This document describes the semantics of aspects of the SPL language relating to SIP RFC 3261.

1 Syntax

The semantics considers a much simplified version of the SPL language. An SPL program has the following form.

```
service service {
  processing {
    (type x = exp;)*
    (direction? type method_name { (branch branch {decl* stmt})+ })*
    registration {
      (type x = exp;)*
      (direction? type method_name { (branch branch {decl* stmt})+ })*
      dialog {
        (type x = exp;)*
        (direction? type method_name { (branch branch {decl* stmt})+ })*
      }
    }
  }
}
```

The grammar is defined as follows:

program	::=	`service` *service* `{processing {`*decl** *method** `registration {`*decl** *method** `dialog {`*decl** *method** `}}}}`
decl	::=	*type* *x* `=` *exp*`;`
method	::=	*direction*$^?$ *type* *method_name* `{ (branch` *branch* `{`*decl** *stmt*`})`$^+$ `}`
direction	::=	`incoming` | `outgoing`
type	::=	`response` | `uri` | `uri list` | `void` | `bool` | `string` | `status`
stmt	::=	`{}` | `{`*stmt*$_1$ *stmt*$_2$`}` | *x* `=` *exp* | `return` *exp*$^?$ `branch` *b*`;` | `if (`*exp*`)` *stmt*$_1$ `else` *stmt*$_2$
exp	::=	`forward` *exp* | `parallel forward` *exp* | `<` *exp* `>` | *exp*$_1$ `==` *exp*$_2$ | *x* | *c* | `requestURI` | *exp*$_1$ `with` *field* `=` *exp*$_2$

service, *branch*, and *x* are arbitrary identifiers. The `{`*decl*$^+$ *stmt*`}` form is only allowed at the root of a method handler. *method_name* is restricted according to the level at which it occurs:

- At the `service` level, the allowed methods are `deploy` and `undeploy`.
- At the `registration` level, the allowed methods are `initial REGISTER`, `medial REGISTER`, and `unregister`.
- At the `dialog` level, the allowed methods are `initial INVITE`, `medial INVITE`, `CANCEL`, `initial ACK`, `medial ACK`, `BYE`, and `end_dialog`.

Constants *c* are strings, responses (*e.g.*, `/SUCCESS`, `/ERROR`, and various refinements thereof), and lists of uris (*e.g.*, `'sip:joe@bigcorp.com'`). `status` is an enumerated type with elements `SUCCESS` and `ERROR`.

2 Static semantics

The only types *ty* used by the type system are request, response, uri, uri list, void, bool, string, and status. The type system is defined in terms of an environment τ of type $var \to ty$. The judgments for statements, expressions, declarations, method declarations and programs have the following form, where $t_0, t \in ty$:

- $t_0, \tau \vdash stmt : t$
- $t_0, \tau \vdash exp : t$
- $t_0, \tau \vdash decl : \tau'$
- $\tau \vdash method$
- $\vdash program$

t_0 is always the type of the enclosing method, and is used to determine the return type of `forward`. For example, there is no response associated with an `initial ACK` or `medial ACK`, and so a `forward` in an `initial ACK` or `medial ACK` handler has return type void.

The judgments for statements are defined as follows:

$$t_0, \tau \vdash \texttt{\{\}} : \mathsf{void} \qquad \frac{t_0, \tau \vdash stmt_1 : \mathsf{void} \qquad t_0, \tau \vdash stmt_2 : t}{t_0, \tau \vdash \texttt{\{}stmt_1 \;\; stmt_2\texttt{\}} : t}$$

$$\frac{t_0, \tau \vdash exp : t \qquad t = \tau(x)}{t_0, \tau \vdash x \texttt{ = } exp : \mathsf{void}} \qquad \frac{t_0, \tau \vdash exp : t}{t_0, \tau \vdash \texttt{return } exp \texttt{ branch } b\texttt{;} : t} \qquad t_0, \tau \vdash \texttt{return branch } b\texttt{;} : \mathsf{void}$$

$$\frac{t_0, \tau \vdash exp : \mathsf{bool} \qquad t_0, \tau \vdash stmt_1 : t_1 \qquad t_0, \tau \vdash stmt_2 : t_2 \qquad t_1 = t_2}{t_0, \tau \vdash \texttt{if (}exp\texttt{)} \;\; stmt_1 \;\; \texttt{else} \;\; stmt_2 : t_1}$$

The judgments for expressions are defined as follows:

$$\frac{t_0, \tau \vdash exp : \mathsf{uri\ list}}{t_0, \tau \vdash \texttt{forward } exp : t_0} \qquad \frac{t_0, \tau \vdash exp : \mathsf{uri\ list}}{t_0, \tau \vdash \texttt{parallel forward } exp : t_0} \qquad \frac{t_0, \tau \vdash exp : \mathsf{uri}}{t_0, \tau \vdash \texttt{<} \;\; exp \;\; \texttt{>} : \mathsf{uri\ list}}$$

$$\frac{t_0, \tau \vdash exp_1 : t_1 \qquad t_0, \tau \vdash exp_1 : t_2 \qquad t_1 = t_2}{t_0, \tau \vdash exp_1 \texttt{ == } exp_2 : \mathsf{bool}} \qquad t_0, \tau \vdash x : \tau(x)$$

$$\frac{c \in \{\texttt{SUCCESS}, \texttt{ERROR}, \ldots\}}{t_0, \tau \vdash c : \mathsf{status}} \qquad \frac{c \in \{\texttt{/SUCCESS}, \texttt{/ERROR}, \ldots\}}{t_0, \tau \vdash c : \mathsf{response}}$$

$$\frac{c \in \{\texttt{"..."}, \ldots\}}{t_0, \tau \vdash c : \mathsf{string}} \qquad \frac{c \in \{\texttt{<'sip:joe@bigcorp.com'}, \ldots\texttt{>}, \ldots\}}{t_0, \tau \vdash c : \mathsf{uri\ list}} \qquad t_0, \tau \vdash \texttt{requestURI} : \mathsf{uri}$$

$$\frac{t_0, \tau \vdash exp_1 : \mathsf{request} \qquad t_0, \tau \vdash exp_2 : \mathsf{string}}{t_0, \tau \vdash exp_1 \texttt{ with } field \texttt{ = } exp_2 : \mathsf{request}} \qquad \frac{t_0, \tau \vdash exp_1 : \mathsf{response} \qquad t_0, \tau \vdash exp_2 : \mathsf{string}}{t_0, \tau \vdash exp_1 \texttt{ with } field \texttt{ = } exp_2 : \mathsf{response}}$$

The judgments for declarations are defined as follows, in which we use *type* as either the syntactic type name or the name of the corresponding type in the type system, as convenient. The handler return type argument to the analysis of each expression is $\bot$ because declarations are not allowed to use `(parallel) forward`.

$$\frac{\bot, \tau \vdash exp : t \qquad t = type}{\tau \vdash type \;\; x \texttt{ = } exp\texttt{;} : \tau[x \mapsto type]} \qquad \frac{\bot, \tau \vdash exp : t \qquad t = type \qquad \tau[x \mapsto type] \vdash decl_2 \ldots : \tau'}{\tau \vdash type \;\; x \texttt{ = } exp\texttt{;} \;\; decl_2 \;\; \ldots : \tau'}$$

The judgments for method declarations are defined as follows:

$$\frac{\begin{array}{c}\tau \vdash decl_{1_1} \ldots decl_{1_{m_1}} : \tau_1 \qquad \mathsf{status}, \tau_1 \vdash stmt_1 : t_1 \\ \ldots \\ \tau \vdash decl_{n_1} \ldots decl_{n_{m_n}} : \tau_n \qquad \mathsf{status}, \tau_n \vdash stmt_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void}\end{array}}{\begin{array}{l}\tau \vdash direction^? \;\; \texttt{void deploy} \\ \quad \texttt{\{branch } branch_1 \;\; \texttt{\{}decl_{1_1} \ldots decl_{1_{m_1}} \;\; stmt_1 \;\; \texttt{\}...branch } branch_n \;\; \texttt{\{}decl_{n_1} \ldots decl_{n_{m_n}} \;\; stmt_n \;\; \texttt{\}\}}\end{array}}$$

$$\frac{\begin{array}{c} \mathit{method\_name} \in \{\texttt{deploy}, \texttt{undeploy}, \texttt{unregister}, \texttt{uninvite}\} \\ \tau \vdash \mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} : \tau_1 \qquad \mathsf{void}, \tau_1 \vdash \mathit{stmt}_1 : t_1 \\ \ldots \\ \tau \vdash \mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} : \tau_n \qquad \mathsf{void}, \tau_n \vdash \mathit{stmt}_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void} \end{array}}{\begin{array}{l} \tau \vdash \mathit{direction}^? \ \texttt{void} \ \mathit{method\_name} \\ \quad \texttt{\{branch} \ \mathit{branch}_1 \ \texttt{\{}\mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} \ \mathit{stmt}_1 \ \texttt{\}...branch} \ \mathit{branch}_n \ \texttt{\{}\mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} \ \mathit{stmt}_n \ \texttt{\}\}} \end{array}}$$

$$\frac{\begin{array}{c} \mathit{method\_name} \in \{\texttt{initial ACK}, \texttt{medial ACK}\} \\ \tau \vdash \mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} : \tau_1 \qquad \mathsf{void}, \tau_1[m \mapsto \mathsf{request}] \vdash \mathit{stmt}_1 : t_1 \\ \ldots \\ \tau \vdash \mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} : \tau_n \qquad \mathsf{void}, \tau_n[m \mapsto \mathsf{request}] \vdash \mathit{stmt}_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void} \end{array}}{\begin{array}{l} \tau \vdash \mathit{direction}^? \ \texttt{void} \ \mathit{method\_name} \ \texttt{(request} \ m\texttt{)} \\ \quad \texttt{\{branch} \ \mathit{branch}_1 \ \texttt{\{}\mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} \ \mathit{stmt}_1 \ \texttt{\}...branch} \ \mathit{branch}_n \ \texttt{\{}\mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} \ \mathit{stmt}_n \ \texttt{\}\}} \end{array}}$$

$$\frac{\begin{array}{c} \mathit{method\_name} \in \{\texttt{initial REGISTER}, \texttt{medial REGISTER}, \texttt{initial INVITE}, \texttt{medial INVITE}, \texttt{CANCEL}, \texttt{BYE}\} \\ \tau \vdash \mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} : \tau_1 \qquad \mathsf{response}, \tau_1[m \mapsto \mathsf{request}] \vdash \mathit{stmt}_1 : t_1 \\ \ldots \\ \tau \vdash \mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} : \tau_n \qquad \mathsf{response}, \tau_n[m \mapsto \mathsf{request}] \vdash \mathit{stmt}_n : t_n \\ t_1 = \ldots = t_n = \mathsf{response} \end{array}}{\begin{array}{l} \tau \vdash \mathit{direction}^? \ \texttt{response} \ \mathit{method\_name} \ \texttt{(request} \ m\texttt{)} \\ \quad \texttt{\{branch} \ \mathit{branch}_1 \ \texttt{\{}\mathit{decl}_{1_1} \ldots \mathit{decl}_{1_{m_1}} \ \mathit{stmt}_1 \ \texttt{\}...branch} \ \mathit{branch}_n \ \texttt{\{}\mathit{decl}_{n_1} \ldots \mathit{decl}_{n_{m_n}} \ \mathit{stmt}_n \ \texttt{\}\}} \end{array}}$$

The judgments for programs are defined as follows:

$$\frac{\begin{array}{cc} \emptyset \vdash \mathit{decl}_{p_1} \ldots \mathit{decl}_{p_a} : \tau_p & \tau_p \vdash \mathit{method}_{p_1} \ldots \tau_p \vdash \mathit{method}_{p_b} \\ \tau_p \vdash \mathit{decl}_{r_1} \ldots \mathit{decl}_{r_c} : \tau_r & \tau_r \vdash \mathit{method}_{r_1} \ldots \tau_r \vdash \mathit{method}_{r_d} \\ \tau_r \vdash \mathit{decl}_{d_1} \ldots \mathit{decl}_{d_e} : \tau_d & \tau_d \vdash \mathit{method}_{d_1} \ldots \tau_d \vdash \mathit{method}_{d_f} \end{array}}{\begin{array}{l} \vdash \texttt{service} \ \mathit{service} \ \texttt{\{processing \{}\mathit{decl}_{p_1} \ldots \mathit{decl}_{p_a} \ \mathit{method}_{p_1} \ldots \mathit{method}_{p_b} \\ \texttt{registration \{}\mathit{decl}_{r_1} \ldots \mathit{decl}_{r_c} \ \mathit{method}_{r_1} \ldots \mathit{method}_{r_d} \ \texttt{dialog \{}\mathit{decl}_{d_1} \ldots \mathit{decl}_{d_e} \ \mathit{method}_{d_1} \ldots \mathit{method}_{d_f} \ \texttt{\}\}\}} \end{array}}$$

The type checking rules are, however, not sufficient to ensure that a SPL program is well-formed. Other properties to consider include the following:

- There is at most one incoming and at most one outgoing handler for each method name.
- There is at most one successful `forward` per dialog handler. There can be multiple successful `forward`s for a registration.
- There is something to verify about the branch names. Once a branch is created, it has to be used (or matched by `default`) in every handler that can be executed subsequently. Actually, the semantics doesn't care about this. If the branch is not defined, the message is simply forwarded.
- Assignments to header fields should be checked as to whether the field is assignable.
- Forward cannot appear in a declaration, even the declaration of a local variable.
- Handlers for control methods, *i.e.*, `deploy`, `undeploy`, `unregister`, and `end_dialog`, cannot refer to anything about a message, such as headers or `requestURI`. They cannot use `forward`.
- A handler that has `void` return type must use `forward` in tail position.
- If `forward` succeeds or does not succeed, there are some corresponding constraints on the return value.

3 Addresses

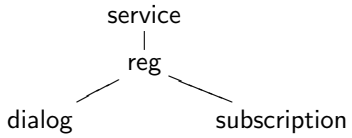
An address is a sequence of unique identifiers defined as follows.

- $\mathsf{label} = \mathsf{session\_type} \times \mathsf{id}$, where id is any form of identifier unique within the $\mathsf{session\_type}$.
- $\mathsf{address} = \mathsf{label\ list}$

The session types depend on the protocol being addressed. For SPL, we thus have:

- $\mathsf{session\_type} = \{\mathsf{service}, \mathsf{reg}, \mathsf{dialog}\}$

In practice, within a given address, the sequence of session types read from left to right form a non-empty prefix of a path in a tree describing session relationships. For example, if we were to add subscriptions to SPL, the tree would be as follows:



Given such a tree, a valid address is one that contains a prefix of the sequence of session types on a path from the root to a leaf of a tree, where some of the session types are optional. For example, for SPL reg is optional, and thus valid addresses have one of the following forms:

- $\langle(\mathsf{service}, x)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y), (\mathsf{dialog}, z)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y), (\mathsf{subscription}, a)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{dialog}, z)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{subscription}, a)\rangle$

The following functions are available for manipulating addresses:

- $\mathsf{parent} : \mathsf{address} \to \mathsf{address}$
- $\mathsf{self} : \mathsf{address} \to \mathsf{label}$
- $\mathsf{service} : \mathsf{address} \to \mathsf{label}$

These functions are defined as follows:

$$\frac{n \geq 1}{\mathsf{parent}(\langle label_1, \ldots, label_n\rangle) = \langle label_1, \ldots, label_{n-1}\rangle}$$

$$\frac{n \geq 1}{\mathsf{self}(\langle label_1, \ldots, label_n\rangle) = label_n}$$

$$\frac{n \geq 1}{\mathsf{service}(\langle label_1, \ldots, label_n\rangle) = label_1}$$

These functions can also be applied to lists of items in one-to-one mapping with a list of addresses, such as the list of associated environments.

4 Global state

The global state, σ describes the state of the various sessions and the relationships between them. The state maps an address to a configuration, described as follows:

- $\mathsf{state} = \mathsf{address} \rightarrow \mathsf{branch\ list} \times \mathsf{status} \times (\mathsf{var} \rightarrow \mathsf{value}) \times \mathsf{address\ list}$
- $\mathsf{status} = \mathsf{bool} \times \mathsf{bool} \times \mathsf{string}_\perp \times \mathsf{int}$

status is a tuple composed of four pieces of information: *live*, $persistent_\perp$, *close_fn*, and *open*. *live* indicates whether the session is accepting messages, either its own messages or create session messages of the immediate subsessions. The value is true if this is the case. $persistent_\perp$ indicates whether on ending the session, the data associated with the session persists until the closing of all immediate subsessions. The value is true if this is the case. This field only has a meaningful value if *live* is false, as it is at the moment of ending the session that the semantics chooses whether the ending of the session is persistent. *close_fn* is a string indicating the name of the handler to execute on ending the session, or $\perp$ if there is none. *open* is an integer indicating how many transactions are in progress at the current level. address list is the list of addresses of the subsessions.

There are various kinds of configurations depending on the session being modeled:

- Dialog: In this case, the list of addresses is always empty, *persistent* is false (since a dialog has no subsessions, whether it is persistent or not doesn't matter), and the $\mathsf{string}_\perp$ component has value `end_dialog`.
- Registration: In this case, the list of addresses always contains only addresses of dialogs. The branch list component always contains only one element. The $\mathsf{string}_\perp$ component has value `unregister`.
- Service: In this case, the list of addresses always contains only addresses of registrations. The branch list component always contains one element. The $\mathsf{string}_\perp$ component has value `undeploy`.

If a session has more than one kind of subsession (*i.e.*, if we add subscriptions as subsessions of registrations), then the addresses of these different kinds of subsessions are all mixed together in the address list component. The information about a specific kind of subsession can be obtained by filtering on the session_type in the label at the end of each address.

5 Useful functions

We define some functions for accessing and updating the global state σ. The types of these functions are as follows:

- $\mathsf{init\_session} : \mathsf{state} \times \mathsf{address} \times \mathsf{branch\ list} \times \mathsf{bool} \times \mathsf{bool}_\perp \times \mathsf{string}_\perp \times \mathsf{int} \rightarrow \mathsf{state}$
- $\mathsf{lookup\_decls} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{decl\ list}$
- $\mathsf{lookup\_handlers} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{handlers}$
- $\mathsf{lookup\_branches} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{branch\ list}$
- $\mathsf{lookup\_live} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{bool}$
- $\mathsf{lookup\_persistent} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{bool}$
- $\mathsf{lookup\_close\_fn} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{string}_\perp$
- $\mathsf{lookup\_open} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{int}$

- lookup_envs : state × address → (var → value) list
- lookup_dependents : state × address → address list
- update_branches : state × address × branch list → state
- set_persistence : state × address × bool → state
- inc_open : state × address → state
- dec_open : state × address → state
- update_envs : state × address × (var → value) list → state
- delete_config : state × address → state

These functions are defined in terms of the functions get_config : state × address → config and set_config : state × address × config → state, which are defined as follows:

$$\mathsf{get\_config}(\sigma, \mathit{address}) = \sigma(\mathit{address})$$

$$\mathsf{set\_config}(\sigma, \mathit{address}, \mathit{config}) = \sigma[\mathit{address} \mapsto \mathit{config}]$$

The definitions of the above functions are then as follows:

$$\frac{\begin{array}{c}\mathsf{parent}(\mathit{address}) = \langle\rangle \\ \mathit{config} = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \bot, \emptyset\rangle \\ \sigma' = \mathsf{set\_config}(\sigma, \mathit{address}, \mathit{config})\end{array}}{\mathsf{init\_session}(\sigma, \mathit{address}, \mathit{branches}, \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}) = \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{parent}(\mathit{address}) \neq \langle\rangle \\ \mathsf{get\_config}(\sigma, \mathsf{parent}(\mathit{address})) = \langle \mathit{branches}_p, \mathit{status}_p, \mathit{env}_p, \mathit{subsessions}_p\rangle \\ \mathsf{set\_config}(\sigma, \mathsf{parent}(\mathit{address}), \langle \mathit{branches}_p, \mathit{status}_p, \mathit{env}_p, \mathit{address} :: \mathit{subsessions}_p\rangle) = \sigma' \\ \mathit{config} = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \bot, \emptyset\rangle \\ \sigma'' = \mathsf{set\_config}(\sigma', \mathit{address}, \mathit{config})\end{array}}{\mathsf{init\_session}(\sigma, \mathit{address}, \mathit{branches}, \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}) = \sigma''}$$

$$\frac{\mathsf{self}(\mathit{address}) = \langle \mathit{session\_type}, \_\rangle}{\mathsf{lookup\_decls}(\phi, \mathit{address}) = \phi_{\mathsf{decls}}(\mathsf{service}(\mathit{address}))(\mathit{session\_type})}$$

$$\mathsf{lookup\_handlers}(\phi, \mathit{address}) = \phi_{\mathsf{handlers}}(\mathsf{service}(\mathit{address}))\rangle$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \mathit{status}, \mathit{env}, \mathit{subsessions}\rangle}{\mathsf{lookup\_branches}(\sigma, \mathit{address}) = \mathit{branches}}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \mathit{env}, \mathit{subsessions}\rangle}{\mathsf{lookup\_live}(\sigma, \mathit{address}) = \mathit{live}}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \mathit{env}, \mathit{subsessions}\rangle}{\mathsf{lookup\_persistent}(\sigma, \mathit{address}) = \mathit{persistent}}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \mathit{env}, \mathit{subsessions}\rangle}{\mathsf{lookup\_close\_fn}(\sigma, \mathit{address}) = \mathit{close\_fn}}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open}\rangle, \mathit{env}, \mathit{subsessions}\rangle}{\mathsf{lookup\_open}(\sigma, \mathit{address}) = \mathit{open}}$$

$$\mathsf{lookup\_envs}(\sigma, \langle\rangle) = \langle\rangle$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \_, \_, \mathit{env}, \_ \rangle \quad \mathsf{lookup\_envs}(\sigma, \mathsf{parent}(\mathit{address})) = \mathit{envs}}{\mathsf{lookup\_envs}(\sigma, \mathit{address}) = \mathit{envs} +\!\!+ \langle \mathit{env} \rangle}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \mathit{status}, \mathit{env}, \mathit{subsessions} \rangle}{\mathsf{lookup\_dependents}(\sigma, \mathit{address}) = \mathit{subsessions}}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \_, \mathit{status}, \mathit{env}, \mathit{subsessions} \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}, \langle \mathit{branches}, \mathit{status}, \mathit{env}, \mathit{subsessions} \rangle) == \sigma'}{\mathsf{update\_branches}(\sigma, \mathit{address}, \mathit{branches}) = \sigma'}$$

After a call to set_persistence, the *live* attribute is always false. The boolean that is passed to set_persistence is the persistence. There are two rules because it doesn't make sense for something that is not live and not persistent to become persistent. In this case, as treated by the second rule, the persistence is ignored.

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}', \mathit{close\_fn}, \mathit{open} \rangle, \mathit{env}, \mathit{subsessions} \rangle \quad \mathit{live} = \mathsf{true} \vee \mathit{persistent}' = \mathsf{true} \quad \mathsf{set\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathsf{false}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open} \rangle, \mathit{env}, \mathit{subsessions} \rangle}{\mathsf{set\_persistence}(\sigma, \mathit{address}, \mathit{persistent}) = \sigma'}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}', \mathit{close\_fn}, \mathit{open} \rangle, \mathit{env}, \mathit{subsessions} \rangle \quad \mathit{live} = \mathsf{false} \wedge \mathit{persistent}' = \mathsf{false}}{\mathsf{set\_persistence}(\sigma, \mathit{address}, \mathit{persistent}) = \sigma}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open} \rangle, \mathit{env}, \mathit{subsessions} \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open} + 1 \rangle, \mathit{env}, \mathit{subsessions} \rangle}{\mathsf{inc\_open}(\sigma, \mathit{address}) = \sigma'}$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open} \rangle, \mathit{env}, \mathit{subsessions} \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \langle \mathit{live}, \mathit{persistent}, \mathit{close\_fn}, \mathit{open} - 1 \rangle, \mathit{env}, \mathit{subsessions} \rangle}{\mathsf{dec\_open}(\sigma, \mathit{address}) = \sigma'}$$

$$\mathsf{update\_envs}(\sigma, \langle \rangle, \langle \rangle) = \sigma$$

$$\frac{\mathsf{get\_config}(\sigma, \mathit{address}) = \langle \mathit{branches}, \mathit{status}, \_, \mathit{subsessions} \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}, \langle \mathit{branches}, \mathit{status}, \mathsf{self}(\mathit{envs}), \mathit{subsessions} \rangle) = \sigma' \quad \mathsf{update\_envs}(\sigma', \mathsf{parent}(\mathit{address}), \mathsf{parent}(\mathit{envs})) = \sigma''}{\mathsf{update\_envs}(\sigma, \mathit{address}, \mathit{envs}) = \sigma''}$$

In the second rule for delete_config, we delete the address from the address list of the parent.

$$\frac{\mathsf{parent}(\mathit{address}) = \langle \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}, \bot) = \sigma'}{\mathsf{delete\_config}(\sigma, \mathit{address}) = \sigma'}$$

$$\frac{\mathsf{parent}(\mathit{address}) \neq \langle \rangle \quad \mathsf{set\_config}(\sigma, \mathit{address}, \bot) = \sigma' \quad \mathsf{get\_config}(\sigma', \mathsf{parent}(\mathit{address})) = \langle \mathit{branches}, \mathit{status}, \mathit{env}, \mathit{subsessions} \rangle \quad \mathsf{set\_config}(\sigma', \mathsf{parent}(\mathit{address}), \langle \mathit{branches}, \mathit{status}, \mathit{env}, \mathit{subsessions} - \mathit{address} \rangle) = \sigma''}{\mathsf{delete\_config}(\sigma, \mathit{address}, \mathit{envs}) = \sigma''}$$

In terms of these functions, we define the functions create_session, prepare_method_invocation, continue_session, and end_session, having the following types:

- create_session : program × state × address × branch list × $\mathsf{string}_\bot$ → state
- prepare_method_invocation : program × state × address × direction × method → var × decl list × stmt × env list × state

- continue_session : program × state × address × env list × branch list → state
- end_session : program × state × address → state

The definitions of these functions are as follows. In the second rule, *handlers* returns the parameter m of the method, and the local declarations *decls* and the code *stmt* associated with the given branch.

$$\frac{\begin{array}{c}\mathsf{lookup\_decls}(\phi, address) = decls \\ \mathsf{init\_session}(\sigma, address, branches, \mathsf{true}, \bot, close\_fn, 0) = \sigma' \\ \mathsf{lookup\_envs}(\sigma', \mathsf{parent}(address)) = parent\_envs \\ \langle parent\_envs, \langle \bot, \emptyset \rangle, \emptyset \rangle \models decls \Rightarrow session\_env \\ \mathsf{update\_envs}(\sigma', address, parent\_envs +\!\!+ \langle session\_env \rangle) = \sigma''\end{array}}{\mathsf{create\_session}(\phi, \sigma, address, branches, close\_fn) = \sigma''}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_handlers}(\phi, address) = handlers \\ \mathsf{lookup\_branches}(\sigma, address) = branches \\ handlers(direction, method, branches) = \langle m, decls, stmt \rangle \\ \mathsf{lookup\_envs}(\sigma, address) = envs \\ \mathsf{inc\_open}(\sigma, address) = \sigma'\end{array}}{\mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, method) = \langle m, decls, stmt, envs, \sigma' \rangle}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, envs) = \sigma' \\ \mathsf{update\_branches}(\sigma', address, branches) = \sigma'' \\ \mathsf{dec\_open}(\sigma'', address) = \sigma''' \\ \mathsf{lookup\_live}(\sigma''', address) = \mathsf{true}\end{array}}{\mathsf{continue\_session}(\phi, \sigma, address, envs, branches) = \sigma'''}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, envs) = \sigma' \\ \mathsf{update\_branches}(\sigma', address, branches) = \sigma'' \\ \mathsf{dec\_open}(\sigma'', address) = \sigma''' \\ \mathsf{lookup\_live}(\sigma''', address) = \mathsf{false} \\ \mathsf{end\_session}(\phi, \sigma''', address) = \sigma''''\end{array}}{\mathsf{continue\_session}(\phi, \sigma, address, envs, branches) = \sigma''''}$$

All of the remaining rules are used to define end_session. The definition uses the functions end_parents, end_children, and close_session, having the following types:

- end_children : program × state × address → state
- end_parents : program × state × address → state
- close_session : program × state × address → state

Ending a session entails ending all the children, with end_children, and then, if this process is successful, ending any of the parents that are not live. Ending the children entails first moving down the tree of children, setting *live* in each case to false, and then starting from the leaves, closing each child that has no pending transactions and no dependents. This process is oblivious to whether the children are persistent. Note, however, that when the current level is persistent, this process is only initiated when there are no children, in which case end_children only closes the current level. Ending the parents entails looking for parents that are not live and that if persistent have no dependents. This process works its way up the tree, stopping when reaching a parent that does not have these properties, or when reaching the root of the tree.

close_session invokes the close handler of the current session, once we have decided that the current session should be ended. When there is code to evaluate, it is evaluated with respect to the empty continuation because the code cannot use `forward`.

The following rules decide whether to end the session at the current level. On invoking end_session, the *live* attribute of the current level must be false.

$$\frac{\mathsf{lookup\_persistent}(\sigma, \mathit{address}) = \mathsf{true} \wedge \mathsf{lookup\_dependents}(\sigma, \mathit{address}) \neq \emptyset}{\mathsf{end\_session}(\phi, \sigma, \mathit{address}) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_persistent}(\sigma, \mathit{address}) = \mathsf{false} \vee \mathsf{lookup\_dependents}(\sigma, \mathit{address}) = \emptyset \\ \mathsf{end\_children}(\phi, \sigma, \mathit{address}) = \sigma' \\ \mathsf{end\_parents}(\phi, \sigma', \mathsf{parent}(\mathit{address})) = \sigma''\end{array}}{\mathsf{end\_session}(\phi, \sigma, \mathit{address}) = \sigma''}$$

The following rules end the session and all of its children, starting at the leaves. These rules assume that σ is up to date with respect to the various session environments. end_children' of type program × state × address × label list → state is used to loop over the list of dependents.

$$\frac{\begin{array}{c}\mathsf{lookup\_dependents}(\sigma, \mathit{address}) = \emptyset \wedge \mathsf{lookup\_open}(\sigma, \mathit{address}) = 0 \\ \mathsf{close\_session}(\phi, \sigma, \mathit{address}) = \sigma'\end{array}}{\mathsf{end\_children}'(\phi, \sigma, \mathit{address}, \langle\rangle) = \sigma'}$$

$$\frac{\mathsf{lookup\_dependents}(\sigma, \mathit{address}) \neq \emptyset \vee \mathsf{lookup\_open}(\sigma, \mathit{address}) > 0}{\mathsf{end\_children}'(\phi, \sigma, \mathit{address}, \langle\rangle) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, \mathit{child\_address}, \mathsf{false}) = \sigma' \\ \mathsf{end\_children}(\phi, \sigma', \mathit{child\_address}) = \sigma'' \\ \mathsf{end\_children}'(\phi, \sigma'', \mathit{address}, \pi) = \sigma'''\end{array}}{\mathsf{end\_children}'(\phi, \sigma, \mathit{address}, \mathit{child\_address} :: \pi) = \sigma'''}$$

$$\frac{\mathsf{end\_children}'(\phi, \sigma, \mathit{address}, \mathsf{lookup\_dependents}(\sigma, \mathit{address})) = \sigma'}{\mathsf{end\_children}(\phi, \sigma, \mathit{address}) = \sigma'}$$

The following rules end any of the parents that are not live and now have no more subsessions. These rules assume that σ is up to date with respect to the various session environments.

$$\mathsf{end\_parents}(\phi, \sigma, \langle\rangle) = \sigma$$

$$\frac{\mathsf{lookup\_live}(\sigma, \mathit{address}) = \mathsf{true} \vee \mathsf{lookup\_dependents}(\sigma, \mathit{address}) \neq \emptyset}{\mathsf{end\_parents}(\phi, \sigma, \mathit{address}) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_live}(\sigma, \mathit{address}) = \mathsf{false} \wedge \mathsf{lookup\_dependents}(\sigma, \mathit{address}) = \emptyset \\ \mathsf{close\_session}(\phi, \sigma, \mathit{address}) = \sigma' \\ \mathsf{end\_parents}(\phi, \sigma', \mathsf{parent}(\mathit{address})) = \sigma''\end{array}}{\mathsf{end\_parents}(\phi, \sigma, \mathit{address}) = \sigma''}$$

The following rules perform the actions required locally to close a single session. These rules assume that σ is up to date with respect to the various session environments.

$$\frac{\mathsf{lookup\_close\_fn}(\sigma, \mathit{address}) = \bot \quad \mathsf{delete\_config}(\sigma, \mathit{address}) = \sigma'}{\mathsf{close\_session}(\phi, \sigma, \mathit{address}) = \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_close\_fn}(\sigma, \mathit{address}) = \mathit{close\_fn} \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, \mathit{address}, \bot, \mathit{close\_fn}) = \langle \_, \mathit{decls}, \mathit{stmt}, \mathit{envs}, \sigma' \rangle \\ \tau = \langle \sigma, \mathit{address} \rangle \qquad r = \langle \mathit{envs}, \langle \bot, \langle \bot, \emptyset \rangle \rangle \rangle \\ \tau, r, \langle \texttt{CLOSE} \rangle \models \mathit{decls}, \mathit{stmt} \Rightarrow^* \_, \sigma' \\ \mathsf{delete\_config}(\sigma', \mathit{address}) = \sigma''\end{array}}{\mathsf{close\_session}(\phi, \sigma, \mathit{address}) = \sigma''}$$

$$\frac{\mathsf{update\_envs}(\sigma, \mathit{address}, \rho_{\mathsf{envs}}) = \sigma'}{\langle \sigma, \mathit{address} \rangle, \rho \models \langle \texttt{CLOSE} \rangle, \_, \_ \Rightarrow \bot, \sigma'}$$

5.1 Environments

The semantics uses the following environments:

- *envs*: A sequence of environments corresponding to the current address. For SPL, this contains at most the following environments:
  - *env*: The service environment.
  - *rid_reg_env*: The register environment for a registration instance.
  - *did_dialog_env*: The dialog environment for a dialog instance.
- *message_info*: A pair of the handler parameter name and a pair of the requestURI and an environment containing bindings derived from the message headers.
- *local_env*: The local environment.

All of these environments (except *envs*) are mappings from variables to values. The semantics refers to these environments collectively as ρ which is a tuple of the above environments in the above order. The individual environments are accessed by *e.g.* $\rho_{\mathsf{message\_info}}$.

In the evaluation of the entry point of a handler, the environment r is created, which is the same as ρ, but does not contain a local environment.

5.2 Judgments for declarations, statements and expressions

The semantics is organized as an abstract machine containing the following kinds of configurations:

- Service: $\mathit{message}, \phi, \sigma \models \mathit{service}$

  A *message* has the form $\langle \mathit{method}, \mathit{direction}, \mathit{headers}, \mathit{rq} \rangle$, where *rq* is the request URI of the message.
- Service continuation: $\tau, \rho \models s, \mathit{resp}$

  τ gives context information and has the form $\langle \sigma, \mathit{address} \rangle$. This information is not used in the normal course of executing the body of a handler, but is used in the treatment of (`parallel`) `forward`.

  s is the stack of continuations.
- Handler body: $\tau, \rho, s \models \mathit{decls}, \mathit{stmt}$
- Statement: $\tau, \rho, s \models \mathit{stmt}$

- Statement continuation: $\tau, \rho \models s, resp_\perp, branch_\perp$

  $resp_\perp$ is either a response or $\perp$. $\perp$ is used for statements other than `return`, which do not have a response. $branch_\perp$ is defined similarly.

- Expression: $\tau, \rho, s \models exp$

- Expression continuation: $\tau, \rho \models s, value$

  value is an arbitrary value, which is not necessarily a response.

- SIP: A SIP configuration is a call to a function of the underlying platform, such as forward.

- Terminal: $resp_\perp, \sigma$

The execution of a handler is represented by a sequence of configurations, related according to the semantic rules. The initial configuration is a service configuration. The final configuration is a terminal configuration. A configuration between the initial and final configurations is any kind of configuration other than a service configuration or a terminal configuration.

The semantics is essentially a continuation-based semantics, but where the continuations have been defunctionalized as abstract machine instructions. We introduce the various instructions as they are needed.

In general, a semantic rule has the form:

$$\frac{\begin{array}{c} precondition_1 \\ \vdots \\ precondition_n \end{array}}{configuration \Rightarrow configuration'}$$

$precondition_1 \ldots precondition_n$ describe operations that must be performed given the information provided in *configuration* in order to produce $configuration'$.

5.3 Entry point of the semantics

The first step of the semantics is to identify the handler to execute.

Service methods

The only service methods are `deploy`, which is used to create a service, and `undeploy`, which is used to end a service.

The following rules define the semantics of `deploy`, which is assumed like all control methods, to succeed. There is one continuation `DEPLOY` ϕ.

$$\frac{\begin{array}{c} address = \langle service \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{undeploy}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, \perp, \texttt{deploy}) = \langle \_, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle \perp, \langle \perp, \emptyset \rangle \rangle \rangle \end{array}}{\mathsf{deploy}(service, \phi, \sigma) \Rightarrow \tau, r, \perp, \langle \texttt{DEPLOY } \phi \rangle \models decls, stmt}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho, \_ \models \langle \texttt{DEPLOY } \phi \rangle, \texttt{SUCCESS}, b \Rightarrow \perp, \sigma'}$$

$$\frac{\begin{array}{c} \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'' \end{array}}{\langle \sigma, address \rangle, \rho, \_ \models \langle \texttt{DEPLOY } \phi \rangle, \texttt{ERROR}, b \Rightarrow \perp, \sigma''}$$

The semantics of `undeploy` simply ends the session. Because a service is not persistent, this ends all subsessions as well.

$$\frac{\begin{array}{c} address = \langle service \rangle \\ \mathsf{set\_persistence}(\sigma, address, persistent) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma'' \end{array}}{\mathsf{undeploy}(service, \phi, \sigma, persistent) \Rightarrow \bot, \sigma''}$$

Registration methods

The only registration methods are `initial REGISTER`, which is used to create a registration, and `medial REGISTER`, which is used to keep the registration alive. An initial `REGISTER` creates an entry for a new registration id, initializes the registration variables, and sets the live bit associated with the registration entry to true. The initial values of the registration variables cannot refer to the branch lists of the message (headers or request URI). A medial `REGISTER` can update the variables associated with the registration.

The following three rules define the semantics of an initial `REGISTER`. There is one continuation: `INITIAL_REG` ϕ.

$$\frac{\begin{array}{c} address = \langle service, rid \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{unregister}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \texttt{initial REGISTER}) = \langle m, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle \end{array}}{\langle \texttt{initial REGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models service \Rightarrow \tau, r, \langle \texttt{INITIAL\_REG } \phi \rangle \models decls, stmt}$$

There are two rules for the continuation `INITIAL_REG` ϕ: one where the registration has succeeded and one where it has failed.

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho \models \langle \texttt{INITIAL\_REG } \phi \rangle, \texttt{/SUCCESS/}resp(resp\_headers), b \Rightarrow \texttt{/SUCCESS/}resp(resp\_headers), \sigma'}$$

$$\frac{\begin{array}{c} \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'' \end{array}}{\langle \sigma, address \rangle, \rho \models \langle \texttt{INITIAL\_REG } \phi \rangle, \texttt{/ERROR/}resp(resp\_headers), b \Rightarrow \texttt{/ERROR/}resp(resp\_headers), \sigma''}$$

The following rule is for a medial `REGISTER`. There is one continuation in this case: `MEDIAL_REG` ϕ.

$$\frac{\begin{array}{c} address = \langle service, rid \rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, \texttt{medial REGISTER}) = \langle m, decls, stmt, envs, \sigma' \rangle \\ \tau = \langle \sigma', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle \end{array}}{\langle \texttt{medial REGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models service \Rightarrow \tau, r, \langle \texttt{MEDIAL\_REG } \phi \rangle \models decls, stmt}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho \models \langle \texttt{MEDIAL\_REG } \phi \rangle, resp, b \Rightarrow resp, \sigma'}$$

The lifetime of a registration is limited by a header associated with the `REGISTER` method (either initial or medial) or a default value. At the time of this expiration, the support for SPL in the underlying platform generates a registration_timeout(*service, rid, ϕ, σ*) message. This message sets the *live* field of the registration to false. If there are not more sessions associated with the registration, then the registration is deleted. If there are still sessions associated with the registration, then the registration is deleted on the completion of a subsequent `BYE` method when there are no remaining live sessions. All of this is taken care of by end_session, because a registration is declared to be persistent.

$$\frac{\begin{array}{c}address = \langle service, rid \rangle \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{true}) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{registration\_timeout}(service,\ rid,\ \phi,\ \sigma) \Rightarrow \bot, \sigma''}$$

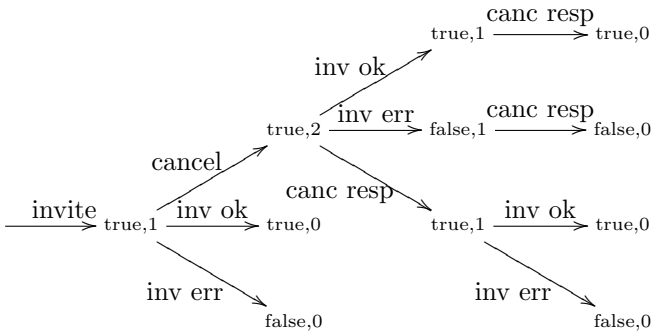
$$\frac{\begin{array}{c}address = \langle service, rid \rangle \\ \mathsf{set\_persistence}(\sigma, address, persistent) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{unregister}(service,\ rid,\ \phi,\ \sigma, persistent) \Rightarrow \bot, \sigma''}$$

Dialog methods

A dialog is initiated by an `INVITE`. As for `REGISTER`, there are two kinds of `INVITE`: initial and medial. There is no final `INVITE`, because that is implemented as `BYE`. If the service includes a `registration` block, then `INVITE` will only been invoked when `REGISTER` has previously been invoked, and the call to `INVITE` will contain information about the associated *rid*. If the service does not include a `registration` block, then the method has only *did* as an argument, and the address is constructed as $\langle service, did \rangle$. This strategy applies to the other dialog methods as well. The semantics of `initial INVITE` is defined by the following rules. There is one continuation: `INITIAL_INVITE` *direction handlers*.

$$\frac{\begin{array}{c}address = \langle service, rid, did \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(address)) = \langle branch \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle branch \rangle, \mathtt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', direction, \mathtt{initial\ INVITE}) = \langle m, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle\end{array}}{\begin{array}{l}\langle \mathtt{initial\ INVITE}(rid, did), direction, rq, headers \rangle, \phi, \sigma \models service \\ \Rightarrow \tau, r, \langle \mathtt{INITIAL\_INVITE}\ \phi \rangle \models decls, stmt\end{array}}$$

Because we do not control the ordering in which messages are transmitted to the service, there are a number of three possibilities, as shown by the following diagram. At each node, the boolean value indicates whether the dialog is live and the integer value indicates the number of open transactions. At any point where the node contains false and 0, the session will be ended when by continue_session.



The rules are as follows:

$$\frac{\begin{array}{c}\mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma'\end{array}}{\begin{array}{l}\langle \sigma, address \rangle, \rho \models \langle \mathtt{INITIAL\_INVITE}\ \phi \rangle, \mathtt{/SUCCESS/}resp\mathtt{(}resp\_headers\mathtt{)}, b \\ \Rightarrow \mathtt{/SUCCESS/}resp\mathtt{(}resp\_headers\mathtt{)}, \sigma'\end{array}}$$

The function $\mathsf{add}(b, branches)$ adds b at the beginning of *branches* if $b \notin branches$ and returns *branches* otherwise.

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{lookup\_branches}(\sigma', address) = branches \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma''\end{array}}{\begin{array}{l}\langle \sigma, address\rangle, \rho \models \langle \texttt{INITIAL\_INVITE}\ \phi\rangle, \texttt{/ERROR/}resp\texttt{(}resp\_headers\texttt{)}, b \\ \Rightarrow \texttt{/ERROR/}resp\texttt{(}resp\_headers\texttt{)}, \sigma''\end{array}}$$

The following rule is for methods that occur within an existing dialog. The method can be `initial ACK`, `medial ACK`, `CANCEL` or `medial INVITE`. There is one continuation: `MEDIAL_INVITE` ϕ.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, method) = \langle m, decls, stmt, envs, \sigma'\rangle \\ \tau = \langle \sigma', address\rangle \qquad r = \langle envs, \langle m, \langle rq, headers\rangle\rangle\rangle\end{array}}{\langle method(rid, did), direction, rq, headers\rangle, \phi, \sigma \models service \Rightarrow \tau, r, \langle \texttt{MEDIAL\_INVITE}\ \phi\rangle \models decls, stmt}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma'\end{array}}{\langle \sigma, address\rangle, \rho \models \langle \texttt{MEDIAL\_INVITE}\ \phi\rangle, resp, b \Rightarrow resp, \sigma'}$$

The following rules are for `BYE`, which ends a dialog. If the registration is still live or there are other dialogs associated with the registration, then we simply delete the dialog from the dialog environment. If the registration is no longer live and there are no other dialogs associated with the registration, then we run the `unregister` code and delete the entire registration (including the dialog). There is one continuation: `BYE` ϕ.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, \texttt{BYE}) = \langle m, decls, stmt, envs, \sigma'\rangle \\ \tau = \langle \sigma', address\rangle \qquad r = \langle envs, \langle m, \langle rq, headers\rangle\rangle\rangle\end{array}}{\langle \texttt{BYE}(rid, did), direction, headers\rangle, \phi, \sigma \models service \Rightarrow \tau, r, \langle \texttt{BYE}\ \phi\rangle \models decls, stmt}$$

If the registration has not ended or the registration has ended and there are dialogs other than the current one, then the registration is preserved and we only end the dialog. Otherwise we end both the dialog and the enclosing registration. This is taken care of $\mathsf{end\_session}$, because registration is persistent.

$$\frac{\begin{array}{c}\mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma''\end{array}}{\langle \sigma, address\rangle, \rho \models \langle \texttt{BYE}\ \phi\rangle, resp, b \Rightarrow resp, \sigma''}$$

The following only occurs when no `initial ACK` appears after an `initial INVITE`.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{uninvite}(service,\ rid,\ did,\ \phi,\ \sigma) \Rightarrow \bot, \sigma''}$$

Default handling

When there is no handler, and thus nothing to do, the semantics just forwards the message as normal. Actually, the semantics does not distinguish between the case where there is no handler and the case where there is a handler but there is no branch. The latter case should never occur.

The following rule is for an `initial REGISTER`:

$$\frac{\begin{array}{c} address = \langle service, rid \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{unregister}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \texttt{initial REGISTER}) = \langle \bot, \bot, \bot, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad \rho = \langle envs, \langle \bot, headers \rangle, \langle \rangle \rangle \end{array}}{\begin{array}{c} \langle \texttt{initial REGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models service \\ \Rightarrow \tau, \rho, \langle \texttt{INITIAL\_REG } \phi \rangle \models \texttt{return forward <requestURI> branch default} \end{array}}$$

The rules for the other methods are similar, *i.e.*, we pretend that the statement is `return forward <requestURI> branch default` and then use the original continuation.

5.4 Semantics of handler bodies

A handler can only declare local variables at the root of the body. The rules here evaluate such declarations and then continue with the semantics of statements. Note that even the variables associated with `when` must be declared at the top level, and thus `when` is implemented by assignments. This implies that all `when`-declared variables (like all other local declarations) have to be given unique names.

$$\frac{\begin{array}{c} r = \langle envs, message\_info \rangle \\ \langle envs, message\_info, \emptyset \rangle \models decls \Rightarrow local\_env \\ \rho = \langle envs, message\_info, local\_env \rangle \end{array}}{\tau, r, s \models decls, stmt \Rightarrow \tau, \rho, s \models stmt}$$

5.5 Semantics of statements

Sequence

We assume that a sequence has one of the following forms:

- `{}`
- `{`$stmt_1$ $stmt_2$`}`

These forms are not particularly convenient for programming, but it is easy to turn any sequence statement into one of these forms, either by adding braces or by dropping braces that only enclose one statement.

$$\tau, \rho, s \models \texttt{\{\}} \Rightarrow \tau, \rho \models s, \bot, \bot$$

In the case of a sequence of two statements, there is one continuation: `SEQ` $stmt_2$.

$$\tau, \rho, s \models \texttt{\{}stmt_1 \ \ stmt_2\texttt{\}} \Rightarrow \tau, \rho, (\texttt{SEQ} \ \ stmt_2) :: s \models stmt_1$$

$$\tau, \rho \models (\texttt{SEQ} \ \ stmt_2) :: s, \_, \_ \Rightarrow \tau, \rho, s \models stmt_2$$

Assignment

Note that identifiers and sip headers are unified; a sip header is just an identifier that is found in *headers*.

In the case of an assignment, there is one continuation: `ASSIGN` x.

$$\tau, \rho, s \models x \ \texttt{=} \ exp \Rightarrow \tau, \rho, (\texttt{ASSIGN} \ \ x) :: s \models exp$$

$$\tau, \rho \models (\texttt{ASSIGN} \ \ x) :: s, v \Rightarrow \tau, \mathsf{update}(x, v, \rho) \models s, \bot, \bot$$

$\mathsf{update}(x, v, \langle envs, \langle m, headers \rangle, local\_env \rangle) = \mathsf{update}_l(x, v, envs, \langle m, headers \rangle, local\_env)$, which is defined as follows:

$$\frac{x \in \mathsf{dom}(\mathit{local\_env})}{\mathsf{update}_l(x, v, \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}) = \langle \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}[x \mapsto v]\rangle}$$

$$\frac{x \notin \mathsf{dom}(\mathit{local\_env}) \quad \mathsf{update}_h(x, v, \mathit{envs}, \mathit{message\_info}) = \langle \mathit{envs}', \mathit{message\_info}'\rangle}{\mathsf{update}_l(x, v, \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}) = \langle \mathit{envs}', \mathit{message\_info}', \mathit{local\_env}\rangle}$$

$$\frac{x = m}{\mathsf{update}_h(x, v, \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle) = \langle \mathit{envs}, \langle m, \langle \mathit{rq}, v\rangle\rangle\rangle}$$

$$\frac{x \neq m \qquad x \in \mathsf{dom}(\mathit{headers})}{\mathsf{update}_h(x, v, \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle) = \langle \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}[x \mapsto v]\rangle\rangle\rangle}$$

$$\frac{x \neq m \qquad x \notin \mathsf{dom}(\mathit{headers}) \qquad \mathsf{update}_e(x, v, \mathit{envs}) = \mathit{envs}'}{\mathsf{update}_h(x, v, \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle) = \langle \mathit{envs}', \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle\rangle}$$

$$\frac{x \in \mathsf{dom}(\mathit{env})}{\mathsf{update}_e(x, v, \mathit{envs} +\!\!+ \langle \mathit{env}\rangle) = \mathit{envs} +\!\!+ \langle \mathit{env}[x \mapsto v]\rangle} \qquad \frac{x \notin \mathsf{dom}(\mathit{env}) \qquad \mathsf{update}_e(x, v, \mathit{envs}) = \mathit{envs}'}{\mathsf{update}_e(x, v, \mathit{envs} +\!\!+ \langle \mathit{env}\rangle) = \mathit{envs}' +\!\!+ \langle \mathit{env}\rangle}$$

Return

Return is associated with one continuation: `RETURN` b.

$$\tau, \rho, s \models \texttt{return}\ \ \mathit{exp}\ \ \texttt{branch}\ \ b\texttt{;} \Rightarrow \tau, \rho, (\texttt{RETURN}\ \ b) :: s \models \mathit{exp}$$

$$\tau, \rho \models (\texttt{RETURN}\ \ b) :: s, \mathit{resp} \Rightarrow \tau, \rho \models s, \mathit{resp}, b$$

The following returns no value but changes the branch.

$$\tau, \rho, s \models \texttt{return branch}\ \ b\texttt{;} \Rightarrow \tau, \rho \models s, \bot, b$$

If

If is associated with one continuation: `IF` stmt_1 stmt_2.

$$\tau, \rho, s \models \texttt{if (}\mathit{exp}\texttt{)}\ \ \mathit{stmt}_1\ \ \texttt{else}\ \ \mathit{stmt}_2 \Rightarrow \tau, \rho, (\texttt{IF}\ \ \mathit{stmt}_1\ \ \mathit{stmt}_2) :: s \models \mathit{exp}$$

$$\tau, \rho \models (\texttt{IF}\ \ \mathit{stmt}_1\ \ \mathit{stmt}_2) :: s, \mathsf{true} \Rightarrow \tau, \rho, s \models \mathit{stmt}_1$$

$$\tau, \rho \models (\texttt{IF}\ \ \mathit{stmt}_1\ \ \mathit{stmt}_2) :: s, \mathsf{false} \Rightarrow \tau, \rho, s \models \mathit{stmt}_2$$

When

$$\tau, \rho, s \models \texttt{when}\ \ \mathit{exp}\ \ \texttt{(}\mathit{header\_name}_1\ \ \mathit{type}_1\ \ x_1\texttt{, . . . ,}\ \ \mathit{header\_name}_n\ \ \mathit{type}_n\ \ x_n\texttt{)}\ \ \mathit{stmt}_1\ \ \texttt{else}\ \ \mathit{stmt}_2$$
$$\Rightarrow \tau, \rho, (\texttt{WHEN}\ \ \langle \mathit{header\_name}_1\ \ \mathit{type}_1\ \ x_1, \ldots, \mathit{header\_name}_n\ \ \mathit{type}_n\ \ x_n\rangle\ \ \mathit{stmt}_1\ \ \mathit{stmt}_2) :: s \models \mathit{exp}$$

$$\frac{\{\mathit{header\_name}_1, \ldots, \mathit{header\_name}_n\} \subseteq \mathsf{dom}(\mathit{headers}) \qquad \rho' = \mathsf{update}(x_1, \mathsf{coerce}(\mathit{headers}(\mathit{header\_name}_1), \mathit{type}_1), \ldots \mathsf{update}(x_n, \mathsf{coerce}(\mathit{headers}(\mathit{header\_name}_n), \mathit{type}_n), \rho)\ldots)}{\tau, \rho \models (\texttt{WHEN}\ \ \langle \mathit{header\_name}_1\ \ \mathit{type}_1\ \ x_1, \ldots, \mathit{header\_name}_n\ \ \mathit{type}_n\ \ x_n\rangle\ \ \mathit{stmt}_1\ \ \mathit{stmt}_2) :: s, \langle \mathit{rq}, \mathit{headers}\rangle \Rightarrow \tau, \rho', s \models \mathit{stmt}_1}$$

$$\frac{\{header\_name_1, \ldots, header\_name_n\} \not\subseteq \mathsf{dom}(headers)}{\tau, \rho \models (\texttt{WHEN}\ \langle header\_name_1\ type_1\ x_1, \ldots, header\_name_n\ type_n\ x_n\rangle\ \ stmt_1\ \ stmt_2) :: s, \langle rq, headers\rangle \Rightarrow \tau, \rho, s \models stmt_2}$$

$$\frac{\begin{array}{c}\{header\_name_1, \ldots, header\_name_n\} \subseteq \mathsf{dom}(resp\_headers) \\ \rho' = \mathsf{update}(x_1, \mathsf{coerce}(resp\_headers(header\_name_1), type_1), \ldots \\ \mathsf{update}(x_n, \mathsf{coerce}(resp\_headers(header\_name_n), type_n), \rho)\ldots)\end{array}}{\begin{array}{l}\tau, \rho \models (\texttt{WHEN}\ \langle header\_name_1\ type_1\ x_1, \ldots, header\_name_n\ type_n\ x_n\rangle\ \ stmt_1\ \ stmt_2) :: s, code(resp\_headers) \\ \quad \Rightarrow \tau, \rho', s \models stmt_1\end{array}}$$

$$\frac{\{header\_name_1, \ldots, header\_name_n\} \not\subseteq \mathsf{dom}(resp\_headers)}{\begin{array}{l}\tau, \rho \models (\texttt{WHEN}\ \langle header\_name_1\ type_1\ x_1, \ldots, header\_name_n\ type_n\ x_n\rangle\ \ stmt_1\ \ stmt_2) :: s, code(resp\_headers) \\ \quad \Rightarrow \tau, \rho, s \models stmt_2\end{array}}$$

5.6 Semantics of expressions

We consider a simplified set of expressions in which there are no `with` expressions, identifiers and sip headers are unified as described above for assignments, the only binary operator is `==`, there are no unary operators, there are no `pop` expressions, and there are no function calls. An expression and a `branch` are obligatory in every `(parallel) forward`. The expression must evaluate to a list. We add an expression for making a list out of the value of a single expression so that the value of `requestURI` can be put into a list to make a suitable argument for the trivial case.

An expression can reinvoke the underlying machine, via a `forward` or `parallel forward`. For this, we have to bring σ up to date with respect to the changes that have occurred in *envs*. When control returns to the semantics, we similarly have to obtain the new *envs*. For these operations, we have to keep the environments separated. The semantics of expressions also needs to know σ, and the current *address*.

Forward

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers\rangle\rangle\end{array}}{\langle\sigma, address\rangle, \rho, s \models \texttt{forward} \Rightarrow \mathsf{forward}(rq, (\texttt{FORWARD2}\ \ address\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')}$$

$$\tau, \rho, s \models \texttt{forward}\ \ exp \Rightarrow \tau, \rho, \texttt{FORWARD1} :: s \models exp$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers\rangle\rangle\end{array}}{\begin{array}{l}\langle\sigma, address\rangle, \rho \models \texttt{FORWARD1} :: s, v \\ \quad \Rightarrow \mathsf{forward}(v, (\texttt{FORWARD2}\ \ address\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_envs}(\sigma, address) = envs \\ \tau = \langle\sigma, address\rangle \qquad \rho = \langle envs, message\_info, local\_env\rangle\end{array}}{\mathsf{forward\_response}(resp, (\texttt{FORWARD2}\ \ address\ \ message\_info\ \ local\_env) :: s, \sigma) \Rightarrow \tau, \rho \models s, resp}$$

Parallel forward

$$\frac{\rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle}{\begin{array}{l}\tau, \rho, s \models \texttt{parallel forward} \\ \quad \Rightarrow \mathsf{forward}(rq, (\texttt{FORWARD2}\ \ address\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}}$$

$$\tau, \rho, s \models \texttt{parallel forward}\ \ exp \Rightarrow \tau, \rho, \texttt{PFORWARD} :: s \models exp$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle\end{array}}{\begin{array}{l}\langle \sigma, address \rangle, \rho \models \texttt{PFORWARD} :: s, v \\ \quad \Rightarrow \mathsf{parallel\_forward}(v, (\texttt{FORWARD2}\ \ service\ \ rid\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}}$$

The call to parallel_forward eventually produces a call to forward_response. The semantics of this is shown above, in the treatment of `forward`.

List creation

List creation is associated with one continuation: `LIST`.

$$\tau, \rho, s \models \texttt{<}\ \ exp\ \ \texttt{>} \Rightarrow \tau, \rho, \texttt{LIST} :: s \models exp$$

$$\tau, \rho \models \texttt{LIST} :: s, v \Rightarrow \tau, \rho \models s, \langle v \rangle$$

Equality

Equality is associated with two continuations: `EQ1` exp_2 and `EQ2` v_1.

How does equal work when one argument is a response and the other is a code constant?

$$\tau, \rho, s \models exp_1\ \ \texttt{==}\ \ exp_2 \Rightarrow \tau, \rho, (\texttt{EQ1}\ \ exp_2) :: s \models exp_1$$

$$\tau, \rho \models (\texttt{EQ1}\ \ exp_2) :: s, v_1 \Rightarrow \tau, \rho, (\texttt{EQ2}\ \ v_1) :: s \models exp_2$$

$$\tau, \rho \models (\texttt{EQ2}\ \ v_1) :: s, v_2 \Rightarrow \tau, \rho \models s, v_1 == v_2$$

Identifier

$$\tau, \rho, s \models x \Rightarrow \tau, \rho \models s, \mathsf{lookup}(x, \rho)$$

$\mathsf{lookup}(x, \langle envs, \langle m, headers \rangle, local\_env \rangle) = \mathsf{lookup}_l(x, envs, \langle m, headers \rangle, local\_env)$, which is defined as follows:

$$\frac{x \in \mathsf{dom}(\mathit{local\_env})}{\mathsf{lookup}_l(x, \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}) = \mathit{local\_env}(x)}$$

$$\frac{x \notin \mathsf{dom}(\mathit{local\_env})}{\mathsf{lookup}_l(x, \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}) = \mathsf{lookup}_h(x, \mathit{envs}, \mathit{message\_info})}$$

$$\frac{x = m}{\mathsf{lookup}_h(x, \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle) = \langle \mathit{rq}, \mathit{headers}\rangle} \qquad \frac{x \neq m \qquad x \in \mathsf{dom}(\mathit{headers})}{\mathsf{lookup}_h(x, \mathit{envs}, \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle) = \mathit{headers}(x)}$$

$$\frac{x \neq m \qquad x \notin \mathsf{dom}(\mathit{headers})}{\mathsf{lookup}_h(x, \mathit{envs}, \langle \mathit{rq}, \mathit{headers}\rangle) = \mathsf{lookup}_e(x, \mathit{envs})}$$

$$\frac{x \in \mathsf{dom}(\mathit{env})}{\mathsf{lookup}_e(x, \mathit{envs} +\!\!+ \langle \mathit{env}\rangle) = \mathit{env}(x)} \qquad \frac{x \notin \mathsf{dom}(\mathit{env})}{\mathsf{lookup}_e(x, \mathit{envs} +\!\!+ \langle \mathit{env}\rangle) = \mathsf{lookup}_e(x, \mathit{envs})}$$

Constant

$$\tau, \rho, s \models c \Rightarrow \tau, \rho \models s, c$$

RequestURI

$$\frac{\rho_{\mathsf{message\_info}} = \langle m, \langle \mathit{rq}, \mathit{headers}\rangle\rangle}{\tau, \rho, s \models \texttt{requestURI} \Rightarrow \tau, \rho \models s, \mathit{rq}}$$

With

We assume there is only one binding, as multiple bindings can be implemented by nested `with`s. There are two continuations: `WITH1` *field* exp_2 and `WITH2` *field resp*.

$$\tau, \rho, s \models \mathit{exp}_1 \ \texttt{with}\ \mathit{field}\ \texttt{=}\ \mathit{exp}_2 \Rightarrow \tau, \rho, (\texttt{WITH1}\ \ \mathit{field}\ \ \mathit{exp}_2) :: s \models \mathit{exp}_1$$

$$\tau, \rho \models (\texttt{WITH1}\ \ \mathit{field}\ \ \mathit{exp}_2) :: s, v \Rightarrow \tau, \rho, (\texttt{WITH2}\ \ \mathit{field}\ \ v) :: s \models \mathit{exp}_2$$

$$\tau, \rho \models (\texttt{WITH2}\ \ \mathit{field}\ \ \mathit{code}(\mathit{resp\_headers})) :: s, v \Rightarrow \tau, \rho \models s, \mathit{code}(\mathit{resp\_headers}[\mathit{field} \mapsto v])$$

$$\tau, \rho \models (\texttt{WITH2}\ \ \mathit{field}\ \ \langle \mathit{rq}, \mathit{headers}\rangle) :: s, v \Rightarrow \tau, \rho \models s, \langle \mathit{rq}, \mathit{headers}\rangle[\mathit{field} \mapsto v]$$

5.7 Semantics of declarations

The semantics of declarations is defined as below. Each expression is evaluated with respect to the empty continuation, because we know that there is no `forward` in a declaration, even the declaration of a local variable. This is always invoked with $\emptyset$ in the *local_env* position of ρ.

$$\langle \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}\rangle \models \langle\rangle \Rightarrow \mathit{local\_env}$$

$$\frac{\begin{array}{c}\bot, \rho, \langle\rangle \models \mathit{exp}_1 \Rightarrow^* \_, \_, \_ \models \langle\rangle, v_1 \\ \rho = \langle \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}\rangle \\ \langle \mathit{envs}, \mathit{message\_info}, \mathit{local\_env}[x_1 \mapsto v_1]\rangle \models \langle\langle x_2, \mathit{exp}_2\rangle, \ldots\rangle \Rightarrow \mathit{env}\end{array}}{\rho \models \langle\langle x_1, \mathit{exp}_1\rangle, \langle x_2, \mathit{exp}_2\rangle, \ldots\rangle \Rightarrow \mathit{env}}$$