX
$$\frac{(\mathsf{E}, \mathsf{m}, \mathsf{R}) \xrightarrow{e}_{\mathsf{r}} \Sigma}{(\mathsf{K}[\mathsf{E}], \mathsf{m}, \mathsf{R}) \xrightarrow{e}_{\mathsf{r}} \{(\mathsf{K}[\mathsf{E}'], \mathsf{m}', \mathsf{R}') \mid (\mathsf{E}', \mathsf{m}', \mathsf{R}') \in \Sigma\}}$$

Fig. 6. Operational semantics of Rec.

waiting becomes true, because the module is now willing to accept a return to the function call that was just issued (see REC-RET-INCOMING). The language Rec is an evaluation-context based language,

$$\frac{(d' = \mathsf{L} \land \mathsf{f} \in \mathsf{d}_1) \lor (d' = \mathsf{R} \land \mathsf{f} \in \mathsf{d}_2) \lor (d' = \mathsf{E} \land \mathsf{f} \notin \mathsf{d}_1 \cup \mathsf{d}_2) \qquad d \neq d'}{(d, \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', d :: \overline{d_s}, \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}))}\ \text{REC-LINK-CALL}$$

$$\frac{d \neq d'}{(d, d' :: \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2} (d', \overline{d_s}, \mathsf{Return}(\mathsf{v}, \mathsf{m}))}\ \text{REC-LINK-RET}$$

Fig. 7. Definition of semantic linking relation $\rightsquigarrow_{\mathsf{d}_1,\mathsf{d}_2}$ for Rec.

$$\begin{aligned}(\mathsf{e}_1, s_1) \rightharpoonup (\mathbf{e}_2, s_2) \triangleq\ & \exists \mathbf{r}\ \mathbf{m}\ \overline{\mathbf{v}}.\ \mathbf{e}_2 = \mathbf{Jump!}(\mathbf{r}, \mathbf{m}) * \mathsf{inv}(\mathbf{r}(\mathbf{sp}), \mathbf{m}, \mathsf{mem}(\mathsf{e}_1)) * \\ & (\exists \mathsf{f}\ \overline{\mathsf{v}}\ \mathsf{m}.\ \mathsf{e}_1 = \mathsf{Call!}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) * \mathsf{f} \notin \mathsf{d} * \mathbf{r}(\mathbf{x30}) \in \mathbf{d} * a_{\mathsf{f}} = \mathbf{r}(\mathbf{pc}) * \\ & \quad s_2 = \mathbf{r} :: s_1 * \mathop{\Huge *}_{\mathsf{v},\mathbf{v} \in \overline{\mathsf{v}}, \mathsf{take}(|\overline{\mathsf{v}}|, \mathbf{r}(\mathbf{x0} \ldots \mathbf{x8}))} \mathsf{v} \leftrightarrow \mathbf{v} \\ & \lor \exists \mathsf{v}\ \mathsf{m}\ \mathbf{r}'.\ \mathsf{e}_1 = \mathsf{Return!}(\mathsf{v}, \mathsf{m}) * \mathbf{r}' :: s_2 = s_1 * \mathbf{r}(\mathbf{pc}) = \mathbf{r}'(\mathbf{x30}) * \\ & \quad \mathbf{r}(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) = \mathbf{r}'(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) * \mathsf{v} \leftrightarrow \mathbf{r}(\mathbf{x0}))\end{aligned}$$

$$\begin{aligned}(\mathsf{e}_1, s_1) \leftharpoondown (\mathbf{e}_2, s_2) \triangleq\ & \exists \mathbf{r}\ \mathbf{m}\ \overline{\mathbf{v}}.\ \mathbf{e}_2 = \mathbf{Jump?}(\mathbf{r}, \mathbf{m}) * \mathsf{inv}(\mathbf{r}(\mathbf{sp}), \mathbf{m}, \mathsf{mem}(\mathsf{e}_1)) * \\ & (\exists \mathsf{f}\ \overline{\mathsf{v}}\ \mathsf{m}.\ \mathsf{e}_1 = \mathsf{Call?}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}) * \mathsf{f} \in \mathsf{d} * \mathbf{r}(\mathbf{x30}) \notin \mathbf{d} * a_{\mathsf{f}} = \mathbf{r}(\mathbf{pc}) * \\ & \quad s_1 = \mathbf{r} :: s_2 * \mathop{\Huge *}_{\mathsf{v},\mathbf{v} \in \overline{\mathsf{v}}, \mathsf{take}(|\overline{\mathsf{v}}|, \mathbf{r}(\mathbf{x0} \ldots \mathbf{x8}))} \mathsf{v} \leftrightarrow \mathbf{v} \\ & \lor \exists \mathsf{v}\ \mathsf{m}\ \mathbf{r}'.\ \mathsf{e}_1 = \mathsf{Return?}(\mathsf{v}, \mathsf{m}) * \mathbf{r}' :: s_1 = s_2 * \mathbf{r}(\mathbf{pc}) = \mathbf{r}'(\mathbf{x30}) * \\ & \quad \mathbf{r}(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) = \mathbf{r}'(\mathbf{x19} \ldots \mathbf{x29}, \mathbf{sp}) * \mathsf{v} \leftrightarrow \mathbf{r}(\mathbf{x0}))\end{aligned}$$

Fig. 8. Definition of ($\leftharpoondown$) and ($\rightharpoonup$) for $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$.

meaning reductions can happen inside of an arbitrary evaluation context (see REC-EVAL-CTX). The definition of the evaluation contexts K can be found in the Coq development [Sammler et al. 2022b].

Linking. Syntactically, linking of two Rec libraries (*i.e.*, $\mathsf{R}_1 \cup_{\mathsf{r}} \mathsf{R}_2$) denotes merging the function definitions in R_1 and R_2. In case of overlapping function names, the function declaration of the left library is chosen. (This choice is arbitrary.) If we semantically link two Rec modules (*i.e.*, $\mathsf{M}_1\ {}^{\mathsf{d}_1}\!\oplus_{\mathsf{r}}^{\mathsf{d}_2} \mathsf{M}_2$), then we have to synchronize based on the function call and return events. To define the linking $\mathsf{M}_1\ {}^{\mathsf{d}_1}\!\oplus_{\mathsf{r}}^{\mathsf{d}_2} \mathsf{M}_2$, we use the combinator $M_1 \oplus_X M_2$. In the case of Rec, we pick the relation R depicted in Fig. 7. The most interesting difference to Asm is that linking in Rec has to build up and then wind down a call-stack, which is maintained as the internal state of ($\rightsquigarrow$).

E $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$ WRAPPER

Before we can give the definition of the wrapper $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$, we first need to describe its full form: $\lceil \mathsf{M} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}^{a_{-},\mathbf{d},\mathsf{d},\mathbf{m}}$. In particular, the wrapper is parametrized by a mapping a_{-} from Rec function names to **Asm** addresses, by the instruction address of the **Asm** code **d**, by the function names of the Rec code d, and by a (fragment of) the initial memory **m**, which can be used for global variables.

To define the wrapper, we pick a suitable flavor of separation logic. Instead of directly presenting the technical details of the resource algebra that we choose for $R_{\mathsf{r}\rightleftharpoons\mathsf{a}}$, we instead describe the connectives of the resulting separation logic:

- $\mathsf{p} \leftrightarrow \mathbf{v}$ states that the Rec block id p is mapped to **Asm** address **v**. We lift this relation to locations by $\ell \leftrightarrow \mathbf{v}_2 \triangleq \exists \mathbf{v}_1.\ \ell.\mathsf{blockid} \leftrightarrow \mathbf{v}_1 * \mathbf{v}_2 = \mathbf{v}_1 + \ell.\mathsf{offset}$ and to values (*i.e.*, $\mathsf{v} \leftrightarrow \mathbf{v}$) by relating Rec integers with the same integer in **Asm** and Boolean values with 0 and 1.
- $\mathbf{v}_1 \mapsto_{\mathsf{a}} \mathbf{v}_2$ asserts ownership of the address $\mathbf{v}_1$ in **Asm** memory **m** and asserts that it contains the value $\mathbf{v}_2$. The $\mathbf{v}_1 \mapsto_{\mathsf{a}} \mathbf{v}_2$ connective is useful for asserting private ownership of **Asm** memory in assembly libraries (*e.g.*, it is used internally by the coroutine library to manage its global state).
- $\mathsf{p} \mapsto_{\mathsf{r}} \mathsf{V}$ where V is a map from offsets to values asserts that the block with id p contains exactly V. The $\mathsf{p} \mapsto_{\mathsf{r}} \mathsf{V}$ connective is useful for asserting ownership of locations in the Rec memory, *e.g.*, for locations that are not mapped to the **Asm** memory.
- $\mathsf{inv}(\mathbf{v}, \mathbf{m}, \mathsf{m})$ asserts that **m** and m are in an invariant such that all the aforementioned assertions (*i.e.*, $\mathsf{p} \leftrightarrow \mathbf{v}$, $\mathbf{v}_1 \mapsto_{\mathsf{a}} \mathbf{v}_2$, and $\mathsf{p} \mapsto_{\mathsf{r}} \mathsf{V}$) have the meaning described above and **v** points to a valid stack.

This separation logic is used to define the relations ($\leftharpoondown$) and ($\rightharpoonup$) (depicted in Fig. 8) that are used in the definition of $\lceil\cdot\rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}$. Note that these definitions build on the definition of the Kripke wrapper in Appendix A as they maintain the state s for tracking the call stack in addition to the separation logic predicates. We define:

$$\lceil \mathsf{M} \rceil_{\mathsf{r}\rightleftharpoons\mathsf{a}}^{a_{-},\mathbf{d},\mathsf{d},\mathbf{m}} \triangleq \lceil \mathsf{M} \rceil_X \quad \text{where} \quad X \triangleq (\mathsf{List}(\mathbf{Registers}), R_{\mathsf{r}\rightleftharpoons\mathsf{a}}, \leftharpoondown, \rightharpoonup, [], \mathop{\Huge *}_{\mathbf{v}_1 \mapsto \mathbf{v}_2 \in \mathbf{m}} \mathbf{v}_1 \mapsto_{\mathsf{a}} \mathbf{v}_2)$$

F COROUTINE LINKING

Formally, $\mathsf{M}_1 \oplus_{\mathsf{coro}} \mathsf{M}_2$ is defined using the generic linking operator $M_1 \oplus_X M_2$. Concretely, we define $\mathsf{M}_1 \;{}^{\mathsf{d}_1}\!\oplus_{\mathsf{coro}}^{\mathsf{d}_2,\mathsf{f}} \mathsf{M}_2 \triangleq \mathsf{M}_1 \oplus_{\mathsf{X}_{\mathsf{coro}}} \mathsf{M}_2$ where

$$\mathsf{X}_{\mathsf{coro}} \triangleq ((\mathsf{D} \times \mathsf{option}(\mathsf{FnName})), \rightsquigarrow_{\mathsf{coro}}^{\mathsf{d}_1,\mathsf{d}_2}, (\mathsf{E}, \mathsf{Some}(\mathsf{f})))$$

Note that this linking operator is parametrized by a function name f of the initial function on the right side of the linking (stream in the example). The effect of linking is described by $\rightsquigarrow_{\mathsf{coro}}$ shown in Fig. 9. There are many transitions, but most of them are straightforward The rule CORO-LINK-YIELD encodes the core idea of $\oplus_{\mathsf{coro}}$: If either the left side or the right side performs a call to yield, control switches to the other side, and the event is transformed to a Return?(v, m) event. There is one special case to consider: When M_1 calls yield the first time, there is no yield in M_2 from which to return. Instead this first call to yield becomes the invocation of a designated start function f in M_2 (stream in the example), as stated by CORO-LINK-L-YIELD-INIT. CORO-LINK-INIT handles the initial call from the environment to M_1. If the environment tries to call a function in M_1, the behavior is undefined (CORO-LINK-INIT-UB). CORO-LINK-L-RETURN handles the return from M_1 to the environment. M_2 should never return and thus CORO-LINK-R-RETURN states that doing so would lead to undefined behavior. Finally, CORO-LINK-CALL and CORO-LINK-E-RETURN allow both M_1 and M_2 to call external function (like **print**). However, M_1 and M_2 cannot directly call a function in the other module (without going through yield) (CORO-LINK-CALL-UB) and the environment may not call them back recursively (CORO-LINK-CALL-UB).

CORO-LINK-YIELD
$$\frac{(d = \mathsf{L} \land d' = \mathsf{R}) \lor (d = \mathsf{R} \land d' = \mathsf{L})}{(d, (d, \mathsf{None}), \mathsf{Call}(\mathsf{yield}, [\mathsf{v}], \mathsf{m})) \rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}} (d', (d', \mathsf{None}), \mathsf{Return}(\mathsf{v}, \mathsf{m}))}$$

CORO-LINK-YIELD-UB
$$\frac{d = \mathsf{L} \lor d = \mathsf{R} \qquad |\overline{\mathsf{v}}| \neq 1}{(d, (d, \mathsf{None}), \mathsf{Call}(\mathsf{yield}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}} \lightning}$$

CORO-LINK-L-YIELD-INIT
$$(\mathsf{L}, (\mathsf{L}, \mathsf{Some}(\mathsf{f})), \mathsf{Call}(\mathsf{yield}, [\mathsf{v}], \mathsf{m})) \rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}} (\mathsf{R}, (\mathsf{R}, \mathsf{None}), \mathsf{Call}(\mathsf{f}, [\mathsf{v}], \mathsf{m}))$$

CORO-LINK-L-YIELD-INIT-UB
$$\frac{|\overline{\mathsf{v}}| \neq 1}{(\mathsf{L}, (\mathsf{L}, \mathsf{Some}(\mathsf{f})), \mathsf{Call}(\mathsf{yield}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}} \lightning}$$

CORO-LINK-INIT
$$\frac{\mathsf{f} \in |\mathsf{M}_1|}{(\mathsf{E}, (\mathsf{E}, \mathsf{f}^0), \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} (\mathsf{L}, \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}), (\mathsf{L}, \mathsf{f}^0))}$$

CORO-LINK-INIT-UB
$$\frac{\mathsf{f} \notin |\mathsf{M}_1|}{(\mathsf{E}, (\mathsf{E}, \mathsf{f}^0), \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} \lightning}$$

CORO-LINK-L-RETURN
$$(\mathsf{L}, (\mathsf{L}, \mathsf{f}^0), \mathsf{Return}(\mathsf{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} (\mathsf{E}, (\mathsf{E}, \mathsf{f}^0), \mathsf{Return}(\mathsf{v}, \mathsf{m}))$$

CORO-LINK-R-RETURN
$$(\mathsf{R}, \mathsf{R}, \mathsf{Return}(\mathsf{v}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} \lightning$$

CORO-LINK-CALL
$$\frac{\mathsf{f} \neq \mathsf{yield} \qquad (d = \mathsf{L} \land \mathsf{f} \notin |\mathsf{M}_2|) \lor (d = \mathsf{R} \land \mathsf{f} \notin |\mathsf{M}_1|)}{(\mathsf{L}, (d, \mathsf{f}^0), \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} (\mathsf{E}, (d, \mathsf{f}^0), \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m}))}$$

CORO-LINK-CALL-UB
$$\frac{\mathsf{f} \neq \mathsf{yield} \qquad (d = \mathsf{L} \land \mathsf{f} \in |\mathsf{M}_2|) \lor (d = \mathsf{R} \land \mathsf{f} \in |\mathsf{M}_1|)}{(\mathsf{L}, (d, \mathsf{f}^0), \mathsf{Call}(\mathsf{f}, \overline{\mathsf{v}}, \mathsf{m})) \rightsquigarrow_{\mathsf{coro}} \lightning}$$

CORO-LINK-E-RETURN
$$\frac{(s = \mathsf{L} \land d = \mathsf{L}) \lor (s = \mathsf{R} \land d = \mathsf{R}) \qquad e = \mathsf{Return}(\_, \_)}{(\mathsf{E}, (d, \mathsf{f}^0), e) \rightsquigarrow_{\mathsf{coro}} (d, (d, \mathsf{f}^0), e)}$$

CORO-LINK-E-CALL-UB
$$\frac{(s = \mathsf{L} \land d = \mathsf{L}) \lor (s = \mathsf{R} \land d = \mathsf{R}) \qquad e = \mathsf{Call}(\_, \_, \_)}{(\mathsf{E}, (d, \mathsf{f}^0), e) \rightsquigarrow_{\mathsf{coro}} \lightning}$$

Fig. 9. Definition of linking relation $\rightsquigarrow^{\mathsf{d}_1,\mathsf{d}_2}_{\mathsf{coro}}$.

REFERENCES

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022a. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI*. ACM, 825–840. https://doi.org/10.1145/3519939.3523434

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2022b. DimSum: A Decentralized Approach to Multi-language Semantics and Verification (Coq development). https:

//gitlab.mpi-sws.org/iris/dimsum/.