Backwards Type Analysis of Asynchronous Method Calls [⋆]

Einar Broch Johnsen and Ingrid Chieh Yu [*]

Department of Informatics, University of Oslo, Norway

Abstract

Asynchronous method calls have been proposed to better integrate object orientation with distribution. In the Creol language, asynchronous method calls are combined with so-called processor release points in order to allow concurrent objects to adapt local scheduling to network delays in a very flexible way. However, asynchronous method calls complicate the type analysis by decoupling input and output information for method calls, which can be tracked by a type and effect system. Interestingly, backwards type analysis simplifies the effect system considerably and allows analysis in a single pass. This paper presents a kernel language with asynchronous method calls and processor release points, a novel mechanism for local memory deallocation related to asynchronous method calls, an operational semantics in rewriting logic for the language, and a type and effect system for backwards analysis. Source code is translated into runtime code as an effect of the type analysis, automatically inserting inferred type information in method invocations and operations for local memory deallocation in the process. We establish a subject reduction property, showing in particular that method lookup errors do not occur at runtime and that the inserted deallocation operations are safe.

Key words: distributed object-oriented systems, futures, type and effect systems

1 Introduction

In the distributed setting, the object-oriented programming model may be criticized for its tight coupling of communication and synchronization, as found

[⋆] This work is in the context of the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (`http://credo.cwi.nl`).
[*] Corresponding author.
Email addresses: `einarj@ifi.uio.no` (Einar Broch Johnsen), `ingridcy@ifi.uio.no` (Ingrid Chieh Yu).

in, for example, remote procedure calls. Creol is a novel object-oriented language which targets distributed systems by combining *asynchronous method calls* with *processor release points* inside concurrent objects [19]. The language is class-based, supports inheritance, and object interaction happens through method calls only. Processor release points influence the implicit control flow inside concurrent objects. This way, the object may temporarily suspend its current activity and proceed with another enabled activity. This allows the execution to adapt to network delays in a flexible way, and objects to dynamically change between active and reactive behavior (client and server). Asynchronous method calls decouple the invocation and the return of method calls as seen from the caller. As a result the execution may continue after an external method is invoked, introducing a notion of concurrency similar to that of *futures* [6, 11, 13, 23, 41]. This decoupling is *linear*; i.e., exactly one method invocation is required for every return and at most one return is allowed for every invocation. For asynchronous method calls in Creol, processor release points may be associated with method returns, extending this notion of concurrency. Baker and Hewitt observed that futures need not be used in the scope in which they are bound, leading to memory leakage [3]. A similar issue arises with returns to asynchronous calls in Creol, as a caller may ignore method replies in some or all execution branches after the method invocation.

A type system for Creol is presented in [20]. However asynchronous calls complicate the type analysis by decoupling the call's input and output information, leading to a complex type and effect system [32] to track information during type analysis. Furthermore, the memory leakage due to method replies is not addressed. The type system can be significantly simplified for backwards type analysis, allowing each asynchronous call to be analyzed directly at the call site. This paper develops a type based translation of source programs into runtime code, based on backwards type analysis. The main contributions of this paper are:

- a novel light-weight mechanism for local garbage collection, addressing redundant method replies by explicit deallocations in the runtime code. The mechanism is integrated in a simplified kernel language which focuses on Creol's asynchronous method call mechanism, and in an executable operational semantics for this kernel language in rewriting logic;
- a type and effect system for the language, designed for backwards analysis, which directly translates source code into runtime code during type analysis;
- the exploitation of the backwards analysis to simplify the extraction of runtime signatures for asynchronous method calls;
- the exploitation of type analysis to automatically insert deallocation operations for local garbage collection, addressing the memory leakage problem caused by redundant method replies; and
- subject reduction showing the correctness of late binding and of the deallocation mechanism.

Whereas a traditional tracing garbage collector cannot free memory space that is reachable from the state of an object [21], we exploit type and effect analysis to identify method replies that are semantically garbage in specific execution paths and insert operations to explicitly free memory in those paths. The backwards analysis allows signature extraction and operations for garbage collection to be inserted in the code as it is type checked, in a single pass. The type system is given an algorithmic formulation and is directly implementable.

Paper overview. The remainder of this paper is structured as follows. Section 2 defines the kernel language with asynchronous method calls and Section 3 defines a type based translation into runtime code. Section 4 presents the operational semantics and Section 5 the runtime type system and its properties, Section 6 discusses related work, and Section 7 concludes the paper.

2 Creol: A Language for Distributed Concurrent Objects

An object in Creol has an encapsulated state on which various processes execute. A process corresponds to the activation of one of the object's methods. Objects are concurrent in the sense that each object has a processor dedicated to executing the processes of that object, so processes in different objects execute in parallel. The state is encapsulated in the sense that external manipulation of the object state is indirect by means of calls to the object's methods. Only one process may be active in an object at a time; the other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking causes process execution to stop, but does not hand control over to a suspended process. Releasing a process suspends the execution of that process and reschedules control to another (suspended) process. Thus, if a process is blocked there is no execution in the object, whereas if a process is released another process in the object may execute. Although processes need not terminate, the object may combine the execution of several processes using release points within method bodies. At a release point, the active process may be released and a suspended process may be activated.

A process which makes a remote method call must *wait* for the return of the call before proceeding with its activity. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. In an unreliable environment where communication can disappear, the process can even be permanently blocked (and the object deadlocked). Asynchronous method calls allow the process to be either blocked or released by introducing a processor release point between the invocation of the method and the access to its return value. Release points, expressed using Boolean guards, influence the implicit control flow inside objects by allowing process release when the guard evaluates to `false`. This way, processes may choose between blocking

$$
\begin{array}{ll}
P & ::= \overline{L}\ \{\overline{T\ x}; s_p\} \\
L & ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{T\ f}; \overline{M}\} \\
M & ::= T\ m\ (\overline{T\ x})\{\overline{T\ x}; s_p; \texttt{return}\ e\} \\
s_p & ::= s \mid s_p; s_p \mid s_p \,\Box\, s_p \mid s_p \,|||\, s_p \mid t!e.m(\overline{e}) \\
s & ::= v := e \mid v := \texttt{new}\ C() \mid \texttt{await}\ g \mid \texttt{release} \mid \texttt{skip} \mid t?(v) \\
g & ::= b \mid t? \mid g \wedge g
\end{array}
$$

Figure 1. The language syntax. Here, program variables v may be fields (f) or local variables (x), a type T may be a class name C, a `Bool`, or a `Label`, b is an expression of type `Bool` and t is a variable of type `Label`.

and releasing control while waiting for the reply to a method call. Remark that the use of release points makes it straightforward to combine active (e.g., nonterminating) and reactive processes in an object. Thus, an object may behave both as a client and as a server for other objects without the need for an active loop to control the interleaving of these different roles.

Syntax. A kernel subset of Creol is given in Figure 1, adapting the syntax of Featherweight Java [17]. For a syntactic construct c, we denote by $\overline{c}$ a list of such constructs. We assume given a simple functional language of expressions e without side effects. A program P is a list of class definitions followed by a method body. A class extends a superclass, which may be `Object`, with additional fields $\overline{f}$, of types $\overline{T}$, and methods $\overline{M}$. A method's formal parameters and local variables $\overline{x}$ of types $\overline{T}$, are declared at the start of the method body. The self reference `this` provides access to the object running the current method and `return` e dispatches the value of e to the method's caller. The statement `release` is an unconditional process release point. Conditional release points are written `await` g, where g may be a conjunction of Boolean guards b and method reply guards $t?$ (for some label variable t). An asynchronous call is written $t!e.m(\overline{e})$, where t provides a reference to the call, e is an object expression (when e evaluates to `this`, the call is local), m a method name, and $\overline{e}$ an expression list with the actual in-parameters supplied to the method. A blocking reply assignment is written $t?(v)$; here, the result from the call is assigned to v. A nonblocking call combines reply guards with a reply assignment; e.g., in $t!e.m(\overline{e});$ `await` $t?; t?(v)$ the processor may be released while waiting for the reply, allowing other processes to execute. A synchronous call is obtained by immediately succeeding an invocation by a (blocking) reply assignment on the same label, e.g., $t!e.m(\overline{e}); t?(v)$. Reply assignments can be omitted if return values are not needed. Finally, $\overline{s}_1 \,\Box\, \overline{s}_2$ is the nondeterministic choice and $\overline{s}_1 \,|||\, \overline{s}_2$ is nondeterministic merge between $\overline{s}_1$ and $\overline{s}_2$, defined as $\overline{s}_1; \overline{s}_2 \,\Box\, \overline{s}_2; \overline{s}_1$. Informal explanations of the language constructs are given in Examples 1 and 2 below.

The correspondence between the label variables associated with method invocations, reply guards, and reply assignments can be made more precise by adapting the notion of live variables [27] to labels. Reply guards and assignments are read operations on a label variable t, and invocations are write operations. Say that t is *live* before a statement list $\overline{s}$ if there exists an execution path through $\overline{s}$ where a read operation on t precedes the next write operation on t. Note that reply assignments are *destructive* read operations; for a label t, only a write operation on t may succeed a reply assignment on t.

Definition 1 Let $\overline{s}$ be a list of program statements and t a label. Define $\mathit{live}(t, \overline{s})$ by induction over statement lists as follows:

$$\begin{aligned}
&\mathit{live}(t, \epsilon) = \texttt{false}\\
&\mathit{live}(t, t?(v); \overline{s}) = \texttt{true}\\
&\mathit{live}(t, \texttt{await}\ g; \overline{s}) = \texttt{true} \qquad \text{if } g \text{ contains } t?\\
&\mathit{live}(t, t!o.m(\overline{e}); \overline{s}) = \texttt{false}\\
&\mathit{live}(t, \overline{s}_1 \,\square\, \overline{s}_2; \overline{s}) = \mathit{live}(t, \overline{s}_1; \overline{s}) \vee \mathit{live}(t, \overline{s}_2; \overline{s})\\
&\mathit{live}(t, s; \overline{s}') = \mathit{live}(t, \overline{s}') \qquad \text{otherwise}
\end{aligned}$$

Here and throughout the paper, "otherwise" denotes the remaining cases for an inductive definition. Given a statement list $t!e.m(\overline{e}); \overline{s}$, the return values from the method call $t!e.m(\overline{e})$ will never be used if $\neg\mathit{live}(t, \overline{s})$. In this case the asynchronous method call gives rise to memory leakage through passive storage of the future containing the method reply, as observed by Baker and Hewitt [3]. This information is exploited in Section 3 to automatically insert an explicit instruction $\texttt{free}(t)$ for the *runtime deallocation* of the future when this memory leakage can be statically detected be the type analysis.

Example 1 We consider a peer-to-peer file sharing system consisting of nodes distributed across a network. In the example a node plays both the role of a server and of a client. A node may request a file from another node in the network and download it as a series of packets until the file download is complete. As nodes may appear and disappear in the unstable network, the connection to a node may be blocked, in which case the download should automatically resume if the connection is reestablished.

In the `Node` class (Figure 2), the methods *reqFile*, *enquire*, and *getPacket* requests the file associated with a file name from a node, enquires which files are available from a node, and fetches a particular packet in a file transmission, respectively. To motivate the use of asynchronous method calls and release points, we consider the method *availFiles* in detail. The method takes as formal parameter a list *nList* of nodes and, for each node, finds the files that may be downloaded from that node. The method returns a list of pairs, where each pair contains a node name and a list of (available) files. All method calls are *asynchronous*. Therefore, after the first call, identified by label t_1, the rest of the node list may be explored recursively, without waiting for the reply to the first call. *Process release points* ensure that if a node temporarily becomes unavailable, the process is suspended and may resume at any time after the

```
class Node{
 DB $db$;
 List[Str] $enquire()$ {...}
 List[Data] $getPacket$(Str $fId$; Nat $pNbr$) {...}
 Str $reqFile$(Node $nId$; Str $fId$) {...}

 List[Node × List[Str]] $availFiles$(List[Node] $nList$) {
   Label$t_1$; Label$t_2$; List[Str]$fList$; List[Node × List[Str]] $files$;
   if $(nList = empty)$ then $files := empty$
   else $t_1!hd(nList).enquire()$; $t_2!this.availFiles(tl(nList))$;
    await $t_1? \wedge t_2?$; $t_1?(fList)$; $t_2?(files)$; $files := ((hd(nList), fList); files)$ fi;
   return $files$}
}
```

Figure 2. A class capturing nodes in a peer-to-peer network

node becomes available again. Thus, if the statement `await` $t_1? \wedge t_2?$ evaluates to false, the processor will not be blocked, and other processes may execute. A node may have several interleaved activities: several downloads may be processed simultaneously with uploads to other nodes, etc. If `await` $t_1? \wedge t_2?$ is enabled, the return value *files* is assigned to the out parameter of the caller (and retrieved by $t_2?(files)$), which completes the process.

While asynchronous invocations and process release points allow objects to dynamically change between active and reactive behavior, nondeterministic choice and merge statements allow different tasks within a process to be selected based on how communication with other objects actually occur. A process may take advantage of delays and adapt to the environment without yielding control to other processes. A typical computational pattern in many wide-area applications is to acquire data from one or more remote services, manipulate these data, and invoke other remote services with the results. We show how Creol's high-level branching structures can be applied in wide-area computing through the following examples, adapted from [25].

Example 2 Consider services which provide news at request. These may be modeled by a class `News` offering a method *news* to the environment, which for a given date returns an XML document with the news of that day as presented by a “site”. Let *CNN* and *BBC* be two such sites; i.e., two variables bound to instances of the class `News`. By calling $CNN.news(d)$ or $BBC.news(d)$ the news for a specified date d will be downloaded. Another service distributes emails to clients at request, modeled by a class `Email` with a method $send(m, a)$ where

m is some message content and a is the address of a client. We define a class

```
class NewsService {
 News $CNN$; Email $email$;
 Bool $newsRelay$(Date $d$, Client $a$){
   XML $v$, Label $t$;  $CNN :=$ new News();  $t!CNN.news(d)$;  await $t?$;  $t?(v)$;
   $email :=$ new Email();  $!email.send(v, a)$;  return true }
}
```

In this class, a method relays news from CNN for a given date d to a given client a by invoking the *send* service of the *email* object with the result returned from the call to *CNN.news*, if *CNN* responds. For simplicity, the method returns `true` upon termination. Note that an object executing a *newsRelay* process is not blocked while waiting for the object to respond. Instead the process is suspended until the response has arrived. Once the object responds, the process becomes enabled and may resume its execution, forwarding the news by email. If the object never responds the object may proceed with its other activity, only the suspended process will never become enabled.

Now consider the case where a client wants news from both objects *CNN* and *BBC*. Because there may be significant delays until an object responds, the client desires to have the news from each object relayed whenever it arrives. This is naturally modeled by the merge operator, as in the following method:

```
Bool $newsRelay2$(Date $d$, Client $a$) {
  XML $v$, Label $t$, Label $t'$;
  $t!CNN.news(d)$;  $t'!BBC.news(d)$;  (await $t?$;  $t?(v)$;  $t!email.send(v, a)$)
   $|||$ (await $t'?$;  $t'?(v)$;  $t'!email.send(v, a)$);  return true }
```

The merge operator allows the news pages from *BBC* and *CNN* to be forwarded to the *email.send* service once they are received. If neither service responds, the whole process is suspended and can be activated again when a response is received. By executing the method, at most two emails are sent.

If the first news page available from either *CNN* or *BBC* is desired, the non-deterministic choice operator may be used to invoke the *email.send* service with the result received from either *CNN* or *BBC*, as in the following method:

```
Bool $newsRelay3$(Date $d$, Client $a$){
  XML $v$, Label $t$, Label $t'$;  $t!CNN.news(d)$;  $t'!BBC.news(d)$;
  ((await $t?$;  $t?(v)$) $\Box$ (await $t'?$;  $t'?(v)$));  $t!email.send(v, a)$;  return true }
```

Here news is requested from both news objects, but only the first response will be relayed. The latest arriving response is ignored.

Asynchronous method calls add flexibility to execution in the distributed setting but complicate the type analysis. For asynchronous calls the type information provided at the call site is not sufficient to ensure the correctness of method binding as the out-parameter type is unavailable. In Creol, the correspondence between in- and out-parameters is controlled by label values. The combination of branching and asynchronous calls further complicates the type analysis as there may be several potential invocation and reply pairs associated with a label value at a point in the execution. For example, consider the statement list $(t!o.m(\overline{e}); \ldots \quad \Box \quad t!o'.m'(\overline{e}'); \ldots); \ldots; \;\; (t?(v) \Box t?(v'))$ and assume that '$\ldots$' contains no invocations nor replies on the label t. In this case we say that each of the two invocations on t *reaches* each of the two reply assignments, depending on the execution path. Consequently, this use of nondeterministic choice gives us four possible invocation and reply pairs and it is nondeterministic which pair will be evaluated at runtime. In contrast, in $t!o.m(\overline{e}); \; t!o'.m'(\overline{e}'); \; t?(v)$ the second call *redefines* the binding of label t. Here only $t!o'.m'(\overline{e}')$ reaches $t?(v)$, and the reply to the first call cannot be accessed. (It would result in memory leakage unless the first method reply is deallocated). The type system developed in this paper derives a signature for each call such that runtime configurations are well-typed independent of the selected execution branch, and inserts this signature in the runtime code.

3 Type Based Translation

Type analysis transforms source programs P into runtime code P_r. For this purpose a type and effect system is used to statically derive signatures for asynchronous method invocations, inferring the type of the out-parameter from constraints imposed by reply assignments, to ensure that late binding succeeds at runtime. Furthermore in order to efficiently remove redundant method returns, the type system ensures that

- reply assignments $t?(v)$ are type safe;
- reply guards `await` $t?$ are type safe;
- method returns are not deallocated if they can be read later; and
- there is no terminating execution path in which a method return is neither deallocated nor destructively read

By type safety for $t?(v)$, we mean that there is an invocation on t which reaches this reply assignment in every execution path and that v can be type correctly assigned the return values from any such invocation $t!e.m(\overline{e})$. Recall that reply assignments are destructive in the sense that a reply assignment on t consumes an invocation on t. Given a well-typed statement list $t!e.m(\overline{e}); \; t?(v); \; \overline{s}$, the first invocation $t!e.m(\overline{e})$ does not reach any statement in $\overline{s}$. By type safety for `await` $t?$, we mean that there is an invocation on t which reaches this reply guard in every execution path. Finally, the deallocation instruction `free`(t) may only occur at a program point where t is not live but where this instruction is reachable by an invocation on t in every execution path.

$$
\begin{array}{ll}
P_r & ::= \overline{L_r}\ \langle s_r, sub\rangle \\
L_r & ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{sub, \overline{M_r}\} \\
M_r & ::= T\ m\ (\overline{T}\ \overline{x})\langle s_r, sub\rangle \\
sub & ::= \overline{v \mapsto val} \\
s_r & ::= s \mid s_r;\ s_r \mid s_r \Box s_r \mid t!e.m(T \to T, \overline{e}) \mid \texttt{free}(\overline{t}) \mid \texttt{return}\ e \\
s & ::= v := e \mid v := \texttt{new}\ C() \mid \texttt{await}\ g \mid \texttt{release} \mid \texttt{skip} \mid t?(v) \\
val & ::= oid \mid \texttt{default}(T) \mid b \mid mid
\end{array}
$$

Figure 3. The syntax for runtime code. The syntax for s is as given in Figure 1.

The syntax for runtime code is given in Figure 3. A state *sub* is a ground substitution mapping variable v to values *val*, which include identifiers *oid* for objects and *mid* for method calls. Let $\texttt{default}(T)$ denote some value of type T. In contrast to source programs, the invocation $t!e.m(T \to T, \overline{e})$ has been expanded with type information and there is an explicit statement $\texttt{free}(\overline{t})$ which allows the deallocation of messages corresponding to the labels $\overline{t}$.

Let $(\mathcal{T}, \preceq)$ be a poset of nominal types with `Data` as its top and `Error` as its bottom element; $\mathcal{T}$ includes `Bool`, `Label`, and class names, and for all types $T \in \mathcal{T}$, $\texttt{Error} \preceq T \preceq \texttt{Data}$. The reflexive and transitive partial order $\preceq$ expresses subtyping and restricts a structural subtype relation which ensures substitutability; If $T \preceq T'$, then any value of type T can masquerade as a value of T'. The subtype relation is derived from the class hierarchy: If a class C extends a class C', then $C \preceq C'$. However, if the methods of C include those of C' (with signatures obeying the subclass relation for function types, defined below), we may safely extend $\preceq$ with $C \preceq C'$ since the internal state of other objects is inaccessible in the language. For product types R and R', $R \preceq R'$ is the point-wise extension of the subtype relation. For the typing and binding of methods, $\preceq$ is extended to function spaces $A \to B$, where A is a (possibly empty) product type: $A \to B \preceq A' \to B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$. Let $T \cap T'$ denote a (largest) common subtype of T and T'. (In fact there may be more than one, in which case any can be selected.) Assume that $\perp \notin \mathcal{T}$ and let $\mathcal{T}_\perp = \mathcal{T} \cup \{\perp\}$. We lift type intersection to $\mathcal{T}_\perp$ by defining $\perp \cap T = T$ for all $T \in \mathcal{T}_\perp$ and lift $\preceq$ to $\mathcal{T}_\perp$ by defining $\perp \not\preceq T$ and $T \not\preceq \perp$ for all $T \in \mathcal{T}$. Conceptually, $\perp$ represents missing type information.

Mappings bind variables and constants to types in $\mathcal{T}_\perp$ and are built from the empty mapping ϵ by means of bindings $[v \mapsto e]$. For a mapping σ and variables v and v', *mapping application* is defined by $\epsilon(v) = \perp$, $\sigma[v \mapsto e](v) = e$, and $\sigma[v \mapsto e](v') = \sigma(v')$ if $v' \neq v$. A mapping σ is *well-defined* if $\sigma(v) \neq \texttt{Error}$ for every v. Let $\sigma \cdot \sigma'$ denote the *concatenation* of mappings σ and σ', defined by $\sigma \cdot \epsilon = \sigma$ and $\sigma \cdot \sigma'[v \mapsto e] = (\sigma \cdot \sigma')[v \mapsto e]$. The domain and range of a mapping are defined by induction; $\mathrm{dom}(\epsilon) = \mathrm{ran}(\epsilon) = \emptyset$, $\mathrm{dom}(\sigma[v \mapsto e]) = \{v\} \cup \mathrm{dom}(\sigma)$, and $\mathrm{ran}(\sigma[v \mapsto e]) = \{e\} \cup \mathrm{ran}(\sigma)$. Let $\sigma \subseteq \sigma'$ if $\sigma(x) = \sigma'(x)$ for all $x \in \mathrm{dom}(\sigma)$. For convenience, we let $[x_1 \mapsto T_1, x_2 \mapsto T_2]$ denote $\epsilon[x_1 \mapsto$

$T_1][x_2 \mapsto T_2]$. Note that a variable may be explicitly bound to $\bot$. Consequently we get $\mathrm{dom}(\epsilon[v \mapsto \bot]) \neq \mathrm{dom}(\epsilon)$, although $\epsilon[v \mapsto \bot](v') = \epsilon(v')$ for all v'. For mappings σ and σ', we define their *union* by $\sigma \cup \sigma'(v) = \sigma(v) \cap \sigma'(v)$; i.e., the mapping union includes bindings which only occur in one mapping and takes a (least) common subtype if a variable is bound in both mappings.

The type analysis uses a type and effect system, given in Figure 4. Let Γ and Δ be well-defined mappings from variable and constant to type names. Judgments $\Gamma, \Delta \vdash \overline{s} \rhd \overline{s}_r, \Delta'$ express that $\overline{s}$ is well-typed in the typing context Γ of program and Δ of label variables, $\overline{s}_r$ is an *expansion* of $\overline{s}$ into runtime code, and Δ' is an *update* of Δ. (To improve readability, $\overline{s}_r$ is omitted if there is no expansion and Δ' is omitted if there is no update.) The analysis is *from right to left*, as apparent from (Seq): $\overline{s}_2$ is analyzed first, and $\overline{s}_1$ is analyzed in the context updated by the effect of the analysis of $\overline{s}_2$. The rule is not symmetric since mapping composition is not commutative. In the type rules, let T, T', and T_0 range over types in $\mathcal{T} \setminus \{\mathtt{Error}\}$.

Rule (Seq) illustrates the backwards analysis approach. In the analysis of $\overline{s}_1$; $\overline{s}_2$, the statements $\overline{s}_2$ are analyzed first, deriving a context update Δ_2. The statements $\overline{s}_1$ are analyzed subsequently and depend on the effect of the analysis of $\overline{s}_2$. Rules (Assign), (New), (Release), and (Skip) are as expected and have no effect. However, assignment to label variables is not allowed. For simplicity, we identify the empty list ϵ of program statements with `skip`. The analysis of `await` statements decomposes guards with ($\wedge$-guards). Here, the effects of the analysis of the two branches are composed into the joint effect of the conjunction. Rule (B-guards) for Boolean guards has no effect. However, for a reply guard $t?$, (t-guards) records an effect which places a type constraint on t. The type constraint may be understood as follows: If $\Delta(t) = \bot$, there is no previous constraint on t and the effect records the weakest constraint possible; i.e., $[t \mapsto \mathtt{Data}]$. However, if there is a previous constraint on t, this constraint is kept; i.e, if $\Delta(t) \neq \bot$, then $\Delta(t) \cap \mathtt{Data} = \Delta(t)$. For the analysis of a statement `await` g in (Await), the type system analyzes the guard g, deriving an effect Δ'. There are two possibilities, depending on whether new constraints have been introduced in the analysis of g. This corresponds to deciding $\mathit{live}(t, \overline{s})$, where $\overline{s}$ is the statement list with effect Δ and $t \in \mathrm{dom}(\Delta')$. Thus, if $t \in \mathrm{dom}(\Delta')$ and $\neg \mathit{live}(t, \overline{s})$ then t will not be read in $\overline{s}$ and can be deallocated after executing `await` g. In the type system, the new constraints are collected in a set t. If $L = \emptyset$, the instruction `free`$(\emptyset)$ is inserted, which is considered the same as `skip`. On the other hand if $L \neq \emptyset$ then there are new constraints on labels in t. In this case, these constraints are propagated as the effect and since we know that $\neg \mathit{live}(t, \overline{s})$ for all $t \in L$, these labels are never needed later, they are deallocated in the runtime code, i.e., by the statement `free`(L).

When arriving at a reply assignment $t?(v)$, it is required that a corresponding asynchronous invocation with label t is encountered later during type checking.

$$(\textsc{Seq})\ \frac{\Gamma,\Delta \vdash \overline{s}_2 \rhd \overline{s}'_2, \Delta_2 \quad \Gamma, \Delta\cdot\Delta_2 \vdash \overline{s}_1 \rhd \overline{s}'_1,\Delta_1}{\Gamma,\Delta\vdash \overline{s}_1;\ \overline{s}_2 \rhd \overline{s}'_1;\ \overline{s}'_2, \Delta_2\cdot\Delta_1} \qquad (\textsc{Skip})\ \frac{}{\Gamma,\Delta\vdash \texttt{skip}\rhd}$$

$$(\textsc{Assign})\ \frac{\Gamma\vdash e:T' \quad \Gamma(v)\neq \mathsf{Label} \quad T'\preceq\Gamma(v)}{\Gamma,\Delta\vdash v := e\rhd} \qquad (\textsc{New})\ \frac{C\preceq\Gamma(v)}{\Gamma,\Delta\vdash v := \texttt{new}\ C()\rhd}$$

$$(\wedge\textsc{-guards})\ \frac{\Gamma,\Delta\vdash g_1\rhd g_1,\Delta_1 \qquad \Gamma,\Delta\vdash g_2\rhd g_2,\Delta_2}{\Gamma,\Delta\vdash g_1\wedge g_2 \rhd\ \ \Delta_1\cdot\Delta_2} \qquad (\textsc{Release})\ \frac{}{\Gamma,\Delta\vdash\texttt{release}\rhd}$$

$$(t\textsc{-guards})\ \frac{\Gamma(t)=\texttt{Label}}{\Gamma,\Delta\vdash t?\rhd\ \ [t\mapsto\Delta(t)\cap\texttt{Data}]} \qquad (\textsc{b-guards})\ \frac{\Gamma\vdash b:\texttt{Bool}}{\Gamma,\Delta\vdash b\rhd}$$

$$(\textsc{Await})\ \frac{\Gamma,\Delta\vdash g\rhd g,\Delta' \quad L=\{t\mid \Delta'(t)=\texttt{Data}\wedge\Delta(t)=\bot\}}{\Gamma,\Delta\vdash \texttt{await}\ g\rhd\texttt{await}\ g;\ \texttt{free}(L),\Delta'}$$

$$(\textsc{Reply})\ \frac{\Gamma(t)=\texttt{Label}\qquad \Delta(t)=\bot \qquad \Gamma(v)=T}{\Gamma,\Delta\vdash t?(v)\rhd[t\mapsto T]}$$

$$(\textsc{Call1})\ \frac{\Gamma(t)=\texttt{Label}\quad \Gamma\vdash\overline{e}:T\quad\Gamma\vdash e:C\quad\Delta(t)\neq\bot\quad \texttt{lookup}(C,m,T\to\Delta(t))}{\Gamma,\Delta\vdash t!e.m(\overline{e})\rhd t!e.m(T\to\Delta(t),\overline{e}),[t\mapsto\bot]}$$

$$(\textsc{Call2})\ \frac{\Gamma(t)=\texttt{Label}\quad \Gamma\vdash\overline{e}:T\quad\Gamma\vdash e:C\quad\Delta(t)=\bot\quad \texttt{lookup}(C,m,T\to\texttt{Data})}{\Gamma,\Delta\vdash t!e.m(\overline{e})\rhd t!e.m(T\to\texttt{Data},\overline{e});\ \texttt{free}(t),[t\mapsto\bot]}$$

$$(\textsc{Choice})\ \frac{\begin{array}{c}\Gamma,\Delta\vdash\overline{s}_1\rhd\overline{s}'_1,\Delta_1 \qquad \Gamma,\Delta\vdash\overline{s}_2\rhd\overline{s}'_2,\Delta_2 \qquad \textit{well-defined}((\Delta\cdot\Delta_1)\cup(\Delta\cdot\Delta_2))\\ L_1=\{t\mid\Delta_1(t)\preceq\texttt{Data}\wedge\Delta(t)=\bot\} \qquad L_2=\{t\mid\Delta_2(t)\preceq\texttt{Data}\wedge\Delta(t)=\bot\}\end{array}}{\Gamma,\Delta\vdash\overline{s}_1\,\Box\,\overline{s}_2\rhd\texttt{free}(L_2\setminus L_1);\ \overline{s}'_1\,\Box\,\texttt{free}(L_1\setminus L_2);\ \overline{s}'_2,(\Delta\cdot\Delta_1)\cup(\Delta\cdot\Delta_2)}$$

$$(\textsc{Method})\ \frac{\begin{array}{c}\Gamma'=\Gamma[\overline{x\mapsto T_0}][\overline{x'\mapsto T'}] \qquad T_0\neq\mathsf{Label}\\ \Gamma'\vdash e:T \qquad \Gamma',\epsilon\vdash\overline{s}\rhd\overline{s}',\Delta \qquad \mathrm{ran}(\Delta)=\{\bot\}\end{array}}{\begin{array}{l}\Gamma,\epsilon\vdash T\ m\ (\overline{T_0\ x})\{\overline{T'\ x'};\ \overline{s};\ \texttt{return}\ e\}\\ \qquad\rhd T\ m\ (\overline{T_0\ x})\langle\overline{s}';\ \texttt{return}\ e,\overline{x'\mapsto\texttt{default}(T')}\rangle\end{array}}$$

$$(\textsc{Class})\ \frac{\texttt{for all}\ M\in\overline{M}\ \cdot\ \Gamma[\texttt{fields}(C)],\epsilon\vdash M\rhd M'}{\begin{array}{l}\Gamma,\epsilon\vdash\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{T\ f};\ \overline{M}\}\\ \qquad\rhd\texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{f\mapsto\texttt{default}(T)},\overline{M}'\}\end{array}}$$

$$(\textsc{Program})\ \frac{\texttt{for all}\ L\in\overline{L}\ \cdot\ \epsilon,\epsilon\vdash L\rhd L' \qquad [\overline{x\mapsto T}],\epsilon\vdash\overline{s}\rhd\overline{s}'}{\Gamma_0,\epsilon\vdash\overline{L}\ \{\overline{T\ x};\ \overline{s}\}\rhd\overline{L}'\ \langle\overline{s}',\overline{x\mapsto\texttt{default}(T)}\rangle}$$

Figure 4. The type and effect system, where static type information is captured by an initial mapping $\Gamma_0 = [\texttt{true}\mapsto\texttt{Bool},\texttt{false}\mapsto\texttt{Bool},\texttt{default}(T)\mapsto T]$ for all T.

Furthermore, the type of v is used to infer a correct output type for the method call on t. For this purpose, (Reply) imposes a constraint $[t\mapsto\Gamma(v)]$ as an effect of the type analysis. The reply assignment is destructive (see rule R11 in the operational semantics of Section 4). Therefore a reply assignment is only allowed when $\neg\textit{live}(t,\overline{s})$ where $\overline{s}$ is the statement list that has already been type checked. Technically, we here require $\Delta(t) = \perp$ in the backwards analysis.

For (Call1) and (Call2) the actual signature of an asynchronous invocation $t!e.m(\overline{e})$ can now be directly derived, as $\Delta(t)$ provides the constraint associated with the out-parameter type. If $\Delta(t) = \perp$, there is no reply assignment or guard corresponding to the invocation, so $\neg \mathit{live}(t, \overline{s})$ where $\overline{s}$ gives effect Δ, and (Call2) applies. In this case any out-parameter type is acceptable, which is checked by $\mathtt{lookup}(C, m, T \rightarrow \mathtt{Data})$, and the invocation in the runtime code gets this signature. Furthermore, since there is no reply assignment or guard for this invocation, deallocation is immediately introduced after the call. In contrast if $\Delta(t) \neq \perp$, there are reply assignments or guards corresponding to the invocation, and (Call1) applies. In this case, the out-parameter type is given by the constraint $\Delta(t)$ and checked by $\mathtt{lookup}(C, m, T \rightarrow \Delta(t))$. Finally, the runtime invocation gets this derived signature and the label t is reset by updating Δ with the effect $[t \mapsto \perp]$. In this case, deallocation is not needed.

In a nondeterministic choice $\overline{s}_1 \Box \overline{s}_2$, only one branch is evaluated at runtime. Hence, in (Choice) each branch is type checked in the same context. If there is a reply assignment with label t in each of the branches that corresponds to the same invocation, which is an invocation textually occurring to the left of the choice statement, then Δ is updated by mapping t to a largest common subtype of the two return types, i.e., with $[t \mapsto (\Delta_1(t) \cap \Delta_2(t))]$. Note that this type intersection may give $[t \mapsto \mathtt{Error}]$, in which case the mapping $(\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)$ is not well-defined and the rule fails. If the operation succeeds, represented by the predicate *well-defined*, the invocation has a return type that is well-typed for both executions by subtyping. Moreover, the type system ensures that a reply assignment with label t can occur after a nondeterministic choice only if a call with label t is pending after the evaluation of either branch. Deallocation applies in one branch if there are reply assignments or guards only in the other branch. There may be many such labels in each branch, identified by the sets L_1 and L_2. Thus $L_1 \setminus L_2$ and $L_2 \setminus L_1$ contain exactly the labels which should be deallocated in branches $\overline{s}_2$ and $\overline{s}_1$, respectively.

Note that if a call is pending on a label t before a choice statement (say $\Delta(t) = T$) and this call is overwritten in one branch (e.g., $\overline{s}_1$ is $t?(\overline{v})$; $t!o.m(\overline{e})$ so that $\Delta_1(t) = \Gamma(\overline{v})$) but the label is untouched in the other branch (i.e., $t \notin \mathrm{dom}(\Delta_2)$), then updating Δ to $\Delta_1 \cup \Delta_2$ would result in $(\Delta \cdot (\Delta_1 \cup \Delta_2))(t) = \Delta_1 \cup \Delta_2(t) = \Gamma(\overline{v})$. This would be incorrect; if the branch $\overline{s}_2$ were chosen, the reply should still be assigned to $\Delta(t) = T$. In order to avoid losing this information, (Choice) instead updates Δ to $(\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)$. This has the advantage that if $t \notin \mathrm{dom}(\Delta_2)$ then $\Delta \cdot \Delta_2(t) = \Delta(t)$, which solves the problem of one branch not touching the label. Remark that it may be the case that $t \in \mathrm{dom}(\Delta_2)$ such that $\Delta_2(t) = \perp$, for example if $\overline{s}_2$ is $t!o.m(\overline{e})$. In this case the branch has assigned $\perp$ to t by (Call1) and $\Delta \cdot \Delta_2(t) = \Delta[t \mapsto \perp](t) = \perp$. Thus if $t \in \mathrm{dom}(\Delta_1) \cup \mathrm{dom}(\Delta_2)$, then $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) = (\Delta \cdot (\Delta_1 \cup \Delta_2))(t)$.

In $(\textsc{Class})$, the function $\texttt{fields}(C)$ provides the typing context for the fields of a class C and its superclass, including $\texttt{this}$ of type C. Similarly in $(\textsc{Method})$ the type information for input variables and local variables extend the typing context. Note that the analysis in $(\textsc{Method})$, $(\textsc{Class})$, and $(\textsc{Program})$ starts with empty label mappings and the update after analysis in $(\textsc{Method})$ only contains $\perp$ constraints. Furthermore, labels may not be passed as method parameters. These requirements enforce the encapsulation of method replies within the scope of a method body; a reply assignment or reply guard must be preceded by a method invocation on the same label within the method body. Otherwise a deadlock would be possible when executing the method body, as the processor would block permanently waiting for a reply to a non-existing or previously consumed method invocation. Method calls in the runtime code include signatures derived by the type analysis, which will be used in the runtime method lookup.

Example 3 Now reconsider the wide-area network services of Example 2, which provide news at request. We derive the runtime code generated by the type analysis for *newsRelay*, *newsRelay2*, and *newsRelay3*, assuming that variables CNN and BBC have type $\texttt{News}$ in the class where the methods are declared. Hence, $(\textsc{Class})$ builds a typing context $\Gamma = [\mathit{CNN} \mapsto \texttt{News}, \mathit{BBC} \mapsto \texttt{News}, \mathit{email} \mapsto \texttt{Email}]$ in which the methods are type checked. For the method

$$\begin{array}{l}\texttt{Bool}\ \mathit{newsRelay}(\texttt{Date}\ d, \texttt{Client}\ a)\{\\ \quad \texttt{XML}\ v, \mathsf{Label}\ l;\ \ \mathit{CNN} := \texttt{new News}();\ \ t!\mathit{CNN.news}(d);\ \ \texttt{await}\ t?;\ \ t?(v);\\ \quad \mathit{email} := \texttt{new Email}();\ \ t!\mathit{email.send}(v,a);\ \ \texttt{return true}\}\end{array}$$

we get the typing environment $\Gamma' = \Gamma \cdot [d \mapsto \texttt{Date}, a \mapsto \texttt{Client}, v \mapsto \texttt{XML}, t \mapsto \mathsf{Label}]$ and the judgment

$$\begin{array}{l}\Gamma', \epsilon \vdash \mathit{CNN} := \texttt{new News}();\ \ t!\mathit{CNN.news}(d);\ \ \texttt{await}\ t?;\ \ t?(v);\\ \qquad\qquad \mathit{email} := \texttt{new Email}();\ \ t!\mathit{email.send}(v,a) \rhd \overline{s}_r\end{array}$$

where $\overline{s}_r$ is the derived runtime code obtained by the analysis of the method body. We start with the analysis of $t!\mathit{email.send}(v, a)$, which is type checked by $(\textsc{Call2})$ since $\epsilon(t) =\perp$. Since $\neg \mathit{live}(t, \epsilon)$, t will not be read by any reply or guard statements later and the insertion of $\texttt{free}(t)$ is safe. Assuming that *lookup* succeeds for the given types of t, v, and a, we get the following judgment:

$$\Gamma', \epsilon \vdash t!\mathit{email.send}(v,a) \rhd t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \ \texttt{free}(t)$$

We proceed with the statement $\mathit{email} := \texttt{new Email}()$ and get the judgment

$$\Gamma', \epsilon \vdash \mathit{email} := \texttt{new Email}() \rhd \mathit{email} := \texttt{new Email}()$$

Continue with the statement $t?(v)$, for which we get the judgment

$$\Gamma', \epsilon \vdash t?(v) \rhd t?(v), [t \mapsto \texttt{XML}]$$

by applying (Reply). For $t?(v)$, we have $\Gamma', [t \mapsto \texttt{XML}] \vdash t? \rhd t?, [t \mapsto \texttt{XML}]$ by (t-Guards), since $\texttt{XML} \cap \texttt{Data} = \texttt{XML}$ and no new constraints on label t have been introduced. Consequently, we can apply (Await) and obtain the judgment

$$\Gamma', [t \mapsto \texttt{XML}] \vdash \texttt{await}\ t? \rhd \texttt{await}\ t?$$

Applying (Call1) to $t!\mathit{CNN.news}(d)$ since $[t \mapsto \texttt{XML}](t) \neq \perp$, we get the judgment

$$\Gamma', [t \mapsto \texttt{XML}] \vdash t!\mathit{CNN.news}(d) \rhd t!\mathit{CNN.news}(\texttt{Date} \rightarrow \texttt{XML}, d), [t = \perp]$$

Finally, we apply rule (New) to $\mathit{CNN} := \texttt{new News}()$ and obtain the judgment

$$\Gamma', \epsilon \vdash \mathit{CNN} := \texttt{new News}() \rhd \mathit{CNN} := \texttt{new News}()$$

Since $\mathrm{ran}([t = \perp]) = \perp$ the type analysis is correct by rule (Method) for the reassembled runtime code $\overline{s}_r$, which becomes

```
  $\mathit{CNN} := \texttt{new News}();\ t!\mathit{CNN.news}(\texttt{Date} \rightarrow \texttt{XML}, d);\ \texttt{await}\ t?;\ t?(v);$
    $\mathit{email} := \texttt{new Email}();\ t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a));\ \texttt{free}(t)$
```

If we let σ denote the state $v \mapsto \texttt{default}(\texttt{XML}), t \mapsto \texttt{default}(\mathsf{Label}), t' \mapsto \texttt{default}(\mathsf{Label})$, we similarly obtain for *newsRelay2* and *newsRelay3* the following runtime method representations:

```
Bool $\mathit{newsRelay2}(\texttt{Date}\ d, \texttt{Client}\ a)\langle$
  $t!\mathit{CNN.news}(\texttt{Date} \rightarrow \texttt{XML}, d);\ t'!\mathit{BBC.news}(\texttt{Date} \rightarrow \texttt{XML}, d);$
  $(\texttt{await}\ t?;\ t?(v);\ t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t);$
  $\texttt{await}\ t'?;\ t'?(v);\ t'!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t'))$
   $\Box\ (\texttt{await}\ t'?;\ t'?(v);\ t'!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t');$
  $\texttt{await}\ t?;\ t?(v);\ t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t));$
  $\texttt{return true}, \sigma\rangle$
Bool $\mathit{newsRelay3}(\texttt{Date}\ d, \texttt{Client}\ a)\langle$
  $t!\mathit{CNN.news}(\texttt{Date} \rightarrow \texttt{XML}, d);\ t'!\mathit{BBC.news}(\texttt{Date} \rightarrow \texttt{XML}, d);$
     $((\texttt{free}(t');\ \texttt{await}\ t?;\ t?(v))\ \Box\ \ (\texttt{free}(t);\ \texttt{await}\ t'?;\ t'?(v)));$
       $t!\mathit{email.send}(\texttt{XML} \times \texttt{Client} \rightarrow \texttt{Data}, v, a);\ \texttt{free}(t));\ \texttt{return true}, \sigma\rangle$
```

The type system in Figure 4 is basically syntax-directed; i.e., only one rule applies to any syntactic construct. Consequently the implementation of the type system is straightforward: each rule corresponds to one case in the algorithm. There is one exception: the rules (Call1) and (Call2) for method invocation. Here, rule selection is determined by the additional condition ($\Gamma(t) = \perp$). The exception is due to the insertion of instructions for message deallocation and reflects the low cost of inserting these instructions during the type analysis.

3.1 Properties of the Type Based Translation

We consider some properties of the type based translation (for proofs, see Appendix A). Theorem 1 relates live labels to the type system's effect mapping.

Theorem 1 *If* $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta'$ *then* $\forall t\colon \mathsf{Label} \cdot \neg \mathit{live}(t, \overline{s}) \iff \Delta'(t) = \bot$.

The following corollaries follow from Theorem 1, since $\Delta(t) = \bot$ is a premise of the rules (Reply), (Await), and (Call2).

Corollary 1 *If* $\Gamma, \epsilon \vdash t?(v);\ \overline{s}_0 \rhd t?(v);\ \overline{s}'_0, \Delta'$ then $\neg \mathit{live}(t, \overline{s}_0)$.

Corollary 2 *If* $\Gamma, \epsilon \vdash s;\ \overline{s}_0 \rhd s';\ \mathtt{free}(t);\ \overline{s}'_0, \Delta'$ *then* $\neg \mathit{live}(t, \overline{s}_0)$.

It follows that the type based translation does not introduce too many deallocation statements. Note that only reply assignments and reply guards on a label t make $\Delta(t)$ different from $\bot$, that only (Call1) actually resets $\Delta(t)$ to $\bot$, and that (Method) requires that $\mathrm{ran}(\Delta) = \bot$. Therefore, it follows from Theorem 1 that there are enough method invocations in well-typed programs. Deallocation operations are only inserted by rules (Await) and (Call2), corresponding to the cases without a (destructive) reply assignment on a label t before a reply guard or invocation on t; i.e., the rules capture the cases when there is another operation on t than a destructive read and t is not live. Consequently, by Theorem 1, there are enough deallocation operations in the runtime code. Thus, the type based translation has the following properties:

(1) A deallocation statement for a label t is not introduced when t is live.
(2) Every reply assignment and reply guard on a label t in a well-typed program is textually preceded by a corresponding invocation on t.
(3) Method returns are deallocated in all program branches where they are not destructively read.

Furthermore, the signatures which expand the method invocations in the runtime code ensure that the assignment of reply values is well-typed in every execution path. This ensures that late binding preserves the well-typedness of the local object state of the caller:

Theorem 2 (Well-typedness of return value) *Given* $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta$. *If* $t!o.m(\overline{e})$ *is an invocation in* $\overline{s}$ *which gets translated into* $t!o.m(T \rightarrow T', \overline{e})$ *and* $t?(v)$ *is reply assignment corresponding to that invocation, then* $T' \preceq \Gamma(v)$.

The requirements to the type system which were stated at the beginning of Section 3 are now reconsidered. The type safety of reply guards is given by Property 2 above. The type safety of reply assignments is given by Theorem 2 and Property 2. Property 1 ensures that the deallocation of a label t only

$$\begin{aligned}
\mathit{config} &::= \epsilon \mid \mathit{object} \mid \mathit{msg}\ \mathbf{to}\ o \mid \mathit{config}\ \mathit{config}\\
\mathit{object} &::= \langle \mathit{oid} : C \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr}\colon \mathit{active}, \mathit{PrQ}\colon \mathit{processQ}, \mathit{EvQ}_{(\overline{\mathit{mid}},\overline{\mathit{mid}})} : \mathit{eventQ}\rangle\\
\mathit{msg} &::= \mathit{invoc}(m, T \to T, \overline{\mathit{val}}, \langle o, \mathit{mid}\rangle) \mid \mathit{comp}(\mathit{mid}, T, \mathit{val})\\
\mathit{active} &::= \mathit{process} \mid \texttt{idle}\\
\mathit{process} &::= \langle s_r, \mathit{sub}\rangle \mid \texttt{lookup-error} \mid \texttt{deallocation-error}\\
\mathit{processQ} &::= \epsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ}\\
\mathit{eventQ} &::= \epsilon \mid \mathit{msg} \mid \mathit{eventQ}\ \mathit{eventQ}
\end{aligned}$$

Figure 5. Runtime configurations.

occurs when t is not live and Property 3 that deallocation statements are inserted for all invocations that are not destructively read.

4 Operational Semantics

The operational semantics of the language is given in rewriting logic [24] and is executable on the Maude machine [7]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. Rewrite rules apply to terms of given types. Types are specified in (membership) equational logic (Σ, E). Different system components will be modeled by terms of the different types defined in the equational logic. The global state configuration is defined as a multiset of these terms.

Rewriting logic extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules supplementing the equations which define the term language. Let $\to$ denote the reflexive and transitive closure of the rewrite relation. A rewrite rule $p \to p'$ may be interpreted as a *local transition rule* allowing an instance of the pattern p to evolve into the corresponding instance of the pattern p'. Conditional rewrite rules $p \to p'$ **if** *cond* are allowed, where the condition *cond* can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the rewrite transitions. Rewrite rules apply to local fragments of a state configuration. If rewrite rules may be applied to nonoverlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic.

The runtime configurations of Creol's operational semantics, given in Figure 5, are multisets combining objects and messages. An object identifier consists of a value *oid* and a class name. When the class of an object is of no significance, the object identifier is denoted o. With a slight abuse of notation we denote by $\overline{c}$ in the presentation of the operational semantics either a list, set, or multiset of a construct c, depending on the context. An asynchronous method call in the language is reflected by a pair of messages. An invocation message $\mathit{invoc}(m, T \rightarrow T', \overline{e}, \langle o, \mathit{mid}\rangle)$ addressed to a callee o' represents a call to method m of object o' with actual signature $T \rightarrow T'$, and actual in-parameter values $\overline{e}$, and the value mid (of type `Label`) is a locally unique label value identifying the call for the caller o. In a corresponding completion message $\mathit{comp}(\mathit{mid}, T, e)$ addressed to a caller o, the label value mid identifies the call, and e is the return value with type T. Thus, the label value associated with a method call identifies the invocation and completion messages for that call.

In order to improve readability in the presentation of the rules the different elements of runtime objects are tagged and elements that are irrelevant for a particular rule are omitted, as usual in rewriting logic. Thus in an object $\langle \mathit{oid} : C \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr}\colon \mathit{active}, \mathit{PrQ}\colon \mathit{processQ}, \mathit{EvQ}_{(\overline{t},\overline{g})}\colon \mathit{eventQ}\rangle$, oid is an object identifier tagged by a class name C, sub are the fields, active is the active process, $\mathit{processQ}$ is the process queue, and eventQ is the event queue. (When the class of an object is of no significance, the object identifier $\mathit{oid} : C$ is simply denoted o.) A process consists of a state for local variables and a list of program statements. (Any process derived from a well-typed method definition has `return` e as its final statement.) Note that each field and local process variable is initialized with a type-correct default value. The active process may be `idle`. A special process `lookup-error` is introduced in the program queue to represent failed method lookup. The event queue eventQ consists of incoming unprocessed messages to the object and has associated sets $\overline{\mathit{mid}}$ representing completion messages to be deallocated and completion messages that have been deallocated. A special process `deallocation-error` is introduced to represent the dereferencing of a deallocated completion message. For process and event queues, we let $\emptyset$ be the empty queue and whitespace the associative and commutative constructor (following rewriting logic conventions [7]).

For simplicity, the operational semantics given in Figures 6 and 7 abstracts from the representation of classes. Consequently the (local) uniqueness of a name n is represented by a predicate $\mathit{fresh}(n)$, which provides (global) uniqueness in combination with either a class or object identifier. We assume given two auxiliary functions which depend on the representation of classes. The function $\mathit{fields}(C)$ returns an initial object state in which `this` and the declared fields of class C and its superclasses are bound to default values. The function lookup returns a process, given a class, a method name, the signature of the actual in- and out-parameters to the call, and actual in-parameters. This process has a state in which the local variables have been given default values. The reserved local variables γ, α, and β are instantiated and store values for the method activation's return label, caller, and return type, respectively. Note that we assume that `this` as well as γ, α, and β are read-only variables. Consequently, these local variables ensure that each activation's return value

$$\begin{array}{ll}
 & \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle v := e;\ \overline{s}, \overline{l} \rangle\ \rangle \\
\text{(R0)} & \longrightarrow\ \textbf{if}\ v\ \mathit{in}\ \overline{l}\ \textbf{then}\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, (\overline{l}[v \mapsto [\![e]\!]_{(\overline{f} \cdot \overline{l})}]) \rangle\ \rangle \\
 & \qquad\qquad\qquad\quad \textbf{else}\ \ \langle o \mid \mathit{Fld} : (\overline{f}[v \mapsto [\![e]\!]_{(\overline{f} \cdot \overline{l})}]), \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle\ \rangle\ \textbf{fi}
\end{array}$$

$$\begin{array}{ll}
 & \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (v := \texttt{new}\ C();\ \overline{s}), \overline{l} \rangle\ \rangle \\
\text{(R1)} & \longrightarrow\ \langle o \mid \mathit{Fld}\colon \overline{f}, \mathit{Pr}\colon \langle (v := (n : C);\ \overline{s}), \overline{l} \rangle\ \rangle \\
 & \langle (n : C) \mid \mathit{Fld} : \mathit{fields}(C), \mathit{Pr}\colon \langle \texttt{this} := (n : C), \epsilon \rangle, \mathit{PrQ}\colon \epsilon, \mathit{EvQ}_{(\emptyset,\emptyset)} : \epsilon \rangle\ \textbf{if}\ \mathit{fresh}(n)
\end{array}$$

$$\begin{array}{ll}
\text{(R2)} & \begin{array}{l} \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \texttt{await}\ g;\ \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{enabled}(g, (\overline{f} \cdot \overline{l}), \overline{q}) \end{array}
\end{array}$$

$$\begin{array}{ll}
\text{(R3)} & \begin{array}{l} \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \texttt{await}\ g;\ \overline{s}, \overline{l} \rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \texttt{deallocation-error}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{dealloc}(g, (\overline{f} \cdot \overline{l}), \mathcal{G}) \end{array}
\end{array}$$

$$\begin{array}{ll}
 & \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\text{(R4)} & \longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \texttt{release;}\ \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
 & \textbf{if}\ \neg \mathit{enabled}(\overline{s}, (\overline{f} \cdot \overline{l}), \overline{q})\ \wedge\ \neg \mathit{dealloc}(\overline{s}, (\overline{f} \cdot \overline{l}), \mathcal{G})
\end{array}$$

$$\begin{array}{ll}
\text{(R5)} & \begin{array}{l} \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \texttt{release;}\ \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \texttt{idle}, \mathit{PrQ} : (\overline{w}\ \langle \overline{s}, \overline{l} \rangle), \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \end{array}
\end{array}$$

$$\begin{array}{ll}
\text{(R6)} & \begin{array}{l} \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \texttt{idle}, \mathit{PrQ} : \langle \overline{s}, \overline{l} \rangle\ \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle \\
\longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}, \overline{l} \rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q} \rangle\ \textbf{if}\ \mathit{ready}(\overline{s}, (\overline{f} \cdot \overline{l}), \overline{q}) \end{array}
\end{array}$$

$$\begin{array}{ll}
 & \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (t!r(\mathit{Sig}, \overline{e});\ \overline{s}), \overline{l} \rangle \rangle \\
\text{(R7)} & \longrightarrow\ \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (t := n;\ \overline{s}), \overline{l} \rangle \rangle\ \mathit{invoc}(m, \mathit{Sig}, ([\![\overline{e}]\!]_{(\overline{f} \cdot \overline{l})}), \langle o, n \rangle)\ \textbf{to}\ [\![x]\!]_{(\overline{f} \cdot \overline{l})} \\
 & \textbf{if}\ \mathit{fresh}(n)
\end{array}$$

Figure 6. The operational semantics (1). Any process $\langle \epsilon, \overline{l} \rangle$ is reduced to `idle`.

may be correctly returned to its caller when process execution is interleaved. If method binding fails, the `lookup-error` process is returned.

If σ is a state and e an expression, we denote by $[\![e]\!]_\sigma$ the reduction of e in σ. We assume that the functional language of (side effect free) expressions is type sound, so well-typed expressions remain well-typed during evaluation. To simplify the presentation we omit the details of this standard reduction [40]. The enabledness of a guard in a given state σ with a given event queue $\overline{q}$ may be defined by induction over the construction of guards, as follows:

$$\begin{array}{ll}
\mathit{enabled}(t?, \sigma, \overline{q}) & = [\![t]\!]_\sigma\ \mathit{in}\ \overline{q} \\
\mathit{enabled}(b, \sigma, \overline{q}) & = [\![b]\!]_\sigma \\
\mathit{enabled}(g \wedge g', \sigma, \overline{q}) & = \mathit{enabled}(g, \sigma, \overline{q}) \wedge \mathit{enabled}(g', \sigma, \overline{q})
\end{array}$$

Here, in checks whether a completion message corresponding to the given label value $[\![t]\!]_\sigma$ is in a message queue $\overline{q}$. The predicate $\mathit{dealloc}$ determines if a guard dereferences a deallocated message in the set of label values S:

$$\begin{array}{ll}
\mathit{dealloc}(t?, \sigma, S) & = [\![t]\!]_\sigma \in S \\
\mathit{dealloc}(b, \sigma, S) & = \texttt{false} \\
\mathit{dealloc}(g \wedge g', \sigma, S) & = \mathit{dealloc}(g, \sigma, S) \vee \mathit{dealloc}(g', \sigma, S)
\end{array}$$

(R8) $(\mathit{msg}\ \mathbf{to}\ o)\ \langle o \mid \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle \ \longrightarrow\ \langle o \mid \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{msg}\rangle$

(R9) $$\begin{array}{l}\langle \mathit{oid} : C \mid \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{invoc}(m, \mathit{Sig}, \overline{e}, \langle o, \mathit{mid}\rangle)\rangle \\ \quad\longrightarrow\ \langle \mathit{oid} : C \mid \mathit{PrQ} : (\overline{w}\ \mathit{lookup}(C, m, \mathit{Sig}, (\overline{e}, o, \mathit{mid}))), \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\end{array}$$

(R10) $$\begin{array}{l}\langle o \mid \mathit{Pr} : \langle (t?(v);\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\ \mathit{comp}(\mathit{mid}, T, e)\rangle \\ \quad\longrightarrow\ \langle o \mid \mathit{Pr} : \langle (v := e;\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \mathbf{if}\ \mathit{mid} = \llbracket t\rrbracket_{\overline{l}}\end{array}$$

(R11) $$\begin{array}{l}\langle o \mid \mathit{Pr} : \langle (t?(v);\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle \\ \quad\longrightarrow\ \langle o \mid \mathit{Pr} : \texttt{deallocation-error}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \mathbf{if}\ \llbracket t\rrbracket_{\overline{l}} \in \mathcal{G}\end{array}$$

(R12) $$\begin{array}{l}\langle o | \mathit{Pr} : \langle \texttt{return}\ v;\ \overline{s}, \overline{l}\rangle, \mathit{Fld} : \overline{f}\rangle \\ \quad\longrightarrow\ \langle o \mid \mathit{Pr} : \langle \overline{s}, \overline{l}\rangle, \mathit{Fld} : \overline{f}\rangle\ \mathit{comp}(\llbracket \gamma\rrbracket_{\overline{l}}, \llbracket \beta\rrbracket_{\overline{l}}, \llbracket v\rrbracket_{\overline{f}\cdot\overline{l}})\ \mathbf{to}\ \llbracket \alpha\rrbracket_{\overline{l}}\end{array}$$

(R13) $$\begin{array}{l}\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (\overline{s}_1 \,\Box\, \overline{s}_2);\ \overline{s}_3, \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle \\ \quad\longrightarrow \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle \overline{s}_1;\ \overline{s}_3, \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \mathbf{if}\ \mathit{ready}(\overline{s}_1, (\overline{f}\cdot\overline{l}), \overline{q})\end{array}$$

(R14) $$\begin{array}{l}\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle (\texttt{free}(\overline{t});\ \overline{s}), \overline{l}\rangle, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle \\ \quad\longrightarrow \langle o \mid \mathit{Pr} : \langle \overline{s}, \overline{l}\rangle, \mathit{EvQ}_{(\{\llbracket \overline{t}\rrbracket_{\overline{f}\cdot\overline{l}}\}\cup\mathcal{L},\mathcal{G})} : \overline{q}\rangle\end{array}$$

(R15) $\langle o \mid \mathit{EvQ}_{(\{\mathit{mid}\}\cup\mathcal{L},\mathcal{G})} : \mathit{comp}(\mathit{mid}, T, e)\ \overline{q}\rangle \ \longrightarrow \langle o \mid \mathit{EvQ}_{(\mathcal{L},\mathcal{G}\cup\{\mathit{mid}\})} : \overline{q}\rangle$

Figure 7. The operational semantics (2).

Enabledness is lifted to statement lists to express whether a process may be suspended or not. All atomic statements except `await` statements are enabled. However, in order to select a branch of a process, we may want to only choose a branch which is ready to execute without immediately blocking. This is expressed by the *ready* predicate. The two predicates are defined by induction over the construction of statement lists:

$$\begin{array}{l}\mathit{enabled}(s;\ \overline{s}, \sigma, \overline{q}) = \mathit{enabled}(s, \sigma, \overline{q}) \\ \mathit{enabled}(\mathit{await}\ g, \sigma, \overline{q}) = \mathit{enabled}(g, \sigma, \overline{q}) \\ \mathit{enabled}(\overline{s}\Box\overline{s}', \sigma, \overline{q}) = \mathit{enabled}(\overline{s}, \sigma, \overline{q}) \vee \mathit{enabled}(\overline{s}', \sigma, \overline{q}) \\ \mathit{enabled}(s, \sigma, \overline{q}) = \texttt{true} \quad \mathit{otherwise}\end{array}$$

$$\begin{array}{l}\mathit{ready}(\texttt{free}(\overline{l});\ \overline{s}, \sigma, \overline{q}) = \mathit{ready}(\overline{s}, \sigma, \overline{q}) \\ \mathit{ready}(s;\ \overline{s}, \sigma, \overline{q}) = \mathit{ready}(s, \sigma, \overline{q}) \\ \mathit{ready}(t?(v), \sigma, \overline{q}) = \mathit{enabled}(\mathit{await}\ t?, \sigma, \overline{q}) \\ \mathit{ready}(\overline{s}\Box\overline{s}', \sigma, \overline{q}) = \mathit{ready}(\overline{s}, \sigma, \overline{q}) \vee \mathit{ready}(\overline{s}', \sigma, \overline{q}) \\ \mathit{ready}(s, \sigma, \overline{q}) = \mathit{enabled}(s, \sigma, \overline{q}) \quad \mathit{otherwise}\end{array}$$

Object activity is organized around a message queue *eventQ* containing a multiset of unprocessed incoming messages and a process queue *processQ* for suspended processes; i.e., remaining parts of method activations. There is exactly one active process. In the assignment rule R1, an expression is evaluated and bound to a program variable. In the object creation rule R2, an object with an unique identifier is constructed. The first execution step of the new object is to reassign to `this` its own identifier. Rules R3 and R5 for guards use the auxiliary *enabledness* function. If a guard evaluates to `true`, the execution of the process may continue in rule R3, otherwise a `release` is introduced in rule R5, given that the guard does not contain any read operation on a deallocated label. In rule R6 the active process is suspended on the process queue. If a guard contains read operations on a deallocated label value, a `deallocation-error` process is returned in rule R4. If the active process is `idle`, any suspended process may be activated in rule R7 if it is ready.

A message is emitted into the configuration by a method call in rule R8. The label is translated into a label assignment where the value uniquely identifies the call. The message is delivered to the callee in rule R9 where *msg* ranges over invocations and completions. Message overtaking is captured by the non-determinism inherent in rewriting logic: messages sent by an object in one order may arrive in any order. In rule R10, the method invocation is bound to a process by the *lookup* function. The reply assignment in rule R11 fetches the return value from the completion message in the object's queue. As the message queue is a multiset, the rule will match any occurrence of $comp(mid, T, e)$. If the completion message has already been deallocated, the `deallocation-error` process is returned in rule R12.

The return of a method call is captured by rule R13 where a uniquely labeled completion message is returned to the caller. This rule makes use of the special local variables γ, α, and β which store information from the associated invocation message. Rule R14 deals with internal branching. The choice operator is associative and commutative, so R14 covers the selection of any branch in a compound nondeterministic choice statement. Here, the *ready* predicate provides a lazy branch selection. Rules R15 and R16 deal with the deallocation of messages from the message queue. The statement $\mathtt{free}(\overline{t})$ expresses that messages with labels in $\overline{t}$ may be deallocated. In R15, these are added to the event queue's label list. In R16, a message with a label value in that list is deallocated. The deallocated label values are stored in the set $\mathcal{G}$.

Example 4 We consider an execution sequence of the wide-area network services of Example 2. In Example 3 we showed the runtime code generated by the type analysis for *newsRelay*. Let an object *o* create an instance of class `NewsService` and make an invocation to the new object $(1 : \mathtt{NewsService})$. Figure 8 shows the execution sequence of the method body *newsRelay*.

5 Typing of Runtime Configurations

In this section, we assume that runtime code is derived by type analysis from well-typed source programs. Consequently, a much simpler type system can be used for type analysis of runtime configurations than for the static analysis. In particular, we can rely on Theorems 1 and 2 for the correctness of the return types and of the placement of deallocation operations.

$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto \mathit{null}, \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(\mathit{CNN}:=\texttt{new News}();$
$t!\mathit{CNN.news}(\texttt{Date}\rightarrow\texttt{XML}, d); \texttt{await } t?; t?(v); \mathit{email}:=\texttt{new Email}(); t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a);$
$\texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31), a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)},$
$t\mapsto\texttt{default(Label)}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R2, $\longrightarrow$ R1
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(t!\mathit{CNN.news}(\texttt{Date}\rightarrow\texttt{XML}, d);$
$\texttt{await } t?; t?(v); \mathit{email}:=\texttt{new Email}(); t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}),$
$(d\mapsto(2007-12-31), a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)}, t\mapsto\texttt{default(Label)},$
$\gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\epsilon, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R8
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(t:=2; \texttt{await } t?; t?(v); \mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)}, t\mapsto\texttt{default(Label)}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\epsilon, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\mathit{invoc}(\mathit{news}, \texttt{Date}\rightarrow\texttt{XML}, (2007-12-31), \langle(1:\texttt{NewsService}), 2\rangle)$ **to** $(1:\texttt{News})$
$\longrightarrow$ R1, $\longrightarrow$ R9, $\longrightarrow$ R10, $\longrightarrow$ R7
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(\texttt{await } t?; t?(v); \mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)}, t\mapsto 2, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})],$
$\mathit{Pr}:\langle\overline{s}; \texttt{return}(\mathit{xml}), (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ we omit the execution of $\overline{s}$, R13
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(\texttt{await } t?; t?(v); \mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)}, t\mapsto 2, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\mathit{comp}(2, \texttt{XML}, [\![\mathit{xml}]\!]_{\overline{l}})$ **to** $(1:\texttt{NewsService})$
$\longrightarrow$ R9, $\longrightarrow$ R3 and let $(...)$ be the value of $[\![\mathit{xml}]\!]_{\overline{l}}$
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(t?(v); \mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31), a\mapsto \mathit{john.doe@email.com},$
$v\mapsto\texttt{default(XML)}, t\mapsto 2, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\mathit{comp}(2, \texttt{XML}, (...))\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R11
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto \mathit{null}), \mathit{Pr}:\langle(v:=(...); \mathit{email}:=\texttt{new Email}();$
$t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}), (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto\texttt{default(XML)}, t\mapsto 2, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon)\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R1, $\longrightarrow$ R2, $\longrightarrow$ R1
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto(1:\texttt{Email})),$
$\mathit{Pr}:\langle(t!\mathit{email.send}(\texttt{XML}\times\texttt{Client}\rightarrow\texttt{Data}, v, a); \texttt{free}(t); \texttt{return true}),$
$(d\mapsto(2007-12-31), a\mapsto \mathit{john.doe@email.com}, v\mapsto(...), t\mapsto 2, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon)\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{Email})\mid \mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})], \mathit{Pr}:\epsilon, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R8, $\longrightarrow$ R1, $\longrightarrow$ R15, $\longrightarrow$ R9, $\longrightarrow$ R10, $\longrightarrow$ R7, $\longrightarrow$ R13, $\longrightarrow$ R9,
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto(1:\texttt{Email})), \mathit{Pr}:\langle(\texttt{return true}), (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto(...), t\mapsto 3, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{3,\emptyset}:\mathit{comp}(3, \texttt{Bool}, \mathit{true})\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{Email})\mid \mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})], \mathit{Pr}:\langle\epsilon, \overline{l}\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\longrightarrow$ R16, $\longrightarrow$ R13
$\langle(1:\texttt{NewsService})\mid \mathit{Fld}:(\mathit{CNN}\mapsto(1:\texttt{News}), \mathit{email}\mapsto(1:\texttt{Email})), \mathit{Pr}:\langle\epsilon, (d\mapsto(2007-12-31),$
$a\mapsto \mathit{john.doe@email.com}, v\mapsto(...), t\mapsto 3, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{o}), \beta\mapsto\texttt{Bool})\rangle, \mathit{EvQ}_{\emptyset,3}:\epsilon\rangle$
$\langle(1:\texttt{News})\mid \mathit{Fld}:\mathit{fields}(\texttt{News})[\mathit{this}\mapsto(1:\texttt{News})], \mathit{Pr}:\langle\epsilon, (\overline{l}, \gamma\mapsto 2, \alpha\mapsto(1:\texttt{NewsService}), \beta\mapsto\texttt{XML})\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$
$\langle(1:\texttt{Email})\mid \mathit{Fld}:\mathit{fields}(\texttt{Email})[\mathit{this}\mapsto(1:\texttt{Email})], \mathit{Pr}:\langle\epsilon, \overline{l}\rangle, \mathit{PrQ}:\epsilon, \mathit{EvQ}_{\emptyset,\emptyset}:\epsilon\rangle$ $\mathit{comp}(2, \texttt{Bool}, \mathit{true})$ **to** $(1:\texttt{o})$

Figure 8. The execution sequence of method *newsRelay*

The initial runtime typing context Γ provides static information; i.e., types for the Booleans, `null`, and the fields and local variables of every class. Without loss of generality, we may assume that all classes, fields, and local variables have unique names. The context is gradually extended with types for identifiers of dynamically created objects and futures. The type system for runtime configurations is given in Figure 9. Runtime type judgments are on the form $\Gamma, \overline{q} \vdash C$ `ok` and express that a construct C is well-typed in Γ with a message queue $\overline{q}$. The message queue is used to type check reply assignments (in (Reply1) and (Reply2)). To simplify the presentation, $\overline{q}$ is omitted when not needed. The rules (Msg1) and (Msg2) are used to type check both messages in transit between objects and messages in the event queues. Type checking the process queue is similar to type checking the active process. Note that `lookup-error` and `deallocation-error` processes are not well-typed.

Definition 2 A runtime configuration *config* is well-typed in an environment Γ if $\Gamma \vdash$ *config* `ok`.

For a program $P = \overline{L}\ \{\overline{T\ x};\ \overline{s}\}$, if $\Gamma_0, \epsilon \vdash P \rhd \overline{L}'\ \{\overline{s}', \overline{x \mapsto \texttt{default}(T)}\}$, the *initial configuration* $\langle o\,|\, \mathit{Fld} : \epsilon, \mathit{Pr} \colon \langle \overline{s}', \overline{x \mapsto \texttt{default}(T)} \rangle, \mathit{PrQ} \colon \emptyset, \mathit{EvQ}_{(\emptyset,\emptyset)} : \emptyset\rangle$ is obviously well-typed in Γ_0 extended with types for the program's variables. A *method activation*, with correctly typed actual parameters, of any method in a well-typed program results in a well-typed runtime process. It is assumed that side effect free functional expressions are *type sound* in well-typed configurations [40]; if an expression e is evaluated in an object state σ in a well-typed configuration with typing environment Γ and $\Gamma \vdash e : T$, then $\Gamma(\llbracket e \rrbracket_\sigma) \preceq T$.

Example 5 To illustrate the typing of runtime configurations, consider the configuration derived from the initial configuration in Figure 8 by applying R2 and R1. By (Process), processes in $(1 : \texttt{NewsService})$ and $(1 : \texttt{News})$ are well-typed. An empty *eventQ* is well-typed by (State). For fields, after evaluating the statement $\mathit{CNN} := \texttt{new News}()$ of the initial configuration, the substitution $\mathit{CNN} \mapsto (1 : \texttt{News})$ is well-typed in the extended typing extended environment $\Gamma' = \Gamma + [(1 : \texttt{News}) \mapsto \texttt{News}]$ by (State). Thus, the configuration is well-typed.

5.1 Subject Reduction

We establish a subject reduction property in the style of Wright and Felleisen [40]. This property ensures that runtime configurations remain well-typed and that the deallocation operations inserted in the code are safe. The dynamic aspect of well-typed runtime configurations is due to the late binding of asynchronous method calls. Technically, this is represented as the absence of the `lookup-error` process in the runtime configurations. Deallocation operations are unsafe if they occur too early; i.e., if method returns are deallocated at a program point p then there should not be any read operations on those returns in the execution paths after p. Similar to method calls, safe deallocation is represented as the absence of the `deallocation-error` process.

$$\textsc{(Assign)}\ \frac{\Gamma \vdash e : T \quad T \preceq \Gamma(v)}{\Gamma \vdash v := e\ \texttt{ok}} \qquad \textsc{(New)}\ \frac{C \preceq \Gamma(v)}{\Gamma \vdash v := \texttt{new}\ C()\ \texttt{ok}} \qquad \textsc{(Skip)}\ \frac{}{\Gamma \vdash \texttt{skip ok}}$$

$$\textsc{(Await)}\ \frac{\Gamma \vdash g\ \texttt{ok}}{\Gamma \vdash \texttt{await}\ g\ \texttt{ok}} \qquad \textsc{(Release)}\ \frac{}{\Gamma \vdash \texttt{release ok}} \qquad \textsc{(free)}\ \frac{\forall\ t \in \overline{t} \cdot \Gamma(t) = \mathsf{Label}}{\Gamma \vdash \texttt{free}(\overline{t})\ \texttt{ok}}$$

$$\textsc{($\wedge$-Guards)}\ \frac{\Gamma \vdash \overline{s}_1\ \texttt{ok} \quad \Gamma \vdash \overline{s}_1\ \texttt{ok}}{\Gamma \vdash \overline{s}_1 \wedge \overline{s}_2\ \texttt{ok}} \textsc{(B-Guards)}\ \frac{\Gamma \vdash b : \mathsf{Bool}}{\Gamma \vdash b\ \texttt{ok}} \textsc{(t-Guards)}\ \frac{\Gamma(t) = \mathsf{Label}}{\Gamma \vdash t?\ \texttt{ok}}$$

$$\textsc{(Calls)}\ \frac{\Gamma \vdash e : C \quad \Gamma \vdash \overline{e} : T_1 \quad T_1 \preceq T \quad \mathit{lookup}(C, m, T \rightarrow T')}{\Gamma \vdash\ t!e.m(T \rightarrow T', \overline{e})\ \texttt{ok}}$$

$$\textsc{(Reply1)}\ \frac{\mathit{comp}(t, T, e) \in \overline{q} \quad T \preceq \Gamma(v)}{\Gamma, \overline{q}\ \vdash t?(v)\ \texttt{ok}} \qquad \textsc{(Choice)}\ \frac{\Gamma, \overline{q} \vdash \overline{s}_1\ \texttt{ok} \quad \Gamma, \overline{q} \vdash \overline{s}_2\ \texttt{ok}}{\Gamma, \overline{q} \vdash \overline{s}_1 \,\square\, \overline{s}_2\ \texttt{ok}}$$

$$\textsc{(Reply2)}\ \frac{\neg\exists\ e, T \cdot \mathit{comp}(t, T, e) \in \overline{q} \quad T \preceq \Gamma(v)}{\Gamma, \overline{q}\ \vdash t?(v)\ \texttt{ok}} \qquad \textsc{(Seq)}\ \frac{\Gamma, \overline{q} \vdash \overline{s}_1\ \texttt{ok} \quad \Gamma, \overline{q} \vdash \overline{s}_2\ \texttt{ok}}{\Gamma, \overline{q} \vdash \overline{s}_1;\ \overline{s}_2\ \texttt{ok}}$$

$$\textsc{(Msg1)}\ \frac{\Gamma \vdash \overline{e} : T \quad T \preceq T_1}{\Gamma \vdash \mathit{invoc}(m, T_1 \rightarrow\ T_2, \overline{e}, \langle o, n\rangle)\ \texttt{ok}} \qquad \textsc{(Msg2)}\ \frac{\Gamma \vdash e : T \quad T \preceq\ T'}{\Gamma \vdash \mathit{comp}(n, T', e)\ \texttt{ok}}$$

$$\textsc{(Msg3)}\ \frac{\Gamma \vdash \mathit{msg}\ \texttt{ok}}{\Gamma \vdash \mathit{msg}\ \textbf{to}\ o\ \texttt{ok}} \qquad \textsc{(EventQ)}\ \frac{\Gamma \vdash \mathit{eventQ}_1\ \texttt{ok} \quad \Gamma \vdash \mathit{eventQ}_2\ \texttt{ok}}{\Gamma \vdash \mathit{eventQ}_1\ \mathit{eventQ}_2\ \texttt{ok}}$$

$$\textsc{(Process)}\ \frac{\begin{array}{c}\Gamma, \overline{q} \vdash \overline{s}\ \texttt{ok} \quad \Gamma \vdash \mathit{sub}\ \texttt{ok} \\ \Gamma \vdash \overline{e} : T \quad T \preceq \mathit{sub}(\beta)\end{array}}{\Gamma, \overline{q} \vdash\ \langle \overline{s};\ \texttt{return}\ \overline{e}, \mathit{sub}\rangle\ \texttt{ok}} \qquad \textsc{(State)}\ \frac{\texttt{for all}\ (v \mapsto \mathit{val}) \cdot \Gamma(\mathit{val}) \preceq \Gamma(v)}{\Gamma \vdash\ \overline{v \mapsto \mathit{val}}\ \texttt{ok}}$$

$$\textsc{(ProcessQ)}\ \frac{\Gamma, \overline{q} \vdash \mathit{processQ}_1\ \texttt{ok} \quad \Gamma, \overline{q} \vdash \mathit{processQ}_2\ \texttt{ok}}{\Gamma, \overline{q} \vdash \mathit{processQ}_1\ \mathit{processQ}_2\ \texttt{ok}} \qquad \textsc{(IdleProcess)}\ \frac{}{\Gamma \vdash \texttt{idle ok}}$$

$$\textsc{(Object)}\ \frac{\begin{array}{c}\Gamma \vdash \mathit{sub}\ \texttt{ok} \qquad \Gamma, \mathit{eventQ} \vdash \mathit{active}\ \texttt{ok} \qquad \Gamma, \mathit{eventQ} \vdash \mathit{processQ}\ \texttt{ok} \\ \texttt{for all}\ t \cdot \Gamma(t) = \texttt{Label} \qquad \Gamma \vdash \mathit{eventQ}\ \texttt{ok}\end{array}}{\Gamma \vdash \langle o \mid \mathit{Fld} : \mathit{sub}, \mathit{Pr}\!:\ \mathit{active}, \mathit{PrQ}\!:\ \mathit{processQ}, \mathit{EvQ}_{\overline{t}}\!:\ \mathit{eventQ}\rangle\ \texttt{ok}}$$

$$\textsc{(Empty)}\ \frac{}{\Gamma \vdash \epsilon\ \texttt{ok}} \qquad \textsc{(Config)}\ \frac{\Gamma \vdash \mathit{config}_1\ \texttt{ok} \quad \Gamma \vdash \mathit{config}_2\ \texttt{ok}}{\Gamma \vdash \mathit{config}_1\ \mathit{config}_2\ \texttt{ok}}$$

Figure 9. The type system for runtime configurations.

Theorem 3 (Subject reduction) *Let config be an initial runtime configuration of a well-typed program such that* $\Gamma \vdash \mathit{config}$ *`ok`. If* $\mathit{config} \rightarrow \mathit{config}'$*, then there exists a* $\Gamma' \supseteq \Gamma$ *such that* $\Gamma' \vdash \mathit{config}'$ *`ok`.*

For well-typed programs, subject reduction leads to the following properties:

Corollary 3 *Let config be the initial configuration of a well-typed program. If* $\mathit{config} \rightarrow \mathit{config}'$*, then* config' *does not contain* `lookup-error`.

Corollary 4 *Let config be the initial configuration of a well-typed program. If* $\mathit{config} \rightarrow \mathit{config}'$*, then* config' *does not contain* `deallocation-error`.

6 Related work

For a detailed comparison of Creol with other distributed object languages here, see [19, 20]. *Asynchronous method calls* as discussed in this paper may be compared to message exchange in languages based on the Actor model [1]. Actor languages are conceptually attractive for distributed programming because they base communication on asynchronous messages, focusing on loosely coupled processes. An actor encapsulates its fields, procedures that manipulate the state, and a single thread of control. Creol objects are similar to actors, except that our methods return values managed by futures, and control can be released at specific points during a method execution. Messages to actors return no result and run to completion before another message is handled. The lack of return makes programming directly with actors cumbersome [1].

Futures express concurrency in a simple manner, reducing the latency dependency by enabling synchronization by necessity. Futures were discovered by Baker and Hewitt [3], and used by Halstead in MultiLisp [13] and by Liskov and Shrira as Promises [23]. Futures now appear in languages like Alice [31], Oz-Mozart [34], Concurrent ML [30], C++ [22], Java [18, 39], and Polyphonic C$^\sharp$ [4], often as libraries. Futures in these languages resemble labels in Creol. Implementations associate a future with the asynchronous execution of an expression in a new thread. The future is a placeholder which is immediately returned to the caller. From the perspective of the caller, this placeholder is a read-only structure [26]. However, the placeholder can also be explicitly manipulated to write data. In some approaches, the placeholder can be accessed in both modes (e.g., CML, Alice, Java, C++), though typically the caller and callee interfaces are distinct, as formalized by the calculus $\lambda(\mathsf{fut})$ [26]. Programming explicitly with promises is quite low-level, so Creol identifies the write operation on the future with method call return. This way, programming with future variables corresponds to programming with Creol's asynchronous method calls without processor release points; in fact, Creol's processor release points extend the computation flexibility originally motivating futures.

In this paper, we follow the approach of Promises [23] and use futures only inside the scope of a method body. This restriction is enforced by our type system (e.g., $\text{ran}(\Delta) = \perp$ in (Method)) and leads to a strict state encapsulation in objects. However futures can also be shared. In this case the futures become global variables, which breaks with object encapsulation but facilitates the delegation of method returns to other objects. Futures can be transparent or non-transparent. Transparent futures cannot be explicitly manipulated, the type of the future is the same as the expected result, and futures are accessed as ordinary program variables, possibly after waiting (e.g., in Multilisp). Flanagan and Felleisen present semantic models of futures in terms of an abstract machine [11]. In contrast to Creol, their language is purely functional. Caromel, Henrio, and Serpett present an imperative, asynchronous object calculus with transparent futures [6]. Their active objects may have internal passive objects which can be passed between the active objects by so-called “sheep” copying. This feature is orthogonal to the issues discussed in this paper.

Non-transparent futures have a separate type to denote the future, and future objects can be explicitly manipulated (e.g., in CML, Alice, Java, C++, and Creol). If the future is shared, its type includes the type of its value (e.g., `future T` is a future of type `T`). However, this limits the *reusability* of the future variable. A formalization and proof system for Creol’s concurrency model combined with shared non-transparent futures is presented in [8]. The present paper investigates a different approach, where the futures are encapsulated within the calling method but future variables can be shared between futures of different types, and shows how type analysis can be used to infer the underlying types such that method calls are bound correctly.

Type and effect systems add context information to type analyses [2, 14, 32] and have been used to, for example, ensure that guards controlling method availability do not have side effects [9], estimate the effects of an object reclassification [10], and ensure security properties. Gordon and Jeffrey use effects to track correspondences for authenticity properties of cryptographic protocols in the spi-calculus, by tracking matching begin- and end-assertions [12]. Broberg and Sands use effects to track different flow locks for dynamic flow policies by open and close operations for the different locks [5].

The decoupling of Creol’s invocation and reply assignments results in a scoped use of labels. Part of the complexity of our type system stems from the lack of explicit scoping. Scoping is implicitly given by the operations on a label, which may cause memory leakage. For a given label, an invocation need not be succeeded by a reply although a reply must be preceded by an invocation. Thus, the analysis formalized in our type system corresponds to a combination of several static analysis techniques: scope detection, type inference for labels, live variable analysis [27], and linearity analysis [36]. Linear type systems guarantee that certain values are used exactly once at runtime, and have been used for resource analysis [16, 36], including file handling and memory management. Type systems have also been used to impose linearity conditions and I/O usage for channels in the π-calculus [15, 29]. In particular, Igarashi and Kobayashi develop a type system and inference algorithm to identify linear use of channels inside explicitly given scopes [15]. They exploit linearity information in compile-time optimizations to reduce the number of dynamically created processes, and to refine the runtime representation of linear channels and operations on these. Whereas labels in Creol always have linearity restrictions, channels may but need not be linear, which adds complexity. The authors identify reclamation of memory as an important application of linearity-based optimization, but leave specific solutions for future work.

The effect system used to infer return types for method calls contains sufficient information to automatically insert `free` operations which enable the deallocation of method replies. This usage of type and effect systems is related to the region-based approach to memory management proposed by Tofte and Talpin [33]. In their work, programs written in a kernel language extracted from SML are transformed by means of region inference analysis into a target program with annotated allocation and deallocation operations. The inference rules use an effect system to capture operations on regions. However the language does not consider branching structures. In Creol deallocation operations are inserted locally for each execution branch. This allows memory leaks to be restricted when the reply to method calls is only needed in certain execution branches. Explicit deallocation is attractive for memory management because it can be implemented using simple constant-time routines [37]. The automatic insertion of deallocation statements, also found in [33, 37], lets the programmer ignore low-level memory management issues, for which it is easy to make mistakes leading to memory leaks. Thus the local deallocation operations automatically inserted for method replies are much simpler than the tracing garbage collector proposed by Baker and Hewitt [3] in order to avoid memory leakage due to redundant futures.

Backwards analysis facilitates an *eager deallocation strategy* which allows the deallocation of method returns in an execution dependent manner even if the return is still accessible from memory. This seems desirable in the case of, e.g., tail-recursive calls. Linear types have been proposed as a basis for eager deallocation [16], as linear values can be garbage collected immediately after being used even if they are still referenced, improving on a tracing garbage collector. This approach is incorporated directly in our operational semantics, because rule R9 consumes the method return (in contrast to reply guards in rule R3). With shared futures, this could no longer be done directly. To the best of the authors' knowledge, an explicit deallocation strategy for shared futures is an open issue. In future work, we plan to extend the approach of this paper to Creol with shared futures [8]. The deallocation of shared method replies seems to require more overhead in the operational semantics, as the deallocation depends on the execution in all objects which might access the future. In future work, we further plan to investigate how far type-based insertion of deallocation operations extends to deallocate the active objects themselves, thus avoiding distributed garbage collectors for the active objects [35, 38].

7 Conclusion

This paper presents a kernel language for distributed concurrent objects communicating by asynchronous method calls. The approach emphasizes flexibility with respect to the possible delays and instabilities of distributed computing, and allows active and reactive behavior to be combined in an object. Asynchronous method calls and process release points add flexibility to execution in the distributed setting because waiting activities may yield processor control to suspended and enabled processes. A type and effect system for backwards analysis of programs is introduced, using effects to track information about method calls. A particular effect of our system is the translation of source programs into runtime code. The effects of the type analysis are explicitly used to expand each asynchronous call statement with a type-correct signature, ensuring a type-correct runtime method lookup. With the proposed type and effect system, this can be done directly in the analysis of the method invocation. Thus, the type analysis and translation to runtime code is done in one pass.

The analysis is exploited to avoiding memory leakage for passively stored method replies at runtime. The lifetime of label values is statically determined and deallocation operations are inserted by the type system, so no runtime traversal of labels is needed to locate redundant method replies. By exploiting effects in a backwards manner, we derive a fine-grained deallocation strategy. Deallocation operations are inserted in the specific execution branches where the method reply is redundant, even if the reply is reachable from the object state. The backwards analysis enables an eager strategy in the sense that deallocation instructions are inserted as early as possible in each branch after the method invocation. An advantage of explicit deallocation is that deallocation can be handled by a simple routine. Another advantage of the approach of this paper is that the insertion of these operations is incorporated in the type analysis. In fact, the correctness of the deallocation operations is asserted from the type system; i.e., a deallocation is safe if the method reply can no longer be accessed from the code. A subject reduction property is established for the language: method binding is guaranteed to succeed at runtime, deallocation operations are safe in all execution paths, and the redundant method replies of the different execution branches are deallocated.

Acknowledgment. We gratefully acknowledge the comments from the anonymous reviewers. This paper was finalized while the first author was enjoying the hospitality of the United Nations University - International Institute for Software Technology in Macau.

References

[1] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153. Chapman & Hall, 1996.

[2] T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems.* Imperial

College Press, London, 1st edition, 1999.

[3] H. G. Baker and C. E. Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[5] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. 15th European Symposium on Programming (ESOP'06)*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2006.

[6] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM Press, 2004.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[8] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, Mar. 2007.

[9] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer, Aug. 1996.

[10] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{\mathrm{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

[11] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[12] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 145–159. IEEE Computer Society Press, 2001.

[13] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[14] F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory management. In Pierce [28], chapter 3, pages 87–135.

[15] A. Igarashi and N. Kobayashi. Type reconstruction for linear π-calculus with I/O subtyping. *Information and Computation*, 161(1):1–44, 2000.

[16] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, Mar. 2005.

[17] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[18] G. S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In A. Omondi and S. Sedukhin, editors, *Proc. 8th Asian-Pacific Computer Systems Architecture Conference (ACSAC 2003)*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003.

[19] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[20] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[21] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley, 1996.

[22] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In J. O. Coplien, J. Vlissides, and N. Kerth, editors, *Proc. Pattern Languages of Program Design*, pages 483–499. Addison-Wesley, 1996.

[23] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[24] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[25] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, Mar. 2007.

[26] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, Berlin, Germany, 2 edition, 2005.

[28] B. C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2004.

[29] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[30] J. H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[31] A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, Feb. 2006.

[32] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[33] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[34] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Mar. 2004.

[35] N. Venkatasubramanian, G. Agha, and C. L. Talcott. Scalable distributed garbage collection for systems of active objects. In Y. Bekkers and J. Cohen, editors, *International Workshop on Memory Management (IWMM'92)*, volume 637 of *Lecture Notes in Computer Science*, pages 134–147. Springer, 1992.

[36] D. Walker. Substructural type systems. In Pierce [28], chapter 1, pages 1–43.

[37] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.

[38] W.-J. Wang and C. A. Varela. Distributed garbage collection for mobile actor systems: The pseudo root approach. In Y.-C. Chung and J. E. Moreira, editors, *Advances in Grid and Pervasive Computing (GPC'06)*, volume 3947 of *Lecture Notes in Computer Science*, pages 360–372. Springer, 2006.

[39] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[40] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[41] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* Series in Computer Systems. The MIT Press, 1990.

A Proofs

A.1 Proof of Theorem 1

The proof is by induction on the derivation of $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta'$. For the base case, $\Gamma, \epsilon \vdash \texttt{skip} \rhd \texttt{skip}$. By definition, $\mathit{live}(t, \epsilon) = \texttt{false}$ and $\epsilon(t) = \perp$. For the induction step, the induction hypothesis (IH) is that $\neg \mathit{live}(t, \overline{s}_0) \iff$

$(\Delta)(t) =\perp$ holds for $\Gamma, \epsilon \vdash \overline{s}_0 \rhd \overline{s}'_0, \Delta$ and consider $\Gamma, \Delta \vdash s \rhd \overline{s}', \Delta'$ by case analysis on s.

Cases (Assign)*,* (Skip)*, and* (New) follow directly from IH.

Case (Await). We have $s = \texttt{await}\ g$. There are two subcases for every label t, depending on whether t is contained in g:

(1) *Guard g contains t?.* From Definition 1 we have $\mathit{live}(t, s; \overline{s}_0)$. Rule ($t$-Guards) results in the update $[t \mapsto \Delta(t) \cap \texttt{Data}]$ which propagates to the update Δ' for the `await` statement in (Await). Consequently, $(\Delta \cdot \Delta')(t) \neq\perp$.
(2) *Guard g does not contain a reply guard on t.* From Definition 1 we get $\mathit{live}(t, s; \overline{s}_0) = \mathit{live}(t, \overline{s}_0)$. From the type analysis of g we get $t \notin \mathrm{dom}(\Delta')$, and it follows by (Await) that $(\Delta \cdot \Delta')(t) = \Delta(t)$. Consequently, $\neg\mathit{live}(t, s; \overline{s}_0) \iff (\Delta \cdot \Delta')(t) =\perp$ follows from IH.

It follows that $\forall l : \mathsf{Label} \cdot \neg\mathit{live}(t, \texttt{await}\ g; \overline{s}_0) \iff (\Delta \cdot \Delta')(t) =\perp$.

Case (Reply). From Definition 1 we have $\mathit{live}(t, t?(v); \overline{s}_0)$. The effect of (Reply) is $[t \mapsto T]$ for some type T, so $(\Delta \cdot \Delta')(t) \neq\perp$. For other labels t', $\neg\mathit{live}(t', s; \overline{s}_0) \iff (\Delta \cdot \Delta')(t') =\perp$ follows directly from IH.

Case (Call1). From Definition 1, $\neg\mathit{live}(t, t!o.m(\overline{e}); \overline{s}_0)$. From (Call1), $\Delta'(t) =\perp$. For other labels t', $\neg\mathit{live}(t', t!o.m(\overline{e}); \overline{s}_0) \iff (\Delta \cdot \Delta')(t') =\perp$ follows from IH.

Case (Call2). Similar to (Call1).

Case (Choice). We assume $\mathit{well\text{-}defined}((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$ and that IH holds for the branches $\overline{s}_1$ (IH1) and $\overline{s}_2$ (IH2). From Definition 1 we have $\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0) = \mathit{live}(t, \overline{s}_1; \overline{s}_0) \vee \mathit{live}(t, \overline{s}_2; \overline{s}_0)$. Now assume that $\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0)$. There are three subcases for every label t:

(1) $\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\neg\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. In this subcase, we get $(\Delta \cdot \Delta_1)(t) \neq\perp$ by IH1 and $(\Delta \cdot \Delta_2)(t) =\perp$ by IH2. It follows that $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) \neq\perp$.
(2) $\neg\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. Similar to Subcase 1.
(3) $\mathit{live}(t, \overline{s}_1; \overline{s}_0)$ and $\mathit{live}(t, \overline{s}_2; \overline{s}_0)$. In this subcase, we get $(\Delta \cdot \Delta_1)(t) \neq\perp$ by IH1 and $(\Delta \cdot \Delta_2)(t) \neq\perp$ by IH2. Since $\mathit{well\text{-}defined}((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$, there must be some type T where $T \neq \perp$ and $T \neq \texttt{Error}$ such that $((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))(t) = T$.

It follows that $\forall l : \mathsf{Label} \cdot \neg\mathit{live}(t, \overline{s}_1 \Box \overline{s}_2; \overline{s}_0) \iff (\Delta \cdot ((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2)))(t) =\perp$.

Case (Seq) follows by induction over the length of s. □

A.2 Proof of Theorem 2

To make the link between method invocations and possible matching reply assignments explicit we introduce a notion of *reachable* replies, inspired by static analysis techniques [27]. Intuitively, a reply assignment $t?(v)$ is reachable in a statement list $\overline{s}$ if there exists an execution path through $\overline{s}$ in which $t?(v)$ is encountered before a method invocation with label t.

Definition 3 The set $\mathcal{R}(\overline{s})$ of reachable reply assignments in $\overline{s}$ is defined as

$$\begin{array}{l}
\mathcal{R}(\epsilon) = \emptyset \\
\mathcal{R}(s;\overline{s}) = \mathcal{R}(\overline{s}) \quad \text{if } s \in \{v := e, v := \texttt{new}\ C, \texttt{await}\ g, \texttt{skip}\} \\
\mathcal{R}(t?(v);\overline{s}) = \{t?(v)\} \cup \mathcal{R}(\overline{s}) \\
\mathcal{R}(t!o.m(\overline{e});\overline{s}) = \{t'?(v) \in \mathcal{R}(\overline{s}) \mid t \neq t'\} \\
\mathcal{R}(\overline{s}_1 \,\Box\, \overline{s}_2;\overline{s}) = \mathcal{R}(\overline{s}_1;\overline{s}) \cup \mathcal{R}(\overline{s}_2;\overline{s})
\end{array}$$

The definition of reachable reply assignments resembles the definition of live variables; in fact, if $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta'$ then $t?(v) \in \mathcal{R}(\overline{s}) \Rightarrow \mathit{live}(t, \overline{s})$. The implication only goes one way because reply guards also affect whether a label is live. We now establish a lemma which is used in the proof of Theorem 2.

Lemma 1 *Let* $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta$ *be well-typed. Then* $\forall t?(v) \in \mathcal{R}(\overline{s}) \cdot \Delta(t) \preceq \Gamma(v)$.

Proof. By induction on the derivation of $\Gamma, \epsilon \vdash \overline{s} \rhd \overline{s}', \Delta$. For the base case, $\mathcal{R}(\epsilon) = \emptyset$. For the induction step, assume as induction hypothesis (IH) that $\forall t?(v) \in \mathcal{R}(\overline{s}_0) \cdot \Delta'(t) \preceq \Gamma(v)$ for $\Gamma, \epsilon \vdash \overline{s}_0 \rhd \overline{s}_0', \Delta'$. Consider $\forall t?(v) \in \mathcal{R}(s;\overline{s}_0) \cdot \Delta(t) \preceq \Gamma(v)$ for $\Gamma, \epsilon \vdash s;\overline{s}_0 \rhd s';\overline{s}_0', \Delta$ by case analysis over s.

Cases (Assign)*,* (Skip)*,* (New)*, and* (Await) follow directly from IH.

Case (Reply). From Definition 3, $\mathcal{R}(t?(v);\overline{s}_0) = \{t?(v)\} \cup \mathcal{R}(\overline{s}_0)$. From (Reply), $\Delta(t) = \Gamma(v)$ and by IH we get $\forall t'?(v') \in \mathcal{R}(t'(v);\overline{s}_0) \cdot \Delta(t') \preceq \Gamma(v')$.

Case (Call1). From Definition 3, $\mathcal{R}(t!o.m(\overline{e});\overline{s}) = \{t'?(v) \in \mathcal{R}(\overline{s}) \mid t \neq t'\}$. Hence, the case follows directly from IH.

Case (Call2). Similar to (Call1).

Case (Choice). We assume that *well-defined*$((\Delta \cdot \Delta_1) \cup (\Delta \cdot \Delta_2))$ and that IH holds for the branches $\overline{s}_1$ (IH1) and $\overline{s}_2$ (IH2). From Definition 3, we have $\mathcal{R}(\overline{s}_1 \,\Box\, \overline{s}_2;\overline{s}_0) = \mathcal{R}(\overline{s}_1;\overline{s}_0) \cup \mathcal{R}(\overline{s}_2;\overline{s}_0)$. There are three cases for a label t:

(1) $t?(v) \in \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \notin \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_1)(t)$. By IH1, $(\Delta_0 \cdot \Delta_1)(t) \preceq \Gamma(v)$.
(2) $t?(v) \notin \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \in \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_2)(t)$. By IH2, $(\Delta_0 \cdot \Delta_2)(t) \preceq \Gamma(v)$.

(3) $t?(v) \in \mathcal{R}(\overline{s}_1;\overline{s}_0)$ and $t?(v') \in \mathcal{R}(\overline{s}_2;\overline{s}_0)$. We have $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) = (\Delta_0 \cdot \Delta_1)(t) \cap (\Delta_0 \cdot \Delta_2)(t)$. Moreover we have that $(\Delta_0 \cdot \Delta_1)(t) \neq \bot$, $(\Delta_0 \cdot \Delta_2)(t) \neq \bot$, $(\Delta_0 \cdot \Delta_1)(t) \neq \texttt{Error}$, and $(\Delta_0 \cdot \Delta_2)(t) \neq \texttt{Error}$. Since *well-defined*$((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))$, we know that $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq (\Delta_0 \cdot \Delta_1)(t)$ and $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq (\Delta_0 \cdot \Delta_2)(t)$. By IH1 and IH2, we then obtain $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq \Gamma(v)$ and $((\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2))(t) \preceq \Gamma(v')$

It follows that $\forall t?(v) \in \mathcal{R}(\overline{s}_1 \,\Box\, \overline{s}_2; \overline{s}) \cdot \Delta(t) \preceq \Gamma(v)$.

Case (Seq) follows by induction over the length of s. ☐

Proof of Theorem 2. There are two cases, depending on the rule used to derive a signature for a method invocation. For (Call1), $\Delta(t) \neq \bot$. It follows from Lemma 1 that any reachable reply assignment $t?(v)$ is such that $\Delta(t) \preceq \Gamma(v)$. For (Call2), $\Delta(t) = \bot$. By Theorem 1, t is not live after the method invocation. Hence, there are no reachable reply assignments. ☐

A.3 Proof of Theorem 3

Consider a well-typed program $P = \overline{L}\ \{\overline{T\ x}, \overline{s}\}$ and assume that the judgment $\Gamma_0, \epsilon \vdash P \rhd \overline{L'}\ \{\overline{s}', \overline{x \mapsto \texttt{default}(T)}\}$ holds. The proof is by induction over the length of an execution *config*$_0$, *config*$_1, \ldots$. Let Γ_1 extend Γ_0 with types for variables declared in P (for simplicity assuming uniqueness of names). For the *base case*, the initial runtime configuration *config*$_0$ is well-typed:

$$\Gamma_1 \vdash \langle o \mid \mathit{Fld}: \epsilon, \mathit{Pr}\colon \langle \overline{s}', \overline{x \mapsto \texttt{default}(T)}\rangle, \mathit{PrQ}\colon \emptyset, \mathit{EvQ}_{(\emptyset,\emptyset)}: \emptyset\rangle.$$

For the *induction step*, assume that $\Gamma \vdash \langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle s;\overline{s},\overline{l}\rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$ *config* $\texttt{ok}$ and consider the reduction

$$\begin{aligned}&\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle s;\overline{s},\overline{l}\rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle\ \mathit{config}\\ &\quad \to \langle o \mid \mathit{Fld} : \overline{f}', \mathit{Pr} : \langle s';\overline{s},\overline{l}'\rangle, \mathit{PrQ} : \overline{w}', \mathit{EvQ}_{(\mathcal{L}',\mathcal{G}')} : \overline{q}'\rangle\ \mathit{config}.\end{aligned}$$

The proof proceeds by case analysis over the reduction rules.

Case R1. The process $\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \langle v := e;\overline{s},\overline{l}\rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$ reduces to $\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \overline{s}, (\overline{l}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})}])\rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$ or to $\langle o \mid \mathit{Fld} : \overline{f}[v \mapsto [\![e]\!]_{(\overline{f}\cdot\overline{l})} \mathit{Pr} : \langle \overline{s},\overline{l}\rangle, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$. Since the object is well-typed, we may assume from (Assign) that $\Gamma(v) = T$, $\Gamma \vdash e : T'$, $T' \preceq T$, and $\Gamma([\![e]\!]_{(\overline{f}\cdot\overline{l})}) = T''$ such that $T'' \preceq T'$. By transitivity $T'' \preceq T$ and by either (Process) or (State) the object is well-typed after the assignment.

Case R2. For $Pr = \langle v := \texttt{new}\ C(); \overline{s}, \overline{l}\rangle$, it follows from (New) that $C \preceq \Gamma(v)$. Consequently, the reduced process $\langle v := (n : C); \overline{s}, \overline{l}\rangle$ is well-typed in the *extended typing context* $\Gamma' = \Gamma[(n : C) \mapsto C]$. The predicate *fresh*$(n)$ guarantees that $(n : C)$ is a unique object identifier, so we have $\Gamma \subseteq \Gamma'$. For the new object $(n : C)$, the state provided by *fields*(C) is well-typed by assumption (binding declared variables to default values of their respective types) and the assignment $\texttt{this} := (n : C)$ is well-typed in Γ'. Finally the queues are empty and, by (Object), the new object is well-typed in Γ'.

Case R3. The process $Pr = \langle \texttt{await}\ g; \overline{s}, \overline{l}\rangle$ reduces to $\langle \overline{s}, \overline{l}\rangle$. By IH, $\Gamma \vdash \langle \overline{s}, \overline{l}\rangle\ \texttt{ok}$.

Case R4. By assumption, the source program is well-typed, so all method bodies in the source program are also well-typed. The considered process is a reduction of the translated runtime code of a well-typed method body. Hence, there must be a static judgment $\Gamma, \epsilon \vdash \texttt{await}\ g \rhd \texttt{await}\ g, \Delta$, where Γ is the static typing context for the method body. Note that the effect of the (Await) rule ensures that $\Delta(t) \neq \bot$. Furthermore two label variables in an object cannot have the same value, due to the freshness condition in rule R7 and the exclusion of explicit assignment to label variables in typing rule (Assign).

Now assume that R4 were applicable at runtime. Then *dealloc*$(g, \overline{f} \cdot \overline{l}, G)$ must hold; i.e., for some guard $t? \in g$, the completion message associated with t has already been deallocated before the execution arrives at $\texttt{await}\ g$. Consequently, a statement $\texttt{free}(t)$ must have been inserted into the runtime code before $\texttt{await}\ g$ by the type-based translation from the source code of the method body, such that the value bound to t is unchanged by the execution between $\texttt{free}(t)$ and $\texttt{await}\ g$.

Let Δ_1 be the effect after the analysis of $\overline{s1}$ in the context Γ, Δ_0. In order for the type-based translation to insert the statement $\texttt{free}(t)$ in the runtime code, three subcases correspond to the different possible judgments:

(1) $\Gamma, \Delta_0 \vdash \texttt{await}\ g'; \overline{s1} \rhd \texttt{await}\ g'; \texttt{free}(t); \overline{s1}', \Delta_0'$
(2) $\Gamma, \Delta_0 \vdash t!e.m(\overline{e}); \overline{s1} \rhd t!e.m(T \to \texttt{Data}, \overline{e}); \texttt{free}(t); \overline{s1}', \Delta_0'$
(3) $\Gamma, \Delta_0 \vdash \overline{s1} \Box \overline{s2} \rhd (\texttt{free}(t); \overline{s1}) \Box \overline{s2}, (\Delta_0 \cdot \Delta_1) \cup (\Delta_0 \cdot \Delta_2)$

In subcases 1 and 2, $\Delta_0(t) = \bot$ by rule (Await) or (Call2) in the static type system. In subcase 3, $\Delta_0 = \bot$ and $\Delta_1 = \bot$ by rule (Choice). (The case where $\texttt{free}(t)$ occurs in the right hand branch is similar. The cases where both $\texttt{await}\ g$ and $\texttt{free}(t)$ occurs in one of the branches are covered previously.) Since the value bound to t at runtime does not change between $\texttt{free}(t)$ and $\texttt{await}\ g$, it follows by induction over the program statements preceding $\texttt{await}\ g$, that $\Delta_0(t) = \Delta(t) \neq \bot$, and we get a contradiction. Thus, rule R4 is not applicable in the execution from a well-typed source program.

Case R5. We have the process $Pr = \langle \texttt{await}\ g; \overline{s}, \overline{l}\rangle$ and the object reduces to

$\langle o | \mathit{Fld} : \overline{f}, \mathit{Pr} : \texttt{release}; \texttt{await}\ g; \overline{s}, \mathit{PrQ} : \overline{w}, \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$. As Pr is well-typed, the resulting configuration is well-typed.

Case R6. We have the process $\mathit{Pr} = \langle \texttt{release}; \overline{s}, \overline{l}\rangle$ and the object reduces to $\langle o \mid \mathit{Fld} : \overline{f}, \mathit{Pr} : \texttt{idle}, \mathit{PrQ} : (\overline{w}\ \langle \overline{s}, \overline{l}\rangle), \mathit{EvQ}_{(\mathcal{L},\mathcal{G})} : \overline{q}\rangle$. As Pr and PrQ are well-typed, the resulting configuration is well-typed.

Case R7. By IH PrQ is well-typed, so $\Gamma \vdash \langle \overline{s}, \overline{l}\rangle\ \overline{w}\ \texttt{ok}$. For PrQ, $\Gamma \vdash \overline{w}\ \texttt{ok}$ and for Pr, $\Gamma \vdash \langle \overline{s}, \overline{l}\rangle\ \texttt{ok}$. Consequently, by (Object), the object is well-typed.

Case R8. The process $\mathit{Pr} = \langle t!r(\mathit{Sig}, \overline{e}); \overline{s}, \overline{l}\rangle$ reduces to $\langle t := \mathit{mid}; \overline{s}, \overline{l}\rangle$. By IH, $\Gamma \vdash \langle t!r(\mathit{Sig}, \overline{e}); \overline{s}, \overline{l}\rangle\ \texttt{ok}$, and we may assume $\Gamma(t) = \texttt{Label}$. Since mid has type $\texttt{Label}$ the assignment is well-typed, and $\Gamma \vdash \langle t := \mathit{mid}; \overline{s}, \overline{l}\rangle\ \texttt{ok}$. A message $\mathit{invoc}(m, T \to T', ([\![e]\!]_{(\overline{f}\cdot\overline{l})}), \langle o, \mathit{mid}\rangle)\ \textbf{to}\ [\![x]\!]_{(\overline{f}\cdot\overline{l})}$ is added to the configuration. (An invocation message from o is uniquely identified by mid) Since $\Gamma \vdash t!x.m(T \to T', \overline{e})\ \texttt{ok}$, we have $\Gamma \vdash \overline{e} : T$, $\Gamma \vdash [\![e]\!]_{(\overline{f}\cdot\overline{l})} : T'$, and $T' \preceq T$. Consequently, by (Msg3) we get $\Gamma \vdash \mathit{invoc}(m, T \to T', ([\![e]\!]_{(\overline{f}\cdot\overline{l})}), \langle o, \mathit{mid}\rangle)\ \textbf{to}\ [\![x]\!]_{(\overline{f}\cdot\overline{l})}\ \texttt{ok}$.

Case R9. We have a well-typed message $\Gamma \vdash \mathit{msg}\ \textbf{to}\ o\ \mathit{ok}$ and a well-typed object o with event queue $\mathit{EvQ}_{(\mathcal{L},\mathcal{G})} = \overline{q}$. Since $\Gamma \vdash \mathit{msg}\ \texttt{ok}$ and $\Gamma \vdash \overline{q}\ \texttt{ok}$, by (EventQ) we get $\Gamma \vdash \overline{q}\ \mathit{msg}\ \texttt{ok}$.

Case R10. We have $\mathit{EvQ}_{(\mathcal{L},\mathcal{G})} = \overline{q}\ \mathit{invoc}(m, \mathit{Sig}, \overline{e}, \langle o, \mathit{mid}\rangle)$ and $\mathit{PrQ} = \overline{w}$. The reduction results in $\mathit{PrQ} = \overline{w}\ \mathit{lookup}(C, m, \mathit{Sig}, (\overline{e}, o, \mathit{mid}))$ and $\mathit{EvQ}_{(\mathcal{L},\mathcal{G})} = \overline{q}$. Since $\Gamma \vdash \overline{q}\ \mathit{invoc}(m, \mathit{Sig}, \overline{e}, \langle o, \mathit{mid}\rangle)\ \texttt{ok}$, $\Gamma \vdash \overline{q}\ \texttt{ok}$. By assumption every object identifier is unique, so the class of o has among its superclasses the statically assumed class for which $\mathit{lookup}(\Gamma(o), m, \mathit{Sig})$ succeeds. Consequently, the runtime $\mathit{lookup}(\Gamma(o), m, \mathit{Sig}, (\overline{e}, o, \mathit{mid}))$ returns a process which correctly matches the call and which is well-typed by (Process).

Case R11. We have $\mathit{Pr} = \langle t?(v); \overline{s}, \overline{l}\rangle$ and $\mathit{EvQ}_{(\mathcal{L},\mathcal{G})} = \overline{q}\ \mathit{comp}(\mathit{mid}, T, e)$ such that t is bound to mid in object o. The process reduces to $\langle v := e; \overline{s}, \overline{l}\rangle$ and the event queue to $\overline{q}$. As completion messages result from method invocations only and mid is fresh by R8, there must be a corresponding well-typed invocation with label t, say $t!x.m(T' \to T'', \overline{e})$, such that $T'' \preceq \Gamma(v)$, $\Gamma(x) = C$, and $\mathit{lookup}(C, m, T' \to T'')$. Consequently $T \preceq T''$, and by Theorem 2 and transitivity, $T \preceq \Gamma(v)$, so (Reply1) holds and $\Gamma \vdash \langle v := e; \overline{s}, \overline{l}\rangle\ \texttt{ok}$ and $\Gamma \vdash \overline{q}\ \texttt{ok}$.

Case R12. Similar to *Case R4*.

Case R13. The completion message $\mathit{comp}([\![\gamma]\!]_{\overline{l}}, [\![\beta]\!]_{\overline{l}}, [\![v]\!]_{\overline{f}\cdot\overline{l}})\ \textbf{to}\ [\![\alpha]\!]_{\overline{l}}$ is introduced into the configuration. The process reduces to $\langle \overline{s}, \overline{l}\rangle$, which is well-typed. Since the program is statically well-typed, we may assume $\Gamma \vdash v : T$ and $\Gamma \vdash [\![v]\!]_{\overline{l}} : T'$ such that $T' \preceq T$. The typing rule (Method) asserts $T \preceq [\![\beta]\!]_{\overline{l}}$. Consequently, $T' \preceq [\![\beta]\!]_{\overline{l}}$ and by (Msg3) we get $\Gamma \vdash \mathit{comp}([\![\gamma]\!]_{\overline{l}}, [\![\beta]\!]_{\overline{l}}, [\![v]\!]_{\overline{l}})\ \textbf{to}\ [\![\alpha]\!]_{\overline{l}}\ \texttt{ok}$.

Case R14. We have $Pr = \langle \overline{s}_1 \Box \overline{s}_2 \rangle$, which reduces $\overline{s}_1 \Box \overline{s}_2$ to either $\overline{s}_1$ or $\overline{s}_2$. The IH gives us directly that the object is well-typed.

Case R15. We have $Pr = \langle (\texttt{free}(\overline{t}); \overline{s}), \overline{l} \rangle$ and $EvQ_{(\mathcal{L},\mathcal{G})} = \overline{q}$. Since the program is well-typed, $\Gamma(t) = \mathsf{Label}$. It follows from IH that the object is well-typed.

Case R16. We have the event queue $EvQ_{(\{mid\} \cup \mathcal{L},\mathcal{G})} = comp(mid, T, e)\ \overline{q}$. By IH, $\Gamma \vdash comp(mid, T, e)\ \overline{q}\ \texttt{ok}$. Consequently by (EventQ), $\Gamma \vdash \overline{q}\ \texttt{ok}$. $\Box$