Efficient and Fully Abstract Routing of Futures in Object Network Overlays

Mads Dam and Karl Palmskog
School of Computer Science and Communication
KTH Royal Institute of Technology
{mfd,palmskog}@kth.se

Abstract

In distributed object systems, it is desirable to enable migration of objects between locations, e.g., in order to support efficient resource allocation. Existing approaches build complex routing infrastructures to handle object-to-object communication, typically on top of IP, using, e.g., message forwarding chains or centralized object location servers. These solutions are costly and problematic in terms of efficiency, overhead, and correctness. We show how location independent routing can be used to implement object overlays with complex messaging behavior in a sound, fully abstract, and efficient way, on top of an abstract network of processing nodes connected point-to-point by asynchronous channels. We consider a distributed object language with futures, essentially lazy return values. Futures are challenging in this context due to the strong global consistency requirements they impose. The key conclusion is that execution in a decentralized, asynchronous network can preserve the standard, network-oblivious behavior of objects with futures, in the sense of contextual equivalence. To the best of our knowledge, this is the first such result in the literature. We also believe the proposed execution model may be of interest in its own right in the context of large-scale distributed computing.

1 Introduction

The ability to transparently and efficiently relocate objects between processing nodes is a basic prerequisite for many tasks in large-scale distributed systems, including tasks such as load balancing, resource allocation, and management. By freeing applications from the burden of resource management, they can be made simpler, more resilient, and easier to manage, resulting in a lower cost for development, operation and management.

The key problem is how to efficiently handle object and task mobility. Since object locations change dynamically in a mobile setting, some form of application-level routing is needed for inter-object messages to reach their destinations. Various approaches have been considered in the literature; Sewell et al. [36] provide a comprehensive survey. One common imple-

mentation strategy is to use some form of centralized, replicated, or decentralized object location register, either for forwarding or for address lookup [1, 13, 17, 36]. This type of solution requires some form of synchronization to keep registers consistent with physical locations, or else it needs to resort to some form of message relaying, or forwarding. Forwarding by itself is another main implementation strategy used in, e.g., the Emerald system [25], or in more recent systems like JoCaml [10]. Other solutions exist, such as broadcast or multicast search, that are useful for recovery or for service discovery, but hardly efficient as general purpose routing devices in large systems.

In general, we consider a mechanism for object mobility with the following properties desirable:

Low stretch In stable state, the ratio between actual and optimal route lengths (costs) should be small.

Compactness The space required at each node for storing route information should be small (sublinear in the number of destinations).

Self-stabilization Even when started in a transient state, computations should proceed correctly, and converge to a stable state. Observe that this precludes the use of locks.

Decentralization To enable scaling to large networks with many objects and tasks, routes and next-hop destinations should be computed in a decentralized fashion, at the individual nodes, and not rely on a centralized facility.

Existing solutions are quite far from meeting these requirements: Location registers (centralized or decentralized) and pointer forwarding regimes both preclude low stretch, and the use of locks precludes self-stabilization.

In earlier work [11], we suggest that the root of the difficulties lies in a fundamental mismatch between the information used for search and identification (typically, object identifiers, OIDs), and the information used for routing, namely, host identifiers, typically IP addresses. If we were to route messages not to the destination *location*, but instead to the destination *object*, it should be possible to build object network overlays which much better fit the desiderata laid out above. In previous work [11], we show that this indeed appears to be true (even if the problem of compactness is left for future investigation). The key idea is to use a form of location independent (also known as *flat*, or *name independent*) routing [2, 20, 21] that allows messages (RPCs) to be routed directly to the called object, independently of the physical node on which that object is currently executing. Using location independent routing, a lot of the overhead and performance constraints associated with object mobility can be eliminated, including latency and bandwidth overhead due to looking up, querying, updating, and locking object location databases, and overhead due to increased traffic for, e.g., message forwarding.

The language considered previously [11] allows to define a collection of objects communicating by asynchronous RPC, and thus its functionality

is not much different from a core version of Erlang [4], or the Nomadic Pict language studied by Sewell et al. [36]. The question we raise is how program behavior is affected by being run in the networked model, as compared with a more standard, network-oblivious "reference" semantics given in the style of rewriting logic [9]. This comparison is of interest, since the reference semantics is given at a high level of abstraction and ignores almost all aspects of physical distribution, such as location, routing, and message passing. We show that, with a maximally nondeterministic network-aware semantics, and in the sense of contextual equivalence [30], programs exhibit the same behavior in both semantics.

Messaging in our earlier work is very simple. The implicit channel abstraction used in the reference semantics is essentially that of a reliable, unordered communication channel. Messages (method calls) are sent according to the program order, but the order in which they are acted upon is arbitrary. Soundness and full abstraction for the network-aware semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behavior to transfer to a networked realization.

In this paper, we address the question of how sensitive these results are to the type of communication taking place at the abstract level. The overlays considered in our earlier work allow only one type of message, with modest requirements on global consistency. It is of interest to examine also languages allowing more complex communication behavior for objects. To this end, we define the richer language mABS, corresponding essentially to a fragment of the ABS (Abstract Behavioral Specification) language core [23], developed in the EU FP7 HATS project. We show that the conclusions of our previous work remain valid, but with more involved constructions. The extensions result in much more complex object overlays involving futures [6, 12, 14, 27, 28, 39], in effect placeholders for remote method return values, that can be shared among objects, but whose eventual instantiated values need to be kept consistent and propagated correctly to all objects that need them. Future variables are used extensively in many concurrent and distributed high-level languages, libraries, and models, including Java, .NET, Scheme, Concurrent LISP, and Oz, to name just a few. Many versions of futures exist in the literature. Our work uses futures as placeholders for forthcoming computational results, as do Caromel et al. [5] and de Boer et al. [12]. Other models exists, such as the concurrent constraint store model of, e.g., Oz [28, 37].

Futures need a messaging infrastructure to propagate instantiations. Consider a remote method call `x = obj!m(args)`. The effect of the call is the creation of two items:

1. A remote thread evaluating method `m` with the arguments `args` in `obj`.
2. A future that becomes assigned to the variable `x`. The future is initially uninstantiated, but is intended to become instantiated after the remote call has returned.

This functionality allows long-running tasks to be offloaded to a remote

thread with the main thread proceeding with other tasks. When the return value is eventually needed, the calling thread can request it by performing a **get** operation on the future. If `x` is uninstantiated, this causes the evaluation to block.

One problem is that first-class futures [5], which we employ, can be transmitted as arguments between threads. If `y` is a future variable occurring in `args`, there must be some means for the value eventually assigned to `y` to find its way to the remote thread computing `obj!m(args)`, either by forwarding the value after it becomes available, or by the remote thread querying either the caller or some centralized lookup server for the value of `y`, if and when it is needed. This creates very similar problems to those arising from object migration. Thus, it would seem likely that location independent routing could be useful for propagation of values for futures, and as we show in this paper, indeed this is so. In the case of futures, however, the problems are aggravated: In order for the network-aware implementation to be correct (sound and fully abstract) we must be able to show that future assignments are unique and propagate correctly to all objects needing the assignment, without resorting to solutions that are overly inefficient such as flooding.

Many strategies for future propagation exist in the literature [18, 31]. In this work, we use what Henrio et al. [18] refer to as an eager forward-based strategy, where assignments are propagated along the flow of futures as soon as they are instantiated. Other propagation strategies exist, including strategies that use various forms of location registers, and lazy strategies which request futures only as needed. Either approach may benefit from the use of location independent routing. However, our chosen strategy is particularly suited for decentralized networks, since it has lower propensity of overloading any particular node when object-node allocations are balanced [18].

Our main result is to show that, with full nondeterminism, the abstract, network-oblivious semantics and the network-aware semantics with futures implemented through eager forwarding correspond in the sense of contextual equivalence. To the best of our knowledge, this is the first such result in the literature, and is interesting in itself, as it shows that the network-aware semantics captures the abstract behavior very accurately. Also, it follows that, for the case when a scheduler is added (pruning some execution branches), a similar correspondence holds, but now for barbed simulation instead of barbed bisimulation.

The proof of the main result uses a normal form construction in two stages. First, we show that each well-formed configuration in the network-aware semantics can be rewritten into an equivalent form with optimal routes. The second stage of the normalization procedure then continues rewriting to a form where, in addition, all messages that can be delivered also are delivered, and where all objects are migrated to some central node. Correctness of the normalization procedure essentially gives a Church-Rosser like property—that transitions in the network-aware semantics commute with normalization. Normalization brings configurations in

the network-aware semantics close to the form of the reference semantics, and this, then, allows the proof to be completed.

The paper is organized as follows: In Section 3, we first introduce the mABS language syntax, and the network-oblivious reference (type 1) semantics of mABS is given in Section 4. In Section 5, we present type 1 contextual equivalence, i.e., the notion of contextual equivalence adapted to the reference semantics. Then, in Section 6, we turn to the network-aware (type 2) semantics and present the runtime syntax and the reduction rules. We proceed by detailing the well-formedness conditions for the network-aware semantics in Section 7 and adapt contextual equivalence to this semantics in Section 8. We then present the normal-form construction in Section 9, and complete the correctness proof in Section 10. In Section 11, we discuss scheduling, and finally in Section 12, we conclude. Long proofs have been deferred to appendices.

2 Notation

We sometimes use a vectorized notation to abbreviate sequences, for compactness. Thus, $\overline{x}$ abbreviates a sequence $x_1, \ldots, x_n$, possibly empty, and $x_0, \overline{x}$ abbreviates $x_0, \ldots, x_n$. Let $g : A \to B$ be a finite map. The update operation for g is $g[b/a](x) = g(x)$ if $x \neq a$ and $g[b/a](a) = b$. We use $\bot$ for bottom elements, and $A_\bot$ for the lifted set with partial order $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if either $a = b \in A$ or else $a = \bot$. Also, if x is a variable ranging over A, we often use $x_\bot$ as a variable ranging over $A_\bot$. For g a function $g : A \to B_\bot$, we write $g(a)\downarrow$ if $g(a) \in B$, and $g(a)\uparrow$ if $g(a) = \bot$. The product of sets (flat CPOs) A and B is $A \times B$ with pairing (a, b) and projections π_1 and π_2.

3 The mABS Language

We define mABS, short for milli-ABS, a small, distributed, object-based language with asynchronous calls and futures. Its syntax is depicted in Figure 1. The mABS language is an extension of the language μABS (micro-ABS) of message-passing processes introduced in earlier work [11] with futures used as placeholders for method return values. The language is fairly self-explanatory. A program is a sequence of class definitions, appended with a set of variables $\overline{x}$ and a "main" statement s, which can use those variables to set up an initial collection of objects. The class hierarchy is flat and fixed. Classes have parameters $\overline{x}$, local variable declarations $\overline{y}$, and methods $\overline{M}$. Methods have parameters $\overline{x}$, local variable declarations $\overline{y}$ and a statement body s. For simplicity, we assume that variables have unique declarations. Expression syntax is left open, but is assumed to include the constant **self**. We require that expressions are side-effect free. We omit types from the presentation. Types could be added, but they would not affect the results of the paper in any significant way. Statements include standard sequential control structures, and a minimal set of constructs for

$x, y \in Var$			Variable
$e \in Exp$			Expression
$C, m \in SID$			Static identifier
P	::=	$\overline{CL}\,\{\overline{x}, s\}$	Program
CL	::=	**class** $C(\overline{x})\,\{\overline{y}, \overline{M}\}$	Class definition
M	::=	$m(\overline{x})\,\{\overline{y}, s\}$	Method definition
s	::=	$s_1; s_2 \mid x = rhs \mid$ **skip** $\mid$ **while** $e\,\{s\}$	Statement
		$\mid$ **if** $e\,\{s_1\}$ **else** $\{s_2\} \mid$ **return** e	
rhs	::=	$e \mid$ **new** $C(\overline{e}) \mid e!m(\overline{e}) \mid e.$**get**	Right-hand side

Figure 1: mABS abstract syntax

```
class Server() { ,
  serve(x) { s1, s2, f1, f2, r1, r2,
    if small(x) {
      return process(x)
    } else {
      s1 = new Server();       s2 = new Server();
      f1 = s1!serve(hi(x));    f2 = s2!serve(lo(x));
      r1 = f1.get;             r2 = f2.get;
      return combine(r1, r2)
    }
  }
}
{
  s, f, r,
  s = new Server(); f = s!serve(1537); r = f.get
}
```

Figure 2: mABS code sample

asynchronous method invocation, object creation, and retrieval of values associated with futures (**get** statements).

Example 3.1. Assume that `combine(hi(`v`),lo(`v`)) = process(`v`)` for integers v. In the class `Server` in the program in Figure 2, the method `serve` returns immediately if its argument is small. Otherwise, two new servers are spawned, and the upper and lower tranches delegated to those respective servers. The results are then retrieved, combined, and returned. In the main block, a call to `serve` on a server object results in a future, stored in the variable `f`, which is then used to retrieve the actual result, stored in the variable `r`. The original call spawns more server objects, which, in a network-aware implementation, can move to other nodes to balance load.

4 Reference Semantics

We first present an abstract reference semantics for mABS in the style of rewriting logic. The presentation follows our earlier work [11] quite closely. We use the abstract semantics for comparison with the more con-

crete network-aware semantics, which we present later. The semantics uses a reduction relation $cn \rightarrow cn'$ where cn and cn' are *configurations*, as determined by the runtime syntax in Figure 3. Later on, we introduce different configurations and transition relations, and so use index 1, or refer to, e.g., configurations of "type 1" for this first semantics when we need to disambiguate. With respect to the runtime syntax, $\preceq$ is the sub-

$x \in Var$			Variable
$o \in OID$			Object identifier
$p \in PVal$			Primitive value
$f \in FID$			Future identifier
$v \in Val$	$=$	$PVal \cup OID \cup FID$	Value
$z \in Name$	$=$	$OID \cup FID$	Name
$l \in MEnv$	$=$	$Var \cup \{\mathbf{ret}\} \rightarrow Val_\perp$	Task environment
$a \in OEnv$	$=$	$Var \cup \{\mathbf{self}\} \rightarrow Val_\perp$	Object environment
$tsk \in Tsk$	$::=$	$\mathsf{t}(o, l, s)$	Task
$obj \in Obj$	$::=$	$\mathsf{o}(o, a)$	Object
$fut \in Fut$	$::=$	$\mathsf{f}(f, v_\perp)$	Future
$call \in Call$	$::=$	$\mathsf{c}(o, f, m, \overline{v})$	Call
$ct \in Ct$	$::=$	$tsk \mid obj \mid call \mid fut$	Container
$cn \in Cn$	$::=$	$0 \mid ct \mid cn\ cn' \mid \mathsf{bind}\ z.cn$	Configuration
$obs \in Obs$	$::=$	$ext!m(\overline{v})$	Observation

Figure 3: mABS type 1 runtime syntax

term relation, and we use disjoint, denumerable sets of object identifiers $o \in OID$, future identifiers $f \in FID$, and primitive values $p \in PVal$. Values v are either primitive values, OIDs, or FIDs. Lifted values are ranged over by $v_\perp \in Val_\perp$, and we use $\sqsubseteq$ for the associated standard partial ordering. We often refer to OIDs and FIDs as *names*, and subject them binding using bind, which is reminiscent of the binder in the π-calculus [32]. Later, in the type 2 semantics, this type of explicit binding is dropped. We use z as a generic name variable, and assume throughout that names are uniquely bound. The free names of a configuration cn is the set $\mathtt{fn}(cn)$, and $OID(cn) = \{o \mid \exists a.\, \mathsf{o}(o, a) \preceq cn\}$ is the set of OIDs of objects occurring in cn. Similarly, $FID(cn) = \{f \mid \exists v_\perp.\, \mathsf{f}(f, v_\perp) \preceq cn\}$ is the set of future identifiers in cn. Standard alpha congruence applies to name binding.

Configurations are "π-scoped" multisets of containers of which there are four types, namely, tasks, objects, futures, and calls. Configuration juxtaposition is assumed to be commutative and associative with unit 0. In addition we assume the standard structural identities $\mathsf{bind}\ z.0 = 0$ and $\mathsf{bind}\ z.(cn_1\ cn_2) = (\mathsf{bind}\ z.cn_1)\ cn_2$ when $z \notin \mathtt{fn}(cn_2)$. We often use a vectorized notation $\mathsf{bind}\ \overline{z}.cn$ as abbreviation, letting $\mathsf{bind}\ \varepsilon.cn = cn$ where ε is the empty sequence. The structural identities then allow us to rewrite each configuration into a *standard form* $\mathsf{bind}\ \overline{z}.cn$ such that each member of $\overline{z}$ occurs free in cn, and cn has no occurrences of the binding operator bind. We use standard forms frequently.

Tasks are used for method body elaboration, and futures are used as centralized stores for assignments to future variables. Task and object envi-

ronments l and a, respectively, map task and object variables to values. Task environments are aware of a special variable **ret** that a task can use in order to identify its return future. Upon method invocation, a task environment is initialized using the operation $\texttt{locals}(o, f, m, \overline{v})$ which maps the formal parameters of method m in the class of o to the corresponding arguments in $\overline{v}$, initializes the method local variables to suitable null values, and maps **ret** to f, the return future of the task being created. Object environments are initialized using the operation $\texttt{init}(C, \overline{v}, o)$, which maps the parameters of the class C to $\overline{v}$, the special variable **self** to o, and initializes the object variables as above. In addition to `locals` and `init`, the reduction rules presented below use the auxiliary operation $\texttt{body}(o, m)$, which retrieves the statement of the shape s in the definition body for m in the class of o, and $[\![e]\!]_{(a,l)} \in \mathit{Val}$ is used for evaluating the expression e in object environment a and task environment l.

Calls play a special role in defining the external observations of a configuration cn. Assume an OID ext representing the "outside world", not allowed to be bound or defined in any well-formed configuration. An observation, or *barb*, is a call of the form $ext!m(\overline{v})$, ranged over by obs. Calls that are not external are meant to be completed in the usual reduction semantics style, by internal reaction with the called object, spawning a new task. External calls could be represented directly, without relying on the call container type, by saying that a configuration cn has the barb $obs = ext!m(\overline{v})$ whenever cn has the shape

$$\mathsf{bind}\ \overline{z}.\big(cn'\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e_1!m(\overline{e_2}); s)\big)\ , \tag{1}$$

where $[\![e_1]\!]_{(a,l)} = ext$ and $[\![\overline{e_2}]\!]_{(a,l)} = \overline{v}$. However, in a semantics with unordered communication, which is what we are after, consecutive calls should commute, i.e., there should be no observational distinction between executing two method calls with the respective statements

$$x = e_1!m_1(\overline{e_1}'); y = e_2!m_2(\overline{e_2}'); s$$

and

$$y = e_2!m_2(\overline{e_2}'); x = e_1!m_1(\overline{e_1}'); s$$

This, however, is difficult to reconcile with the representation in (1). To this end, call containers are used for both internal and external calls, allowing configurations like (1) to produce a corresponding container, and then proceed to elaborate s.

We next present the reduction rules. For ease of notation, the rules assume that sequential statement composition is associative with unit **skip**. The rules in Figure 4 and Figure 5 define the reduction relation. The rules use the notation $cn \vdash cn' \rightarrow cn''$ as shorthand for $cn\ cn' \rightarrow cn\ cn''$. Figure 4 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and Figure 5 gives the more interesting rules that involve method invocation and object creation. A method call causes a new future identifier to be created, along with its future container, with lifted value initialized to $\bot$. Future instantiation is done when **return** statements

CTXT-1: If $cn_1 \to cn_2$, then $cn \vdash cn_1 \to cn_2$

CTXT-2: If $cn_1 \to cn_2$, then $\mathsf{bind}\ z.cn_1 \to \mathsf{bind}\ z.cn_2$

WLOCAL: If $x \in \mathtt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in $\mathsf{t}(o, l, x = e; s) \to \mathsf{t}(o, l[v/x], s)$

WFIELD: If $x \in \mathtt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[v/x])\ \mathsf{t}(o, l, s)$

SKIP: $\mathsf{t}(o, l, \mathbf{skip}; s) \to \mathsf{t}(o, l, s)$

IF-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\,\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_1; s)$

IF-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{if}\ e\,\{s_1\}\ \mathbf{else}\ \{s_2\}; s) \to \mathsf{t}(o, l, s_2; s)$

WHILE-TRUE: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\,\{s_1\}; s) \to \mathsf{t}(o, l, s_1; \mathbf{while}\ e\,\{s_1\}; s)$

WHILE-FALSE: If $[\![e]\!]_{(a,l)} = 0$, then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\,\{s_1\}; s) \to \mathsf{t}(o, l, s)$

Figure 4: mABS type 1 reduction rules, part 1

CALL-SEND: Let $o' = [\![e_1]\!]_{(a,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$ in
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \to \mathsf{bind}\ f.\mathsf{t}(o, l[f/x], s)\ \mathsf{f}(f, \bot)\ \mathsf{c}(o', f, m, \overline{v})$

CALL-RCV: Let $l = \mathtt{locals}(o, f, m, \overline{v})$, $s = \mathtt{body}(o, m)$ in $\mathsf{o}(o, a) \vdash \mathsf{c}(o, f, m, \overline{v}) \to \mathsf{t}(o, l, s)$

RET: Let $f = l(\mathbf{ret})$, $v = [\![e]\!]_{(a,l)}$ in $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{return}\ e; s)\ \mathsf{f}(f, \bot) \to \mathsf{f}(f, v)$

GET: Let $f = [\![e]\!]_{(a,l)}$ in $\mathsf{o}(o, a)\ \mathsf{f}(f, v) \vdash \mathsf{t}(o, l, x = e.\mathbf{get}; s) \to \mathsf{t}(o, l[v/x], s)$

NEW: Let $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{init}(C, \overline{v}, o')$ in
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \to \mathsf{bind}\ o'.\mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a')$

Figure 5: mABS type 1 reduction rules, part 2

are evaluated, and **get** statements cause the evaluating task to hang until the value associated with the future is defined, and then store that value. Object creation (**new**) statements cause new objects to be created along with their OIDs in the expected manner.

We note some basic properties of the reduction semantics.

Proposition 4.1. *Suppose $cn \to cn'$. Then, the following holds:*

1. $\mathtt{fn}(cn') \subseteq \mathtt{fn}(cn)$.
2. *If $\mathsf{o}(o, a) \preceq cn$, then $\mathsf{o}(o, a') \preceq cn'$ for some object environment a'.*
3. *If $\mathsf{f}(f, v_\bot) \preceq cn$, then $\mathsf{f}(f, v'_\bot) \preceq cn'$ for some $v'_\bot$ such that $v_\bot \sqsubseteq v'_\bot$.*

Proof. No structural identity, nor any reduction rule, allows an OID or FID to escape its binder. No rules allow object or future containers to be removed. Also, no rules allow futures to be re-instantiated to $\bot$. The results follow. □

Definition 4.2 (Type 1 Initial Configuration, Type 1 Reachable)**.** Consider a program $\overline{CL}\,\{\overline{x}, s\}$. The program can make calls to a special OID ext, and in this way produce externally observable output. Assume a reserved OID

o_{main} distinct from ext, and a reserved FID f_{init}. A *type 1 initial configuration* for the program has the shape

$$cn_{init} = \mathsf{bind}\ o_{main}, f_{init}.\mathsf{o}(o_{main}, \bot)\ \mathsf{t}(o_{main}, l_{init}, s)\ \mathsf{f}(f_{init}, \bot)\ ,$$

where l_{init} is the initial task environment assigning suitable default values to the variables in $\overline{x}$, and $l_{init}(\mathbf{ret}) = f_{init}$. When there is a derivation $cn_1 \to \cdots \to cn_n$, we say that cn_n is *reachable* from cn_1. If $cn_1 = cn_{init}$, cn_n is said to be *type 1 reachable*.

Definition 4.3 (Type 1 Active Future)**.** Let cn be a type 1 configuration. The future identifier f is *active* for the object o in cn if one of the following holds:

1. There is an object container $\mathsf{o}(o, a) \preceq cn$ such that $a(x) = f$ for some x.
2. There is a task container $\mathsf{t}(o, l, s) \preceq cn$ such that $l(x) = f$ for some x.
3. There is a call container $\mathsf{c}(o, f', m, \overline{v}) \preceq cn$, and $f' = f$ or f occurs in $\overline{v}$.
4. There is a future identifier f' that is active for o in cn, and $\mathsf{f}(f', f) \preceq cn$.

Definition 4.4 (Type 1 Well-formedness)**.** A configuration cn is *type 1 well-formed* (WF1) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1, a_1)$ and $\mathsf{o}(o_2, a_2)$ are distinct object container occurrences in cn, then $o_1 \neq o_2$.
2. *Task-Object Existence*: If $\mathsf{t}(o, l, s) \preceq cn$, then $\mathsf{o}(o, a) \preceq cn$ for some object environment a.
3. *Call Uniqueness*: If $\mathsf{c}(o_1, f_1, m_1, \overline{v}_1)$ and $\mathsf{c}(o_2, f_2, m_2, \overline{v}_2)$ are distinct call container occurrences in cn, then $f_1 \neq f_2$.
4. *Future Uniqueness*: If $\mathsf{f}(f_1, v_{\bot,1})$ and $\mathsf{f}(f_2, v_{\bot,2})$ are distinct future container occurrences in cn, then $f_1 \neq f_2$.
5. *Single Writer*: If $\mathsf{t}(o_1, l_1, s_1)$ and $\mathsf{t}(o_2, l_2, s_2) \preceq cn$ are distinct task container occurrences in cn such that $l_1(\mathbf{ret}) = f_1$ and $l_2(\mathbf{ret}) = f_2$, then $f_1 \neq f_2$, $\mathsf{f}(f_1, \bot) \preceq cn$, and $\mathsf{f}(f_2, \bot) \preceq cn$, and additionally, if $\mathsf{c}(o, f, m, \overline{v}) \preceq cn$, then $f \neq f_1$ and $f \neq f_2$.
6. *Future Existence*: If f is active for o in cn or $\mathsf{f}(f', f) \preceq cn$, then $\mathsf{f}(f, v_\bot) \preceq cn$; if $\mathsf{f}(f, \bot) \preceq cn$, then there is either a call container $\mathsf{c}(o', f, m, \overline{v}) \preceq cn$ with $o' \in OID(cn)$, or a task container $\mathsf{t}(o', l, s) \preceq cn$ such that $l(\mathbf{ret}) = f$.

Well-formedness is important, as it ensures that objects and futures, if defined, are defined uniquely, and that, e.g., tasks are defined only along with their accompanying object. The Single Writer property reflects the fact that only the task that was spawned along with some given future is able to assign to that future, and hence, if the task has not yet returned, the

future remains uninstantiated. Future Existence ensures that there are corresponding future containers for FIDs accessible to objects, and that those containers either carry values or have the potential of carrying a value.

Proposition 4.5 (WF1 Preservation)**.** *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 1 initial configuration, then cn is WF1.*
2. *If cn is WF1 and $cn \to cn'$, then cn' is WF1.*
3. *If cn is type 1 reachable, then cn is WF1.*

Proof. The first two properties hold by inspection of the definitions and the rules. The last property holds by way of the first property and repeated application of the second property. □

5 Type 1 Contextual Equivalence

Our approach to implementation correctness uses contextual equivalence [30]. The goal is to show that it is possible in a network-aware setting to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to a separate scheduler. This allows drawing strong conclusions also in the case where a scheduler is added, as we discuss in Section 11. Contextual equivalence requires of a pair of equivalent configurations, firstly, that the internal transition relation $\to$ is preserved in both directions, and secondly, that the relation is preserved when adding a context configuration, all while preserving a set of external observations. A number of works [22, 33] have established very strong relations between contextual equivalence for reduction oriented semantics and bisimulation/logical relation based equivalences for sequential and higher-order computational models.

Let $obs = ext!m(\overline{v})$. The observation predicate $cn \downarrow obs$ is defined to hold just in case cn can be written in the form

$$\mathsf{bind}\ \overline{z}.\big(cn'\ \mathsf{c}(ext, f, m, \overline{v})\big)\ .$$

The derived predicate $cn \Downarrow obs$ holds just in case $cn \to^* cn' \downarrow obs$ for some configuration cn'.

Definition 5.1 (Type 1 Witness Relation, Type 1 Contextual Equivalence)**.** Let $\mathcal{R}$ range over binary relations on WF1 configurations. The relation $\mathcal{R}$ is a *type 1 witness relation*, if $cn_1\ \mathcal{R}\ cn_2$ implies

1. *Reduction Closure*: If $cn_1 \to cn_1'$, then $cn_2 \to^* cn_2'$ for some cn_2' such that $cn_1'\ \mathcal{R}\ cn_2'$.
2. *Context Closure*: If $cn_1\ cn$ is WF1, then $cn_2\ cn$ is WF1 and $cn_1\ cn\ \mathcal{R}\ cn_2\ cn$.
3. *Barb Preservation*: If $cn_1 \downarrow obs$, then $cn_2 \Downarrow obs$.

Additionally, the converse properties must hold with $\mathcal{R}^{-1}$ for $\mathcal{R}$ above[1]. We define *type 1 contextual equivalence*, $\simeq_1$, as the union of all type 1 witness relations. Additionally, we say that the WF1 configurations cn_1 and cn_2 are *type 1 contextually equivalent* whenever $cn_1 \simeq_1 cn_2$, i.e., whenever $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation $\mathcal{R}$.

We establish some well-known, elementary properties of contextual equivalence for later reference.

Proposition 5.2. *The identity relation is a type 1 witness relation. $\simeq_1$ is a type 1 witness relation. If $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$ are type 1 witness relations then so is*

1. $\mathcal{R}^{-1}$,
2. $\mathcal{R}^*$, *and*
3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$.

Proof. See Appendix 1. □

We conclude that $\simeq_1$ has the expected basic property.

Proposition 5.3. $\simeq_1$ *is an equivalence relation.*

Proof. The result follows from Proposition 5.2. For transitivity, in particular, we use Proposition 5.2.3. □

6 Network-Aware Semantics

We now turn to the second, main part of the paper where we address the problem of efficiently executing mABS programs on an abstract network graph using the location independent routing scheme alluded to in Section 1. The approach follows closely that for the network-aware semantics introduced in earlier work [11], with the important difference that method return values via futures are now included. In addition to the naming, routing, and object migration issues already addressed previously, the additional challenge is to ensure that futures are correctly assigned and propagated at the network level.

In the network-aware semantics, we assume an explicitly given network "underlay": A network of nodes and directional links with which message buffers are associated, modeling a concrete network structure with asynchronous point-to-point message passing. Object execution is localized to each node. At the outset, nodes know only of their "own" objects, but as routing information is propagated, inter-node object-to-object message delivery becomes possible. Objects can migrate between neighboring nodes. When this is done is not addressed here; we discuss possible decentralized adaptation strategies, that in effect impose a scheduler, in other work [29]. The propagation of routing information will automatically lead to routing tables becoming up-to-date. How and when this is done is again left to a

[1]The usual explicit symmetry requirement is slightly too strong for our purpose.

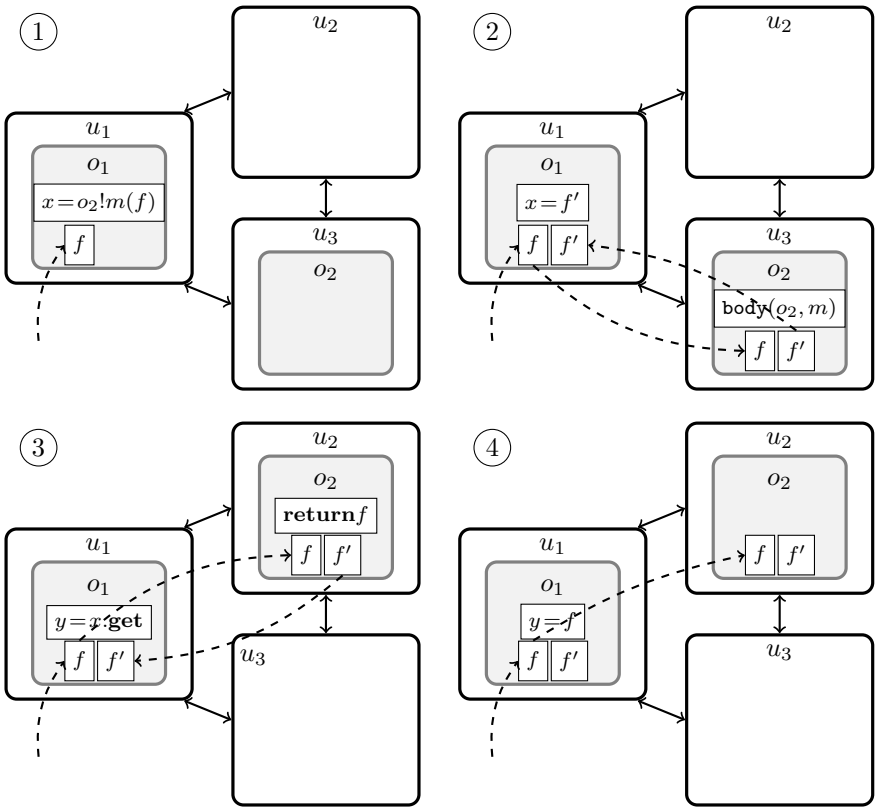

Figure 6: Futures in the mABS network-aware semantics

scheduler. Method calls to an object can be issued if a task has access to the OID of the object, and the associated message can be delivered once a route to the callee becomes known.

For the mABS language, the network-aware semantics must be extended to cover also return values and futures. We use an eager forward-based strategy [5, 18] for handling futures. The strategy's central idea is that, whenever an object shares a future identifier, the object assumes an obligation to send the associated value to the object with which the future identifier is shared. Initially, the value may be unavailable, requiring the use of forwarding lists for futures stored in the object state. Our objective is to prove that this approach is sound and fully abstract for our network-aware semantics, even though routing may be in an unstable state.

Example 6.1. An example of an execution, which illustrates future propagation and its interaction with routing, is shown in Figure 6. In configuration 1 in the upper left corner, a call to method m with argument f, a future identifier, is about to be sent from object o_1 residing on node u_1 to object o_2 on node u_3. The following events then take place as the system evolves into configuration 2 in the upper right corner:

- A new future f' is created at object o_1, as a placeholder for the return value of the call to method m.
- The forwarding list for f at o_1 is augmented to include o_2.
- The call is routed to o_2, and a new task computing the statement $\mathsf{body}(o_2, m)$ is spawned at o_2, associated with f'.
- o_2 is augmented with f and f' as placeholders, and the forwarding list for f' at o_2 is augmented to include o_1.

The scheduler now decides to migrate o_2 from u_3 to u_2. No action is required other than regular routing table updates, as the forward pointers keep referencing the same objects. Finally, the future identifier f is returned by the task at o_2, as shown in the lower left configuration 3. In the lower right configuration 4, the value of the future f', namely, f, has been sent to o_1, where it becomes assigned to the variable y. The obligation of o_1 to forward the value of f to o_2 has therefore become redundant and could be removed without affecting object behavior, but for simplicity we do not include such garbage collection in the semantics. The price for this omission is additional messaging, but the latency remains the same, since o_1 is able to discover an assignment to f at o_1 when it is first made.

6.1 Runtime Syntax

In Figure 7, we present the network-aware mABS runtime syntax, i.e., the shape of the runtime state. Recall from Section 4 that we reuse symbols

$u \in NID$			Node identifier
$a \in OEnv_2$	$=$	$(Var \cup \{\textbf{self}\} \to Val_\perp) \times (FID \to (Val_\perp \times (OID\ \text{list}))_\perp)$	Object environment
$t \in RTable$	$=$	$OID \to (NID \times \omega)_\perp$	Routing table
$q \in Q$	$=$	Msg^*	Message queue
$obj \in Obj_2$	$::=$	$\mathsf{o}(o, a, u, q_{in}, q_{out})$	Object
$nd \in Nd$	$::=$	$\mathsf{n}(u, t)$	Network node
$lnk \in Lnk$	$::=$	$\mathsf{l}(u, q, u')$	Network link
$ct \in Ct_2$	$::=$	$tsk \mid obj \mid nd \mid lnk$	Container
$cn \in Cn_2$	$::=$	$ct_1 \dots ct_n$	Configuration
$msg \in Msg$	$::=$	$\mathsf{call}(o, o', f, m, \overline{v}) \mid \mathsf{future}(o, f, v) \mid \mathsf{table}(t) \mid \mathsf{object}(cn)$	Message

Figure 7: mABS type 2 runtime syntax

as much as possible and use indices to disambiguate. Thus, for instance, Obj_1 is the set Obj of the type 1 semantics in Figure 3, and Obj_2 is the corresponding set in Figure 7. We adopt the same syntactical conventions as in Section 4. Tasks are unchanged from Figure 3. We write $\mathsf{t}(cn)$ for the multiset of tasks in cn, i.e., the multiset $\{tsk \mid tsk \preceq cn\}$, and $\mathsf{o}(cn)$ for

the multiset of objects in cn, similarly defined. We also write $\mathtt{m}(cn)$ for the multiset $\{msg \mid msg \preceq cn\}$.

We proceed to explain the different types of containers and the operations on them, concentrating on the treatment of futures. For a more detailed explanation of other features, in particular routing, we refer to our earlier work [11].

Network and Routing The nodes and links in a configuration cn induce a network graph $\mathtt{graph}(cn)$, which contains a vertex u for each node container $\mathtt{n}(u,t)$ and an edge (u,u') for each link $\mathtt{l}(u,q,u')$. The reduction semantics given later does not allow identifiers in nodes or links to be changed, so in the context of any given transition (or, execution), the network graph remains constant. Note that there is no a priori guarantee that the network graph is a well-formed graph. For the remainder of the paper, we therefore impose some constraints on the well-formedness of the network graph, namely, that (i) there is at least one vertex, (ii) endpoints of edges exist, (iii) vertices and edges are uniquely determined, (iv) the network graph is reflexive and symmetric, and (v) the network graph is connected. For routing, we adopt a simple Bellman-Ford distance vector discipline [38]. For a routing table t, $t(o) = (u,n)$ indicates that, as far as t is concerned, there is a path from the current node (the node to which t is attached) to the object o with distance n, that first visits the node u. We only count hops to compute distance, for simplicity. A more realistic routing scheme associates *weights* with edges, reflecting latency or capacity constraints. Next hop lookup is performed by the operation $\mathtt{nxt}(o,t) = \pi_1(t(o))$ where π_1 is the first projection. There is also an operation $\mathtt{upd}$ for updating a routing table t by a routing table t' received from a neighboring node u, defined by

$$\mathtt{upd}(t,u,t')(o) = \begin{cases} \bot & \text{if } o \notin \mathtt{dom}(t) \cup \mathtt{dom}(t') \\ t(o) & \text{else, if } o \notin \mathtt{dom}(t') \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } o \notin \mathtt{dom}(t) \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_1(t'(o)) = u \\ (u, \pi_2(t'(o)) + 1) & \text{else, if } \pi_2(t'(o)) + 1 < \pi_2(t(o)) \\ t(o) & \text{otherwise.} \end{cases}$$

Finally, there is an operation $\mathtt{reg}(o,u,t,n)$ that returns the routing table t', obtained by registering the object identifier o at t's current node u with distance n, i.e., such that

$$\mathtt{reg}(o,u,t,n)(o') = \begin{cases} (u,n) & \text{if } o = o' \\ t(o') & \text{otherwise.} \end{cases}$$

Message Queues FIFO message queue operations are standard: $\mathtt{enq}(msg,q)$ enqueues a message msg onto the tail of the queue q, $\mathtt{hd}(q)$ returns the head of q, and $\mathtt{deq}(q)$ returns the tail of the q, i.e., q with $\mathtt{hd}(q)$ removed. If q is empty, then $\mathtt{hd}(q) = \mathtt{deq}(q) = \bot$.

Messages The network-aware semantics uses four forms of messages:

- $\mathsf{call}(o, o', f, m, \overline{v})$: A call message, corresponding to a call container in the reference semantics, where o is the identifier of the callee and o' the identifier of the caller, respectively.
- $\mathsf{future}(o, f, v)$: A future instantiation message, meant to inform object o that the future f has been instantiated to the value v.
- $\mathsf{table}(t)$: A routing table update message, with the routing table t.
- $\mathsf{object}(cn)$: A message containing an *object closure* cn of the form $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l_1, s_1)\ \ldots\ \mathsf{t}(o, l_n, s_n)$, as explained in more detail below, used for migrating objects and their tasks across nodes. Note that containers inside object closure messages in a link queue q are included in the subterm relation $\preceq$ for the configuration in which the link with q resides.

Call and future instantiation messages are said to be *object bound*, while routing table messages and object messages are *node bound*. We define $\mathtt{dst}(msg)$, the *destination* of msg, to be o for a call message or a future message as defined above, and $\perp$ in the remaining two cases.

Objects and Object Environments Object containers $\mathsf{o}(o, a, u, q_{in}, q_{out})$ are now attached to a node u and equipped with an ingoing (q_{in}) and an outgoing (q_{out}) FIFO message queue, and the notion of object environment is refined to take futures into account in a localized manner. In the type 2 semantics, object environments a are now augmented by mapping futures f to pairs $(v_\perp, \overline{o})$, where:

- $v_\perp$ is the lifted value currently associated with f at the current object, and
- $\overline{o}$ is a *forwarding list*, containing the identifiers of the objects subscribing to instantiations of f at the current object.

For instance, if $a(f) = (\perp, o_1\, o_2)$, the future f is as yet uninstantiated (at the object to which a belongs), and, if f eventually does become instantiated, the instantiation must be forwarded to o_1 and o_2. Forwarding does not necessarily happen in the given order, since we consider forwarding lists modulo associativity and commutativity with the empty list ε as unit.

We introduce some notation and auxiliary operations to help manipulating object environments:

- $a(x)$ abbreviates $\pi_1(a)(x)$, and $a(f)$ abbreviates $\pi_2(a)(f)$.
- $a[v/x]$ is a with $\pi_1(a)$ replaced by the expected update. Similarly, $a[v/f]$ updates $\pi_2(a)$ by mapping f to the pair $(v, \pi_2(a(f)))$, i.e., the assigned value is updated and the forwarding list remains unchanged. If $f \notin \mathtt{dom}(\pi_2(a))$, then $a[v/f](f) = (v, \varepsilon)$, i.e., the update to the value takes effect. Finally, we use $a[(v, \overline{o})/f]$ for the expected update where both the value and the forwarding list are updated.

- $\texttt{fw}(\overline{v}, o, a)$ updates $\pi_2(a)$ by, for each future f occurring in $\overline{v}$, adding o to the forwarding list of $a(f)$, i.e., by mapping f to either $(\bot, o)$ if $a(f) \uparrow$, or $(\pi_1(a(f)), o\, \pi_2(a(f)))$ otherwise.
- $\texttt{init}(C, \overline{v}, o)$ returns an initial object environment, by mapping the formal parameters of C to $\overline{v}$, and **self** to o; this operation is unchanged from the reference semantics.
- $\texttt{init}(\overline{v}, a)$ augments a by mapping each FID f in $\overline{v}$ which is uninitialized in a (i.e., such that $f \notin \texttt{dom}(a)$) to $(\bot, \varepsilon)$.

As a consequence of these changes, futures are eliminated as containers in the type 2 runtime syntax. In other respects, the type 2 runtime syntax is unchanged: Syntactical conventions that are not explicitly modified in the type 2 syntax above remain the same. In particular, we continue to assume the commutativity and associativity properties of configuration juxtaposition, now with the empty list of containers as unit.

Object Closures Object closures wrap objects with their active tasks. We use the following operations:

- $\texttt{clo}(cn, o)$, the *closure* of object o with respect to the configuration cn, is the multiset of all type 2 containers in cn of the form $\texttt{o}(o', a', u', q'_{in}, q'_{out})$ or $\texttt{t}(o', l', s')$, such that $o' = o$.
- $\texttt{oidof}(cn)$, a partial function returning the OID o, if all the type 2 containers in cn are objects and tasks with OID o.
- $\texttt{place}(cn, u)$ places all object containers in the configuration cn at the node u, i.e., cn and $\texttt{place}(cn, u)$ are identical, except that each object container $\texttt{o}(o', a', u', q'_{in}, q'_{out})$ in cn is replaced by an object container $\texttt{o}(o', a', u, q'_{in}, q'_{out})$ in $\texttt{place}(cn, u)$.

6.2 Reduction Semantics

An important distinction between the reference semantics and the network-aware semantics is the absence of binding. For the standard semantics, name binding plays an important role in avoiding clashes between locally generated names. However, in a language with node identifiers (NIDs) this device is no longer needed, as globally unique name can be guaranteed easily by augmenting names with their generating NID. Since all name generation in the mABS type 2 semantics below takes place in the context of a given NID, we can simply assume the existence of two operations $\texttt{newf}(u)$ and $\texttt{newo}(u)$, that return a new future and a new OID, respectively, that is globally fresh for the "current context", with $\texttt{newo}(u)$ distinct from ext.

We now present the mABS type 2 reduction rules. The first part, in Figure 8, is carried over from the type 1 semantics in Figure 4, with some minor modifications. First, rule CTXT-2 is dropped, since name binding is dropped from the type 2 runtime syntax. Second, the rules WFIELD, IF-TRUE, IF-FALSE, WHILE-TRUE, and WHILE-FALSE are straightforwardly modified to account for

CTXT-1: If $cn_1 \to cn_2$, then $cn \vdash cn_1 \to cn_2$

WLOCAL: If $x \in \texttt{dom}(l)$, then let $v = [\![e]\!]_{(a,l)}$ in $\texttt{t}(o, l, x = e; s) \to \texttt{t}(o, l[v/x], s)$

WFIELD-2: If $x \in \texttt{dom}(a)$, then let $v = [\![e]\!]_{(a,l)}$ in
$\texttt{o}(o, a, u, q_{in}, q_{out})\ \texttt{t}(o, l, x = e; s) \to \texttt{o}(o, a[v/x], u, q_{in}, q_{out})\ \texttt{t}(o, l, s)$

SKIP: $\texttt{t}(o, l, \textbf{skip}; s) \to \texttt{t}(o, l, s)$

IF-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\texttt{o}(o, a, u, q_{in}, q_{out}) \vdash \texttt{t}(o, l, \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s) \to \texttt{t}(o, l, s_1; s)$

IF-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\texttt{o}(o, a, u, q_{in}, q_{out}) \vdash \texttt{t}(o, l, \textbf{if}\ e\ \{s_1\}\ \textbf{else}\ \{s_2\}; s) \to \texttt{t}(o, l, s_2; s)$

WHILE-TRUE-2: If $[\![e]\!]_{(a,l)} \neq 0$, then
$\texttt{o}(o, a, u, q_{in}, q_{out}) \vdash \texttt{t}(o, l, \textbf{while}\ e\ \{s_1\}; s) \to \texttt{t}(o, l, s_1; \textbf{while}\ e\ \{s_1\}; s)$

WHILE-FALSE-2: If $[\![e]\!]_{(a,l)} = 0$, then
$\texttt{o}(o, a, u, q_{in}, q_{out}) \vdash \texttt{t}(o, l, \textbf{while}\ e\ \{s_1\}; s) \to \texttt{t}(o, l, s)$

Figure 8: mABS type 2 reduction rules, part 1

the new runtime shape of objects. The remaining reduction rules are given in Figure 9; these rules can be divided into groups as per below.

Routing The first set of rewrite rules, T-SEND and T-RCV, are concerned with the exchange of routing tables, which only takes place between distinct adjacent nodes.

Message Passing The three rules MSG-SEND, MSG-RCV, and MSG-ROUTE are used to manage message passing, i.e., reading a message from a link queue and transferring it to the appropriate object in-queue, and dually, reading a message from an out-queue and transferring it to the attached link queue. If the destination object does not reside at the current node, the message is routed to the next link. In rule MSG-RCV, note that the receiving node is not required to be present. This, however, is enforced by the well-formedness condition for the network graph, which prohibits output links.

Unstable Routing The three rules MSG-DELAY-1, MSG-DELAY-2, and MSG-DELAY-3 are used to handle the cases where routing tables have not yet stabilized, or a message is unroutable. For instance, it may happen that updates to the routing tables have not yet caught up with object migration. In this case, a message may enter an object out-queue without the hosting node's routing table having information about the message's destination (rule MSG-DELAY-2). Another case is when a node receives a message on a link without knowing where to forward it (rule MSG-DELAY-1). This situation is particularly problematic, as a blocked message may prevent routing table updates to reach the hosting node, thus causing a deadlock. The solution we propose, which is implicit in the rules, is to use the network self-loop links, included in all well-formed networks, as buffers for unroutable mes-

T-SEND: If $u \neq u'$, then $\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u') \rightarrow \mathsf{l}(u, \mathtt{enq}(\mathsf{table}(t),q), u')$

T-RCV: If $\mathtt{hd}(q) = \mathsf{table}(t')$, then $\mathsf{l}(u',q,u)\ \mathsf{n}(u,t) \rightarrow \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{n}(u, \mathtt{upd}(t,u',t'))$

MSG-SEND: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, and $\mathtt{nxt}(o',t) = u'$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u') \rightarrow \mathsf{o}(o,a,u,q_{in}, \mathtt{deq}(q_{out}))\ \mathsf{l}(u, \mathtt{enq}(msg,q), u')$

MSG-RCV: If $\mathtt{hd}(q) = msg$ and $\mathtt{dst}(msg) = o$, then
$\mathsf{l}(u',q,u)\ \mathsf{o}(o,a,u,q_{in},q_{out}) \rightarrow \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{o}(o,a,u, \mathtt{enq}(msg,q_{in}), q_{out})$

MSG-ROUTE: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, $\mathtt{nxt}(o,t) = u''$, and $u'' \neq u$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u'') \rightarrow \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{l}(u, \mathtt{enq}(msg,q'), u'')$

MSG-DELAY-1: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, and $\mathtt{nxt}(o,t) \uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u',q,u)\ \mathsf{l}(u,q',u) \rightarrow \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{l}(u, \mathtt{enq}(msg,q'), u)$

MSG-DELAY-2: If $\mathtt{hd}(q_{out}) = msg$, $\mathtt{dst}(msg) = o'$, and $\mathtt{nxt}(o',t) \uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{l}(u,q,u) \rightarrow \mathsf{o}(o,a,u,q_{in}, \mathtt{deq}(q_{out}))\ \mathsf{l}(u, \mathtt{enq}(msg,q), u)$

MSG-DELAY-3: If $\mathtt{hd}(q) = msg$, $\mathtt{dst}(msg) = o$, and $\mathtt{nxt}(o,t) \uparrow$, then
$\mathsf{n}(u,t) \vdash \mathsf{l}(u,q,u) \rightarrow \mathsf{l}(u, \mathtt{enq}(msg, \mathtt{deq}(q)), u)$

CALL-SEND-2: Let $o' = [\![e_1]\!]_{(a,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$, $f = \mathtt{newf}(u)$, $a' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f,a))$ in
$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x = e_1!m(\overline{e_2});s) \rightarrow$
$\quad \mathsf{o}(o,a',u,q_{in}, \mathtt{enq}(\mathsf{call}(o',o,f,m,\overline{v}), q_{out}))\ \mathsf{t}(o,l[f/x],s)$

CALL-RCV-2: If $\mathtt{hd}(q_{in}) = \mathsf{call}(o,o',f,m,\overline{v})$, then
let $a' = \mathtt{fw}(f,o', \mathtt{init}(\overline{v},a))$, $l = \mathtt{locals}(o,f,m,\overline{v})$, $s = \mathtt{body}(o,m)$ in
$\mathsf{o}(o,a,u,q_{in},q_{out}) \rightarrow \mathsf{o}(o,a',u, \mathtt{deq}(q_{in}), q_{out})\ \mathsf{t}(o,l,s)$

FUT-SEND: If $a(f) = (v, o_1\ \overline{o_2})$, then let $a' = \mathtt{fw}(v, o_1, a[(v, \overline{o_2})/f])$ in
$\mathsf{o}(o,a,u,q_{in},q_{out}) \rightarrow \mathsf{o}(o,a',u,q_{in}, \mathtt{enq}(\mathsf{future}(o_1,f,v)), q_{out})$

FUT-RCV: If $\mathtt{hd}(q_{in}) = \mathsf{future}(o,f,v)$, then
$\mathsf{o}(o,a,u,q_{in},q_{out}) \rightarrow \mathsf{o}(o,a[v/f],u, \mathtt{deq}(q_{in}), q_{out})$

RET-2: Let $v = [\![e]\!]_{(a,l)}$, $f = l(\mathbf{ret})$ in
$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,\mathbf{return}\ e;s) \rightarrow \mathsf{o}(o,a[v/f],u,q_{in},q_{out})$

GET-2: If $[\![e]\!]_{(a,l)} = f$ and $\pi_1(a(f)) = v$, then
$\mathsf{o}(o,a,u,q_{in},q_{out}) \vdash \mathsf{t}(o,l,x = e.\mathbf{get};s) \rightarrow \mathsf{t}(o,l[v/x],s)$

NEW-2: Let $o' = \mathtt{newo}(u)$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{fw}(\overline{v}, o', a)$, $a'' = \mathtt{init}(\overline{v}, \mathtt{init}(C, \overline{v}, o'))$ in
$\mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\overline{e});s) \rightarrow$
$\quad \mathsf{o}(o,a',u,q_{in},q_{out})\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a'',u,\varepsilon,\varepsilon)$

OBJ-REG: $\mathsf{o}(o,a,u,q_{in},q_{out}) \vdash \mathsf{n}(u,t) \rightarrow \mathsf{n}(u, \mathtt{reg}(o,u,t,0))$

OBJ-SEND: If $u \neq u'$, then let $cn' = \mathtt{clo}(cn,o)$ in
$\mathsf{n}(u,t)\ \mathsf{l}(u,q,u')\ cn \rightarrow \mathsf{n}(u, \mathtt{reg}(o,u',t,1))\ \mathsf{l}(u, \mathtt{enq}(\mathsf{object}(cn'),q), u')\ (cn - cn')$

OBJ-RCV: If $\mathtt{hd}(q) = \mathsf{object}(cn)$, then
$\mathsf{l}(u',q,u)\ \mathsf{n}(u,t)\ \rightarrow \mathsf{l}(u', \mathtt{deq}(q), u)\ \mathsf{n}(u, \mathtt{reg}(\mathtt{oidof}(cn),u,t,0))\ \mathtt{place}(cn,u)$

Figure 9: mABS type 2 reduction rules, part 2

sages that may become routable. Rule MSG-DELAY-3 allows messages on this link to be shuffled.

Producing and Consuming Messages The four rules CALL-SEND-2, CALL-RCV-2, FUT-SEND, and FUT-RCV produce and consume method call and future

instantiation messages, respectively. A method call causes a local future identifier to be created and passed along with the call message. Upon receiving the call, the callee first initializes the received futures it does not already know about, and then augments the resulting local object environment to enact forwarding for the received future to the caller, when possible. Observe that it may be the case that the callee already knows about the return future of the call; since message order is not assumed to be preserved, a later call with the original return future may overtake the earlier call. The eventual return value becomes associated with the return future by the assignment to the constant **ret** during initialization of the task's local environment. The rule FUT-SEND lets future instantiations be forwarded to objects in the forwarding list whenever the future is instantiated to a value locally, and FUT-RCV causes the receiving object to update its local environment accordingly. A future may itself be instantiated to a future, making it necessary to update the local forwarding list whenever FUT-SEND is used.

Language Constructs The three rules RET-2, GET-2, and NEW-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained above. Get statements read the value of the future, provided it has received a value, and new statements cause a new object to be created, initialized, and registered at the local node. The arguments provided to the new object may contain future identifiers, whose values must be duly forwarded to the new object by augmenting the old object's forwarding lists.

Object Registration and Migration The rule OBJ-REG registers a new object on the node on which it has been placed. The final two rules concern object migration. Of these, the rule OBJ-SEND is *global* in that it is not allowed to be used in subsequent applications of the CTXT-1 rule. In this way, we can guarantee that only complete object closures are migrated. To remove an object closure cn' from a configuration cn for migration, we take the multiset difference $cn - cn'$.

It is important to note that all of the above rules are strictly local and appeal only to mechanisms directly implementable at the link level, i.e., they correspond to tests and simple datatype manipulations taking place at a single node, or accesses to a single node's link layer interface. The "global" property appealed to above for migration is merely a formal device to enable an elegant treatment of object closures.

The reduction rules can be optimized in several ways. For instance, object self-calls are always routed through the "network interface", i.e., the hosting node's self-loop link. This is not necessary. It would be possible to add a rule to directly spawn a handling task from a self-call without affecting the results of the paper.

We note some elementary properties of the type 2 semantics.

Proposition 6.2. *Suppose that $cn \to cn'$. Then, the following holds:*

1. *If $\mathsf{n}(u,t) \preceq cn$, then $\mathsf{n}(u,t') \preceq cn'$ for some t'.*

2. *If* $\mathsf{l}(u, q, u') \preceq cn$, *then* $\mathsf{l}(u, q', u') \preceq cn'$ *for some* q'.

3. *If* $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, *then there is an object* $\mathsf{o}(o, a', u', q'_{in}, q'_{out}) \preceq cn'$ *(the* derivative *of the object in* cn'*), such that for all variables* x*, if* $a(x)\downarrow$*, then* $a'(x)\downarrow$*, and for all future identifiers* f*, if* $a'(f)\downarrow$*, then* $a'(f)\downarrow$*, and if* $\pi_1(a(f))\downarrow$*, then* $\pi_1(a'(f))\downarrow$.

4. *If* $\mathsf{t}(o, l, s) \preceq cn$ *and* $l(\textbf{\textit{ret}}) = f$*, then either there is a task* $\mathsf{t}(o, l', s') \preceq cn'$ *(the* derivative *of the task in* cn'*), such that* $\mathtt{dom}(l) \subseteq \mathtt{dom}(l')$*, and* $l'(\textbf{\textit{ret}}) = f$*, or there is an object* $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn'$ *such that* $\pi_1(a(f))\downarrow$.

Proof. By straightforward induction on the reduction relation. □

Definition 6.3 (Type 2 Initial Configuration, Type 2 Reachable)**.** Consider a program $\overline{CL}\,\{\overline{x}, s\}$, using the special OID ext to communicate with the outside the world. Assume a reserved OID o_{main}, distinct from ext, and a reserved FID f_{init}. A *type 2 initial configuration* for the program has the shape

$$cn_{init} = \mathsf{o}(o_{main}, a_{init,2}, u_{init}, \varepsilon, \varepsilon)\ \mathsf{t}(o_{main}, l_{init}, s)\ cn_{graph}\ ,$$

where

- $a_{init,2} = \bot[(\bot, \varepsilon)/f_{init}]$,
- l_{init} is unchanged from the definition of type 1 initial configurations,
- cn_{graph} is a configuration consisting only of nodes and links, inducing a well-formed network graph,
- cn_{graph} contains a node $\mathsf{n}(u_{init}, t_{init})$,
- $t_{init}(o_{main}) = (u_{init}, 0)$, and $t_{init}(o) = \bot$ for all OIDs o distinct from o_{main},
- $t(o) = \bot$ for all routing tables $t \neq t_{init}$ in cn_{graph} and for all OIDs o, and
- $\mathsf{l}(u, q, u') \preceq cn_{graph}$ implies $q = \varepsilon$, for all u and u'.

When there is a derivation $cn_1 \to \cdots \to cn_n$, we say that cn_n is *reachable* from cn_1. If cn_1 is a type 2 initial configuration, cn_n is said to be *type 2 reachable*.

7 Type 2 Well-formedness

Well-formedness becomes more complex in the case of the network-aware semantics, since account must be taken of, e.g., queues, messages in transit, and routing, to ensure that, e.g., multiple objects are never given identical names, and that futures are never assigned inconsistent values, as detailed below. A particularly delicate matter concerns the way future instantiations are propagated. The well-formedness condition needs to ensure that either

all objects that may at some time need the value of a future can also eventually receive it, or else no object is able to do so (in which case the task for which the future is a placeholder for the return value fails to terminate). This is the "future liveness" property in Definition 7.5 below. To this end, we first define when a future identifier is assigned in a configuration and active for an object.

Definition 7.1 (Future Assignment)**.** We say that a configuration cn *assigns* the value v *to* f if there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, such that either $\pi_1(a(f)) = v$ or there is a message $\mathsf{future}(o, f, v) \preceq cn$. If there is no such value v, we say that f is *unassigned* in cn.

Definition 7.2 (Type 2 Active Future)**.** Let cn be a type 2 configuration. The future identifier f is *active* for the object with identifier o in cn if one of the following holds:

1. There is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ such that $a(x) = f$ for some variable x.
2. There is a task container $\mathsf{t}(o, l, s) \preceq cn$ such that $l(x) = f$ for some x.
3. There is a call message $\mathsf{call}(o, o', f', m, \overline{v})$ in transit in cn, and $f' = f$ or f occurs in $\overline{v}$.
4. There is a future identifier f' that is active for o, and cn assigns f to f'.

Thus, f is active for o if f occurs in the enviroment of the object or one its tasks, the object is about to receive a method call with f as the return future or as one of its arguments, or f' is active for o and f' has been instantiated to f somewhere.

We next define which objects are due to be notified by eventual future instantiations.

Definition 7.3 (Notification Path)**.** Fix a type 2 configuration cn, an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, and an OID $o' \in OID(cn)$. Let n be a nonnegative integer. Inductively, o is *on the notification path of f by o' in n steps*, if n is the least number such that one of the following conditions hold:

1. $n = 0$, $o' = o$, and $\pi_1(a(f)) = v$.
2. $n = 1$, $o' = o$, and there is a future message $\mathsf{future}(o, f, v) \preceq cn$.
3. $n = 1$, $o' = o$, and there is a task $\mathsf{t}(o, l, s) \preceq cn$ with $l(\mathbf{ret}) = f$.
4. $n = 2$, $o' = o$, and there is a call message $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn$.
5. $n = 4$, and there is a call message $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn$.
6. $n = n' + 2$, with n' a nonnegative integer, and there is an object $\mathsf{o}(o'', a', u', q'_{in}, q'_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and o'' is on the notification path of f by o' in n' steps.

7. $n = 2n' + n''$, with n' and n'' nonnegative integers, if o is on the notification path of f' by o'' in n' steps, cn assigns f to f', and o'' is on the notification path of f by o' in n'' steps.

Say that o is *on the notification path of* f, if o is on the notification path of f by some o' in some number of steps.

Intuitively, the number of steps in a notification path is the number of events that need to take place before a future becomes assigned at the object, considering task evaluation a single event. For instance, if o' has a task evaluating f and has added o to the forwarding list for f, it takes three steps for the value of f to reach o: the evaluation of the task, the sending of a future message, and the reception of that future message by o.

Condition 1 is the base case when f has already been instantiated. Condition 2 holds if a future has been resolved and a future message is in transit to o. Condition 3 holds if o is due to receive the return value of f from one of its pending tasks. Condition 4 holds if a call to o' has been sent off from o with return future f; o is then on the notification path of f since the call message is guaranteed to be received at the callee's site (if at all) and the forwarding list there updated. Condition 5 holds if o has been inserted into a forwarding list for f at closer distance to a "source". Condition 6 holds if f is assigned to another future f', o'' is a "source" of f' for o, and o'' is due to receive the return value for f, which can then reach o through forwarding list additions in transitions using the rule FUT-SEND.

We prove that if o is on the notification path of f, then in the next configuration o remains on the notification path of f, without increasing the number of steps.

Lemma 7.4. *Fix a configuration cn and an object $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$. If o is on the notification path of f by o' in n steps in the configuration cn and $cn \to cn'$, then o is on the notification path of f by o' in at most n steps in cn'.*

Proof. See Appendix 2. □

We can now finally state the conditions of type 2 well-formedness.

Definition 7.5 (Type 2 Well-formedness)**.** A type 2 configuration cn is *type 2 well-formed* (WF2) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2})$ are distinct object container occurrences in cn, then $o_1 \neq o_2$.
2. *Task-Object Existence*: If $\mathsf{t}(o, l, s) \preceq cn$, then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, and q_{out}.
3. *Object-Node Existence*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, then $\mathsf{n}(u, t) \preceq cn$ for some t.
4. *Buffer Cleanliness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ and $msg \preceq q_{in}$ or $msg \preceq q_{out}$, then msg is object bound. Additionally, if $msg \preceq q_{in}$, then $\mathtt{dst}(msg) = o$.

5. *Local Routing Consistency*: If $\mathsf{n}(u,t) \preceq cn$ and $\mathtt{nxt}(o,t) = u'$, then there is a link $\mathsf{l}(u,q,u') \preceq cn$.

6. *Call Uniqueness*: If $\mathsf{call}(o_1, o_1', f_1, m_1, \overline{v}_1)$ and $\mathsf{call}(o_2, o_2', f_2, m_2, \overline{v}_2)$ are distinct call message occurrences in cn, then $f_1 \neq f_2$.

7. *Future Uniqueness*: If cn assigns both v_1 and v_2 to f, then $v_1 = v_2$.

8. *Single Writer*: If $\mathsf{t}(o_1, l_1, s_1)$ and $\mathsf{t}(o_2, l_2, s_2)$ are distinct task container occurrences in cn such that $l_1(\mathbf{ret}) = f_1$ and $l_2(\mathbf{ret}) = f_2$, then $f_1 \neq f_2$ and both f_1 and f_2 are unassigned in cn, and if $\mathsf{call}(o, o', f, m, \overline{v}) \preceq cn$, then $f \neq f_1$ and $f \neq f_2$.

9. *External OID*: $ext \notin OID(cn)$, and if $\mathsf{n}(u,t) \preceq cn$, then $ext \notin \mathtt{dom}(t)$.

10. *Future Liveness*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, and either f is active for o in cn, $a(f) \downarrow$, or cn assigns f to f' and o is on the notification path of f', then o is on the notification path of f.

Conditions 7.5.1 to 7.5.5 and 7.5.9 are inherited from our preceding work [11]. 7.5.4 is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. The rationale behind 7.5.8 is that mABS enforces a single-writer discipline on instantiated values of futures. Once a future has been instantiated through the evaluation of a return statement, the task is "garbage collected" in the rule RET-2. 7.5.9 ensures that messages to ext are only transported to reflexive links, where they remain. Condition 7.5.10 is the future propagation property discussed above. We use the designation Future Liveness not to indicate a guarantee that f will eventually be instantiated at an object, but to indicate that, if eventually f is instantiated somewhere, a notification path to the object exists along which the instantiation can be propagated.

Lemma 7.6 (WF2 Preservation)**.** *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 2 initial configuration, then cn is WF2.*

2. *If cn is WF2 and $cn \to cn'$, then cn' is WF2.*

3. *If cn is type 2 reachable, then cn is WF2.*

Proof. See Appendix 2. □

An easy but important consequence of type 2 well-formedness is that assignments to futures cannot be updated with new values.

Proposition 7.7. *Suppose that cn is WF2 and $cn \to cn'$. If cn assigns v to f and cn' assigns v' to f, then $v = v'$.*

Proof. Since cn is WF2, if cn assigns v to f, there cannot be a task $\mathsf{t}(o, l, s) \preceq cn$ such that $l(\mathbf{ret}) = f$. But the only way of assigning $v' \neq v$ to f is through RET-2. Hence, the result follows. □

8 Type 2 Contextual Equivalence

We adapt the notion of contextual equivalence to the type 2 setting as in the preceding work [11]. The initial problem is to define the type 2 correlate of the observation predicate. Say a configuration cn has the observation, or *barb*, $obs = ext!m(\overline{v})$ if a corresponding call message $\mathsf{call}(ext, o, f, m, \overline{v})$ is located at the head of one of the self-loop link queues in cn. More precisely, the type 2 observability predicate $cn \downarrow obs$ holds whenever cn has the shape

$$cn = cn' \; \mathsf{l}(u, q, u) \; ,$$

and $\mathtt{hd}(q)$ is defined and equal to $\mathsf{call}(ext, o, f, m, \overline{v})$.

It might be thought that the FIFO queue discipline goes against the treatment of external calls in the type 1 semantics, since there, an external call container, once created, will remain as an inert element of all subsequent configurations. Thus, once, say, five call containers have been created, all five calls can be observed at all configurations from that point onwards. This is obviously not the case in the type 2 semantics. On the other hand, in that semantics, external call messages can always be shuffled on a reflexive link using the rule MSG-DELAY-3, allowing specific calls to reach the head of the queue.

There are other ways of defining the observability predicate that may be more natural. For instance, one may attach external OIDs to specific NIDs and restrict observations to those NIDs accordingly. It is also possible to add dedicated output channels to the model, and route external calls to those. None of these design choices have any effect on the subsequent results, however, but add significant notational overhead, particularly in the latter case.

For context closure, a *context* is any configuration cn containing only object and task containers. Thus, contexts do not affect the underlying network graph. This definition is used, since, firstly, it is objects and tasks that induce computational behavior, and secondly, allowing contexts to augment the underlying graph by adding new nodes and links requires a much more complex account of network composition and well-formedness that we prefer to leave to future work.

With the observation predicate set up, the weak observation predicate is derived as in Section 5, and, as there, we define a *type 2 witness relation* as a relation that satisfies reduction closure, and barb preservation, with context closure defined as follows: If $cn_1 \mathcal{R} cn_2$, cn is a context, and $cn_1 cn$ is WF2, then $cn_2 cn$ is WF2, and $cn_1 cn \; \mathcal{R} \; cn_2 cn$. Thus:

Definition 8.1 (Type 2 Contextual Equivalence)**.** Let $cn_1 \simeq_2 cn_2$ whenever $cn_1 \; \mathcal{R} \; cn_2$ for some type 2 witness relation $\mathcal{R}$.

It is straightforward to relate a type 2 context with a type 1 context. Given a type 2 context cn, produce a type 1 configuration by first mapping pointwise an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ to $\mathsf{o}(o, a)$, each message $\mathsf{call}(o, o', f, m, \overline{v})$ in q_{in} and q_{out} to a call container $\mathsf{c}(o, f, m, \overline{v})$, and each task $tsk \preceq cn$ to itself. Then, for all f active for some o in cn, add a future

container $\mathsf{f}(f, v)$ if cn assigns v to f, and $\mathsf{f}(f, \bot)$ if f is unassigned in cn. The resulting configuration is well-defined as long as object queues only contain messages that are object bound, which is the case for configurations that are WF2. With this proviso, we do not need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the respective notions of witness relation and contextual equivalence, by letting the type of the configuration arguments be determined by the context, and use $\simeq$ as the generic notion.

9 Normal Forms

An mABS program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the type 1 behavior of a program is preserved in the type 2 semantics, in the sense that the type 1 and type 2 initial states are contextually equivalent.

The key to the proof is a normal form lemma for mABS saying, roughly, that any well-formed type 2 configuration can be rewritten into a form where queues have been emptied of all routable messages, where routing tables have been in some expected sense normalized, where all futures that are assigned a value somewhere are assigned a value everywhere the value might be needed (by well-formedness, this value is unique), and where all objects have been moved to a single node. We perform this rewriting using two algorithms. The first algorithm stabilizes routing and empties link queues, leaving only external ("unroutable") messages. The second algorithm, which uses the first, empties object queues, propagates futures, and moves all objects to a single node.

9.1 Stabilization

In the scope of a configuration cn, we call a link $\mathsf{l}(u, q, u') \preceq cn$ *proper* whenever $u \neq u'$, and say that a message msg is *routable* whenever $\mathtt{dst}(msg) \in OID(cn)$, and *unroutable* otherwise.

Definition 9.1 (Stable Routing)**.** Let cn be a type 2 configuration. We say that cn has *stable routing*, if for all $\mathsf{n}(u, t)$, $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$, if $\mathtt{nxt}(o, t) = u''$, then there is a minimal length path from u to u' in $\mathtt{graph}(cn)$ which visits u''.

Definition 9.2 (External Link Messages)**.** Let cn be a type 2 configuration. We say that cn has *external link messages*, if $\mathsf{l}(u, q, u') \preceq cn$ and $msg \preceq q$ implies msg is object bound and unroutable.

To converge to a configuration with stable routing, the idea is to empty link queues as far as possible, and let nodes simultaneously exchange routing tables. This is accomplished using Algorithm 1, where we hide uses of CTXT-1 to allow the transition rules to be applied to arbitrary containers. Write $\mathcal{A}_1(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 1 to cn. The resulting configuration is almost unique, but not quite, since routing may stabilize in different ways.

Algorithm 1: Stabilize routing and process link messages

Input A type 2 well-formed configuration cn
Output A configuration with stable routing and external link messages, reachable from cn

repeat
  use OBJ-REG on each object not in transit ;
  use T-SEND on each proper link to broadcast routing tables from all nodes to their neighboring nodes ;
  repeat
    use T-RCV to dequeue one message on a link
  until T-RCV can no longer be used ;
  once for each link, if possible, use MSG-RCV, MSG-ROUTE, MSG-DELAY-1, or OBJ-RCV, or otherwise, use MSG-DELAY-3 on self-loop links with external messages at the queue head
until routing has stabilized and there are only external link messages

Figure 10: Algorithm 1

Proposition 9.3. *Algorithm 1 terminates.*

Proof. See Appendix 3. □

Let $\mathsf{o}_1(cn)$ be the multiset of object containers where, if it is the case that $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, there is a corresponding container $\mathsf{o}(o, a, u', q'_{in}, q_{out})$, such that the NID u' is that of the receiving node if the object is in transit in cn, or otherwise $u' = u$, and additionally, all messages msg in link queues such that $\mathtt{dst}(msg) = o$ have been enqueued in some fixed order in q_{in} to produce q'_{in}. Define the relation $\cong_1$ to hold between multisets of object containers when there is a one-to-one mapping where elements only possibly differ in how in-queue messages are ordered. Finally, let $\mathsf{m}_1(cn)$ be the multiset of both external messages and routable messages in cn.

Definition 9.4 (Stable Form)**.** A WF2 configuration cn is in *stable form*, if cn has stable routing, $\mathsf{o}(cn) \cong_1 \mathsf{o}_1(cn)$, and $\mathsf{m}(cn) = \mathsf{m}_1(cn)$.

Proposition 9.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then*

1. *cn' is reachable from cn,*
2. *cn' is in stable form,*
3. $\mathtt{graph}(cn') = \mathtt{graph}(cn')$,
4. $\mathsf{t}(cn') = \mathsf{t}(cn)$,
5. $\mathsf{o}(cn') \cong_1 \mathsf{o}_1(cn)$, *and*
6. $\mathsf{m}(cn') = \mathsf{m}_1(cn)$.

Proof. Properties 1, and 3 are immediate. Property 2 can be read out of the termination proof. For the remaining three properties, observe first that t,

o_1, and m_1 are all invariant under the transitions used in Algorithm 1. The equations then follow by noting that only externally-addressed messages (and so no object closures or routing tables) are in transit in links in cn'. □

Proposition 9.5 makes precise the "almost unique" property alluded to above. The normal form property suggested by Proposition 9.5 motivates a notion of equivalence "up to stabilization", defined below.

Definition 9.6 ($\equiv_1$)**.** Define a relation $\mathcal{R}_1$ on type 2 configurations such that $cn_1\ \mathcal{R}_1\ cn_2$ whenever $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$, $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$, $\mathsf{o}(cn_1) \cong_1 \mathsf{o}(cn_2)$, and $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$. Let $cn_1 \equiv_1 cn_2$ if cn_1 and cn_2 are WF2 and there are cn_1' and cn_2' such that $\mathcal{A}_1(cn_1) \rightsquigarrow cn_1'\ \mathcal{R}_1\ cn_2' \reflectbox{$\rightsquigarrow$} \mathcal{A}_1(cn_2)$.

Proposition 9.5, together with the termination of $\mathcal{A}_1$, allows the existential quantifiers in Definition 9.6 to be exchanged for universal ones.

Corollary 9.7. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \equiv_1 cn'$.*

Proof. We have $\mathcal{A}_1(cn) \rightsquigarrow cn'\ \mathcal{R}_1\ cn' \reflectbox{$\rightsquigarrow$} \mathcal{A}_1(cn')$. □

Lemma 9.8. *$\equiv_1$ is reduction closed.*

Proof. See Appendix 3. □

Lemma 9.9. *$\equiv_1$ is context closed.*

Proof. See Appendix 3. □

Proposition 9.10. *$\equiv_1$ is a type 2 witness relation.*

Proof. See Appendix 3. □

Corollary 9.11. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

Proof. By Proposition 9.10 and Corollary 9.7. □

9.2 Normalization

We turn to the second procedure, which empties object queues, propagates futures, and migrates all objects and their tasks to a single node. The procedure, Algorithm 2, is shown in Figure 11. Write $\mathcal{A}_2(cn) \rightsquigarrow cn'$ if cn' is a possible result of applying Algorithm 2 to cn. Initially, a node u is chosen towards which all objects will migrate during normalization. Normalization is then performed in cycles, with each cycle starting and ending in a configuration in stable form. In each cycle, one message is read from an object's in- and out-queues. By well-formedness, object queues contain only call and future messages. Receptions of future messages may cause object environments to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and delivered to an object out-queue. Once this is done, objects not yet at u will be migrated.

Proposition 9.12. *Algorithm 2 terminates.*

Algorithm 2: Normalization

Input A type 2 well-formed configuration cn
Output A configuration in normal form, reachable from cn

fix a NID u for a node in cn ;
run Algorithm 1 ;
repeat
  while some object queue is nonempty
    use MSG-SEND, MSG-DELAY-2, CALL-RCV-2, or FUT-RCV to dequeue one message from each nonempty object queue ;
    repeat
      use FUT-SEND to send future instantiation messages
    until FUT-SEND can no longer be used
  end ;
  while an object o exists not located at u
    use OBJ-SEND to send o towards u
  end ;
  run Algorithm 1
until all objects are located at u, all object queues are empty, and there are only external link messages

Figure 11: Algorithm 2

Proof. See Appendix 3. □

We next turn to normal forms, and define first a couple of auxiliary operations.

Let $\mathsf{t}_2(cn)$ be the multiset of task containers $tsk = \mathsf{t}(o, l, s)$ such that

- $tsk \preceq cn$, or
- there is a routable message $\mathsf{call}(o, o', f, m, \overline{v})$ in transit in cn, such that $l = \mathtt{locals}(o, f, m, \overline{v})$ and $s = \mathtt{body}(o, m)$.

Let $\mathsf{m}_2(cn)$ be the multiset of external messages in queues in cn. Let $\mathsf{o}_2(cn)$ be the multiset of object containers $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ for which all of the following holds:

- $u' = u$.
- There is an object container $\mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$.
- $a(x) = a'(x)$, for all variables x, and $a(\mathbf{self}) = a'(\mathbf{self})$.
- $a(f) = (v, \varepsilon)$ if cn assigns v to f, and o is on the notification path of f in cn.
- $a(f) \uparrow$, if f is unassigned in cn, $a'(f) \uparrow$, and there is no f' such that cn assigns f to f' and, additionally, o is on the notification path of f'.

- $a(f) = (\bot, \overline{o})$ if f is unassigned in cn, and $a'(f) \downarrow$ or there is an f' such that cn assigns f to f' and, additionally, o is on the notification path of f', with $\overline{o}$ defined as follows. Let $f_1, \ldots, f_n$ be all future assignments such that, for all f_i, o is on the notification path of f_i, and either cn assigns f_i to f, or there is a sequence of assignments of futures f_j starting from f_i leading to f in cn. Let $\overline{o}_i$ be the forwarding list such that $\overline{o}_i = \pi_2(a'(f_i))$ if $a'(f_i) \downarrow$, and $\overline{o}_i = \varepsilon$ if $a'(f_i) \downarrow$. Let $\overline{o}'$ be the forwarding list such that $\overline{o}' = \pi_2(a'(f))$ if $a'(f) \downarrow$, and $\overline{o}' = \varepsilon$ if $a'(f) \uparrow$. Then, if $\mathsf{call}(o, o', f, m, \overline{v}) \preceq cn$ for some o', m, and $\overline{v}$, set $\overline{o} = o' \, \overline{o}_1 \, \ldots \, \overline{o}_n \, \overline{o}'$, and set $\overline{o} = \overline{o}_1 \, \ldots \, \overline{o}_n \, \overline{o}'$ otherwise.

The intent of these functions is to describe the effects of running Algorithm 2 on a configuration, the details of which we make precise in the following definition and proposition.

Definition 9.13 (Normal Form)**.** A WF2 configuration cn is in *normal form*, if

1. cn has stable routing,
2. $\mathsf{t}(cn) = \mathsf{t}_2(cn)$,
3. $\mathsf{o}(cn) = \mathsf{o}_2(cn)$, and
4. $\mathsf{m}(cn) = \mathsf{m}_2(cn)$.

Proposition 9.14. *If cn is WF2 and $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then the following holds:*

1. *cn' is reachable from cn.*
2. *cn' is in normal form.*
3. $\mathtt{graph}(cn) = \mathtt{graph}(cn')$.
4. $\mathsf{t}_2(cn) = \mathsf{t}(cn')$.
5. $\mathsf{o}_2(cn) = \mathsf{o}(cn')$.
6. $\mathsf{m}_2(cn) = \mathsf{m}(cn')$.

Proof. See Appendix 3. □

Proposition 9.14 motivates the following definition of normal form equivalence.

Definition 9.15 ($\equiv_2$)**.** Define a relation $\mathcal{R}_2$ on type 2 configurations such that $cn_1 \, \mathcal{R}_2 \, cn_2$ whenever $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$, $\mathsf{t}(cn_1) = \mathsf{t}(cn_2)$, $\mathsf{o}(cn_1) = \mathsf{o}(cn_2)$, and $\mathsf{m}(cn_1) = \mathsf{m}(cn_2)$. Let $cn_1 \equiv_2 cn_2$ if cn_1 and cn_2 are WF2 and there are cn_1' and cn_2' such that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \, \mathcal{R}_2 \, cn_2' \nrightsquigarrow \mathcal{A}_2(cn_2)$.

Clearly, $\equiv_2$ relates more extended configurations than $\equiv_1$.

Corollary 9.16. $\equiv_1 \subseteq \equiv_2$.

Proof. If $cn_1 \equiv_1 cn_2$, the two configurations have the same task containers, and the same object bound messages. In addition, there is a one-to-one mapping between object containers where identifiers and object environments coincide. The result follows by noting that any remaining differences between the containers will disappear after running Algorithm 2. □

We also obtain that normalization respects normal form equivalence.

Corollary 9.17. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$.*

Proof. We have $\mathcal{A}_2(cn) \rightsquigarrow cn' \; \mathcal{R}_2 \; cn' \reflectbox{$\rightsquigarrow$} \mathcal{A}_2(cn')$. □

The proof of reduction closure follows that of Lemma 9.8 quite closely.

Lemma 9.18. *$\equiv_2$ is reduction closed.*

Proof. See Appendix 3. □

Lemma 9.19. *$\equiv_2$ is context closed.*

Proof. See Appendix 3. □

Proposition 9.20. *$\equiv_2$ is a type 2 witness relation.*

Proof. Similar to the proof of Proposition 9.10. □

Corollary 9.21. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \simeq_2 cn'$.*

Proof. None of the rules used in Algorithm 2 affect the shape of the normal form. Thus, if $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then $cn \equiv_2 cn'$. But then, $cn \simeq_2 cn'$ by Proposition 9.20. □

10 Correctness

In this Section, we prove the correctness of the network-aware semantics by mapping a well-formed type 1 configuration $\mathsf{bind}\ \overline{z}.cn$ in standard form to a well-formed type 2 configuration $\mathtt{net}(cn)$ close to normal form, with an arbitrary, but well-formed, underlying network graph. After establishing that the mapping $\mathtt{net}$ produces reasonable configurations, we prove that the two configurations are contextually equivalent.

10.1 The Representation Map

We first fix a well-formed network graph represented as a configuration cn_{graph}, containing a node with distinguished NID u_0, similarly to the way initial configurations are defined in Section 6. Thus, cn_{graph} consists of nodes and links only, with each node u in cn_{graph} having the form $\mathsf{n}(u, t)$, and each link having the form $\mathsf{l}(u, \varepsilon, u')$. The routing tables t are defined later.

Representing Names and Values To represent names, one complication is that names in the type 1 semantics need to be related to identifiers in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has two generator functions (the operations `newf` and `newo`). This prevents the name relation from being modeled using the identity relation. To address this, we assume that names in the type 1 semantics are really symbolic, connected to concrete identifiers used in the type 2 semantics by means of an injective *name representation map* `rep`, taking internal names f, o in the type 1 semantics to names $\texttt{rep}(f)$, $\texttt{rep}(o)$ in the type 2 semantics. For convenience, we extend the name representation map `rep` to arbitrary values and task environments as follows:

- $\texttt{rep}(ext) = ext$
- $\texttt{rep}(p) = p$, if $p \in PVal$
- $\texttt{rep}(l)(x) = \texttt{rep}(l(x))$
- $\texttt{rep}(l)(\textbf{ret}) = \texttt{rep}(l(\textbf{ret}))$

Representing Object Environments A problem in extending `rep` to object enviroments is that object environments in the type 2 semantics must be defined partially in terms of the type 1 environments (for object variables), and partially in terms of the future containers available in the "root configuration", since the type 1 semantics uses future containers in place of forwarding lists stored in object environments. To this end, we first define an auxiliary operation $\texttt{oenvmap}(cn, \wp, \texttt{rep}) : FID \to Val_\perp$ on triples of type 1 configurations cn, pools $\wp$ of OID/FID constants, and name representation maps `rep`, as a function which gathers together assignments to futures as determined by the future containers in cn:

- $\texttt{oenvmap}(0, \wp, \texttt{rep})(f) = \perp$
- $\texttt{oenvmap}(tsk, \wp, \texttt{rep})(f) = \perp$
- $\texttt{oenvmap}(obj, \wp, \texttt{rep})(f) = \perp$
- $\texttt{oenvmap}(call, \wp, \texttt{rep})(f) = \perp$
- $\texttt{oenvmap}(\texttt{f}(f, v_\perp), \wp, \texttt{rep})(f') =$ if $\texttt{rep}(f) = f'$ then $\texttt{rep}(v_\perp)$ else $\perp$
- $\texttt{oenvmap}(\texttt{bind } o.cn, \wp \cup \{o'\}, \texttt{rep})(f) = \texttt{oenvmap}(cn, \wp, \texttt{rep}[o'/o])(f)$
- $\texttt{oenvmap}(\texttt{bind } f.cn, \wp \cup \{f'\}, \texttt{rep})(f'') = \texttt{oenvmap}(cn, \wp, \texttt{rep}[f'/f])(f'')$
- $\texttt{oenvmap}(cn_1\ cn_2, \wp, \texttt{rep})(f) =$
  $\texttt{oenvmap}(cn_1, \wp, \texttt{rep})(f) \sqcup \texttt{oenvmap}(cn_2, \wp, \texttt{rep})(f)$

Fix now a root type 1 configuration cn_0 and a large enough pool $\wp_0$ of names (proportional to the size of cn_0, and computed using $\texttt{newf}(u_0)$ and $\texttt{newo}(u_0)$ to conform to our naming policy). Assume that cn_0 is in standard form, i.e., $cn_0 = \texttt{bind } \overline{z_0}.cn_0'$ where cn_0' does not have binders. Fix $g = \texttt{oenvmap}(cn_0, \wp_0, \perp)$ and cn_{graph} as above. We can now extend `rep` to object environments as follows:

- $\pi_1(\mathtt{rep}(a))(\mathbf{self}) = \mathtt{rep}(\pi_1(a)(\mathbf{self}))$
- $\pi_1(\mathtt{rep}(a))(x) = \mathtt{rep}(\pi_1(a)(x))$
- $\pi_2(\mathtt{rep}(a))(f) = \begin{cases} (v, \varepsilon) & \text{if } g(f) = v \\ (\bot, \mathit{OID}(cn_0)) & \text{otherwise} \end{cases}$

Since we have left the nature of expressions unspecified, we need to additionally assume that the representation map commutes with the expression semantics, i.e., for all e, a, and l, we assume the following equation holds:

$$\mathtt{rep}(\llbracket e \rrbracket_{(a,l)}) = \llbracket e \rrbracket_{(\mathtt{rep}(a),\mathtt{rep}(l))}. \tag{2}$$

Proposition 10.1. *Let cn_0 be a type 1 well-formed root configuration in standard form, and $\wp_0$ be a pool as above. Then, $\mathtt{rep}(a)(f) = (v, \varepsilon)$ if and only if $\mathsf{f}(f, v) \preceq cn_0$.*

Proof. By well-formedness, the future container, if it exists, is unique. Pick a name representation map `rep`. Then, $\mathtt{oenvmap}(cn_0, \wp_0, \mathtt{rep})(f)$ is defined and equal to v if and only if $\mathsf{f}(f, v) \preceq cn_0$. This is easily seen by induction on the structure of cn_0. □

Representing Call Containers Another complication is that we need an operation to represent a type 1 call container as a message in the type 2 semantics. Compared to type 1 call containers, type 2 call messages additionally contain the OID of the caller, which is added to the future forwarding list when the message is received. Since future forwarding lists are already extended maximally, it is possible to use the callee's OID as the caller OID when mapping containers to messages without affecting behaviour, and put the message in the self-loop queue for the node u_0. This is done in the obvious way by the operation `send`, as follows:

$$\begin{aligned} &\mathtt{send}(\mathsf{call}(o, o', f, m, \overline{v}), \mathsf{l}(u_0, q, u_0)\ cn) \\ &\quad = \quad \mathsf{l}(u_0, \mathsf{enq}(\mathsf{call}(o, o', f, m, \overline{v})), q), u_0)\ cn \end{aligned}$$

Representing Configurations Given a name representation map `rep`, we can now define the representation of a type 1 configuration as a transformer on type 2 configurations with the mapping `net`, as follows:

- $\mathtt{net}(cn_1\ cn_2, \mathtt{rep}) = \mathtt{net}(cn_1, \mathtt{rep}) \circ \mathtt{net}(cn_2, \mathtt{rep})$
- $\mathtt{net}(0, \mathtt{rep})(cn) = cn$
- $\mathtt{net}(\mathsf{t}(o, l, s), \mathtt{rep})(cn) = \mathsf{t}(\mathtt{rep}(o), \mathtt{rep}(l), s)\ cn$
- $\mathtt{net}(\mathsf{o}(o, a), \mathtt{rep})(cn) = \mathsf{o}(\mathtt{rep}(o), \mathtt{rep}(a), u_0, \varepsilon, \varepsilon)\ cn$
- $\mathtt{net}(\mathsf{f}(f, v_\bot), \mathtt{rep})(cn) = cn$
- $\mathtt{net}(\mathsf{c}(o, f, m, \overline{v}), \mathtt{rep})(cn) =$
  $\mathtt{send}(\mathsf{call}(\mathtt{rep}(o), \mathtt{rep}(o), \mathtt{rep}(f), m, \mathtt{rep}(\overline{v})), cn)$

In other words, we represent type 1 configurations by first assuming some underlying network graph, and then distributing the (centralized) assignments to futures in each object environment with the trivial forwarding lists, and then, once this is done, mapping the containers individually to the type 2 level.

Defining Routing Tables The only detail remaining to be addressed from above concerns the routing tables. For the node with NID u_0, the initial routing table, t_0, needs to have all objects in cn_0 registered, i.e.,

$$t_0 = \mathtt{reg}(\mathtt{rep}(o_1), u_0, \mathtt{reg}(\mathtt{rep}(o_2), u_0, \mathtt{reg}(\cdots, \mathtt{reg}(\mathtt{rep}(o_n), u_0, \bot)\cdots))) \ ,$$

where $OID(cn_0) = \{o_1, \ldots, o_n\}$. For nodes $\mathsf{n}(u, t)$ where $u \neq u_0$, we let t be determined by some stable routing. This is easily computed using Algorithm 1, and we leave out the details.

Defining the Map We can now finally define the representation map.

Definition 10.2 (Representation Map $\mathtt{net}$)**.** Let a network graph configuration cn_{graph} and a name representation map $\mathtt{rep}$ be given for a WF1 configuration bind $\overline{z}.cn$ in standard form. Then, the *type 2 representation* of the configuration is $\mathtt{net}(cn) = \mathtt{net}(cn, \mathtt{rep})(cn_{graph})$.

The most basic property that we expect to hold about the representation map is that it produces type 2 well-formed configuration given well-formed type 1 configurations.

Proposition 10.3. *If* bind $\overline{z}.cn$ *is a WF1 configuration in standard form, then* $\mathtt{net}(cn)$ *is WF2.*

Proof. See Appendix 4. □

10.2 Contextual Equivalence under Extension

Note that in the representation map, future-value mappings and forwarding lists are overapproximated when compared to the type 2 semantics, where future instantiations are only forwarded to and instantiated at objects that are on the notification path of the associated futures. We need to ensure that observable behavior does not change for well-formed configurations when future maps are extended in this way. To do this, we make precise the notion of extension for objects, and ultimately prove that configurations are contextually equivalent to their extended counterparts. Thus, a configuration produced by the mapping captures the behavior of a minimally extended type 2 configuration produced from an initial configuration.

Definition 10.4 (Forwarding Resolved)**.** We say that o' is *forwarding resolved* for the future f in the configuration cn at the object o, if there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, such that either $o' \in \pi_2(a(f))$, or one of the following holds:

1. cn assigns f to $p \in PVal$ and there is a container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = p$.

2. cn assigns f to $f' \in FID$ and there is a container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn$ such that $\pi_1(a'(f)) = f'$ and, additionally, o' is forwarding resolved for f' in cn at o.

Definition 10.5 (Extended Object Container)**.** Let cn and cn' be configurations. An object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ *extends* an object container $\mathsf{o}(o', a', u', q'_{in}, q'_{out}) \preceq cn'$ if $o = o'$, $u = u'$, $q_{in} = q'_{in}$, $q_{out} = q'_{out}$, and

1. $a'(\textbf{self}) = a(\textbf{self})$, and for variables x, $a'(x) = a(x)$,

2. if $\pi_1(a'(f)) = v$, then $\pi_1(a(f)) = v$,

3. if $o'' \in \pi_2(a'(f))$, then o'' is forwarding resolved for f in cn at o,

4. if $\pi_1(a(f)) = v$, then cn' assigns v to f, and if additionally $v \in FID$, then o is on the notification path of v in cn, and

5. if $o'' \in \pi_2(a(f))$, then either $o'' \in \pi_2(a'(f))$ or both $o'' \in OID(cn')$ and o is on the notification path of f in cn.

Definition 10.6 (Extended Configuration)**.** A configuration cn *extends* another configuration cn', written $cn \geqslant cn'$, if there is a one-to-one mapping of object containers $obj \preceq cn$ to object containers $obj' \preceq cn'$, such that obj extends obj', and additionally, $\mathtt{graph}(cn) = \mathtt{graph}(cn')$, $\mathsf{t}(cn) = \mathsf{t}(cn')$, and $\mathsf{m}(cn) = \mathsf{m}(cn')$.

Definition 10.7 (Notification Path Traversal)**.** Let cn be a configuration such that o is on the notification path of f by o' in n steps. Then, this notification path *traverses* o_{path} if

1. $n = 0$, $o' = o$, and $o_{path} = o$.

2. $n = 1$, $o' = o$, there is a future message $\mathsf{future}(o, f, v) \preceq cn$, and $o_{path} = o$.

3. $n = 1$, $o' = o$, there is a task $\mathsf{t}(o, l, s) \preceq cn$, and $o_{path} = o$.

4. $n = 2$, $o' = o$, there is a call message $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn$, and $o_{path} = o$.

5. $n = 4$, and there is a call message $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn$, and $o_{path} = o$ or $o_{path} = o'$.

6. $n = n' + 2$, and there is an object $\mathsf{o}(o'', a', u', q_{in}, q_{out}) \preceq cn$ such that $o \in \pi_2(a'(f))$, and o'' is on the notification path of f by o' in n' steps, and either $o_{path} = o''$, or the notification path from o'' to o' traverses o_{path}.

7. $n = 2n' + n''$, and o is on the notification path of f' by o'' in n' steps, cn assigns f to f', and o'' is on the notification path of f by o' in n'' steps, and o_{path} either traverses the path of f' from o to o'', or the path of f from o'' to o'.

Proposition 10.8. *Let cn_1 and cn_2 be WF2 configurations such that $cn_1 \geqslant cn_2$. Then, if o is on the notification path of f by o_2 in n steps in cn_2, o is on the notification path of f by some o_1 in at most n steps in cn_1, such that the notification path from o to o_2 in cn_2 traverses o_1.*

Proof. See Appendix 4. □

Proposition 10.9. *If cn_1 and cn_2 are WF2 configurations such that $cn_1 \geqslant cn_2$ and there are cn_1' and cn_2' such that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'$, and $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2'$, then $cn_1' \geqslant cn_2'$.*

Proof. See Appendix 4. □

We define a binary relation, $\cong_2$, that is a slight weakening of $\equiv_2$, and therefore includes $\equiv_2$, which relates WF2 configurations that, after running Algorithm 2, yield a pair of configurations such that one extends the other.

Definition 10.10 ($\cong_2$)**.** Let $cn_1 \cong_2 cn_2$ if cn_1 and cn_2 are WF2 configurations and there are cn_1' and cn_2' such that it is either the case that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \geqslant cn_2' \rightsquigarrow \mathcal{A}_2(cn_2)$, or $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2' \geqslant cn_1' \rightsquigarrow \mathcal{A}_2(cn_1)$.

Lemma 10.11. *$\cong_2$ is reduction closed.*

Proof. See Appendix 4. □

Lemma 10.12. *$\cong_2$ is context closed.*

Proof. See Appendix 4. □

Proposition 10.13. *$\cong_2$ is a type 2 witness relation.*

Proof. See Appendix 4. □

Lemma 10.14. *Suppose that cn' is WF2, and $cn \geqslant cn'$. Then, cn is WF2 as well, and $cn \simeq_2 cn'$.*

Proof. See Appendix 4. □

10.3 Correctness Result

We obtain a key lemma allowing us to relate transitions in the network-oblivious and the network-aware semantics under $\cong_2$.

Lemma 10.15. *Let* bind $\overline{z}.cn$ *be a WF1 configuration in standard form.*

1. *If* bind $\overline{z}.cn \rightarrow$ bind $\overline{z}'.cn'$, *then for some cn'',* $\texttt{net}(cn) \rightarrow^* cn'' \cong_2 \texttt{net}(cn')$.

2. *If* $\texttt{net}(cn) \to cn''$*, then for some* $\overline{z}'$ *and* cn'*,* bind $\overline{z}.cn \to^*$ bind $\overline{z}'.cn'$ *and* $cn'' \cong_2 \texttt{net}(cn')$.

Proof. See Appendix 4. □

We can now state the main result.

Theorem 10.16 (Correctness of the Type 2 Semantics)**.** *For all well-formed type 1 configurations* bind $\overline{z}.cn$ *in standard form,* bind $\overline{z}.cn \simeq \texttt{net}(cn)$.

Proof. See Appendix 4. □

11 Scheduling

The type 2 semantics is highly nondeterministic. The semantics says nothing about how frequently routing tables are to be exchanged, when messages should be passed between the different queues, when future messages are to be sent, or when, and to where, objects are to be transmitted. Resolving these choices is a crucial tradeoff between management overhead and performance. For instance, if routing tables are exchanged at a very high frequency, routing can be always assumed to be in stable state. This ensures short end-to-end routes, but at the expense of a large management and messaging overhead. This raises the question of how to determine these parameters, something which we address in more detail in related work [29].

Regardless of how, a real implementation needs to resolve these choices. This is tantamount to eliminating nondeterminism from the type 2 semantics, essentially by removing transitions. Thus, in a sense, Theorem 10.16 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

We view a scheduler abstractly as a predicate on histories in the following way. Let a *scheduled execution* be any sequence $\rho = cn_1 \cdots cn_n$ such that $cn_i \to cn_{i+1}$ for all $i : 1 \leq i < n$ where all cn_i are well-formed type 2 configurations. A *scheduler* is a predicate $\mathcal{S}$ on such sequences, with the property that

1. $\mathcal{S}(\langle cn \rangle)$ for all configurations cn, where $\langle cn \rangle$ is the execution consisting only of cn (a scheduler kicks in only when an execution is started), and

2. if $\mathcal{S}(cn_1 \cdots cn_n)$ and $cn_n \to cn_{n+1}$ for some cn_{n+1}, then it holds that $\mathcal{S}(cn_1 \cdots cn_n cn_{n+1})$ for exactly one cn_{n+1}.

Then, a *scheduled type 2 semantics* is a transition system on executions $\rho = cn_1 \cdots cn_n$ such that $\rho \to \rho'$ if and only if $\rho' = cn_1 \cdots cn_n cn_{n+1}$ and $\mathcal{S}(\rho')$.

Define now the barbed simulation preorder $\sqsubseteq$ on executions by requiring the existence of a witness relation $\mathcal{R}$ which satisfies reduction closure, context closure, and barb preservation (where $cn_1 \cdots cn_n \downarrow obs$ whenever $cn_n \downarrow obs$) but not necessarily the converse properties. We immediately obtain from Theorem 10.16 the following corollary:

Corollary 11.1. *For all well-formed type 1 configurations* bind $\overline{z}.cn$ *in standard form,*

$$\langle \mathtt{net}(cn) \rangle \sqsubseteq \langle \mathsf{bind}\ \overline{z}.cn \rangle \ .$$

Proof. It suffices to note that if $\rho = cn_1 \cdots cn_n \to \rho' = cn_1 \cdots cn_{n+1}$, then $cn_n \to cn_{n+1}$, and if $\rho \downarrow obs$, then $cn_n \downarrow obs$ as well. □

12 Concluding Remarks

The contribution of the present paper has been to show that, using location independent routing, it is possible to devise novel and elegant network-based execution models for object-oriented languages with fairly sophisticated features such as futures, and with attractive properties regarding correctness, performance, and scalability.

This paper focuses on correctness. In other ongoing work [29], we study the use of the model presented here adapted in detail to the ABS language core [23], to investigate decentralized runtime adaptability of objects. The language in our work on adaptability is more practical in that expressions and expression evaluation are both fully defined, variable types are declared, and it includes a type system which guarantees that well-typed programs have safety properties that go beyond well-formedness as defined here. Another difference is that the unit of concurrency is a single object rather than a task, which gives a more intuitive programming model that avoids data races. However, as demonstrated by Henrio et al. [19], an object model with concurrent tasks can still be easy to use and be efficiently executed. Our adaptability-oriented network-aware semantics differs from the type 2 semantics in that network and object configurations are completely separate, but can perform labelled transitions, and synchronize on complementary labelled transitions to transfer data between a node and an object. This makes the interfaces between the network-layer and the object-layer much more clear than in the present work. We have also developed a Java-based simulator which implements the adaptability-oriented network-aware semantics and provides various ways of controlling object migration. We report on experimental results on decentralized performance adaptation for load balancing and latency management, with promising results.

The correctness analysis is based on contextual equivalence, similar to other past work mostly belonging to the π-calculus school of process algebra [8, 15, 16]. A closely related precursor is Nomadic Pict [36]. In comparison with that work, we obtain a simpler and in our opinion more elegant correctness treatment, chiefly because our solution obviates the need for locking and consequently preemption, which has well-known detrimental consequences in a bisimulation-oriented setting. Other related works include JoCaml [10], which also uses forward chaining, along with an elaborate mechanism to collapse the forwarding chains. In the Klaim project [1], compilers are implemented and proved correct for several variants of the Klaim language, using the Linda tuple space communication model and a centralized name server to identify local tuple servers. The Oz kernel lan-

guage [37] uses a monotone shared constraint store in the style of concurrent constraint programming. The Oz/K language [26] adds to this a notion of locality with separate failure and mobility semantics, but no real distribution or communication semantics is given (long distance communication is reduced to explicit manipulation of located agents, in the style of the Ambient Calculus [3]). Past correctness analyses for languages with futures have been carried out, e.g., by Caromel et al. [7], who prove a confluence result for their language of asynchronous sequential processes, however without an explicit treatment of distribution, communication, and routing.

Substantial work has been going on in the HATS project on the ABS language and its extensions, for instance towards software product lines [34]. Johnsen et al. [24] have studied an extension of ABS with deployment components that can be used as a reflection mechanism for explicit performance management. The use of reflection allows many aspects of resource control to be brought into to the ABS framework and managed as part of the regular application development process. By contrast, our focus is on algorithmics and the automation of the performance adaptation processes, and for this type of application reflection is not a main concern. The cobox concurrency programming model described by Schäfer [35], closely related to ABS, has a Java implementation which handles message passing to remote objects via RMI, but which does not support object mobility.

Scalability is not fully resolved in the present work. We use a rather naïve distance vector routing scheme based on Bellman-Ford. Distance vector routing has unit stretch but is not compact: Routing tables need to contain on the order of one entry per object identifier in the system. This is no problem for networks of moderate size, but for scalability, other routing schemes are needed, as outlined in the introduction. The Bellman-Ford scheme has other well-known drawbacks that arise in the case of intermittent network partitioning.

Besides more scalable and robust routing, we see two main directions for future work. The first is to examine richer language semantics, specifically with respect to more dynamicity. In ongoing work, we are studying power control: Adding an explicit knob to the network-aware semantics for turning nodes on and off. Further down the line, it is of interest to consider both crash failures and Byzantine failures. The second, parallel, avenue is to study performance adaptation in richer and more realistic settings. In our work on adaptability [29], our only management knob is object migration, and the management objective is to obtain good load balancing combined with good clustering properties. However, a real implementation, in particular when combined with network dynamicity, will have many more management knobs such as buffer size, processor load, and power control. How to effectively control object network performance in such a multi-dimensional setting is a significant future challenge.

References

[1] L. Bettini, V. Bono, R. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In C. Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20583-8.

[2] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: routing on flat labels. *SIGCOMM Comput. Commun. Rev.*, 36 (4):363–374, Aug. 2006.

[3] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64300-5.

[4] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001.

[5] D. Caromel and L. Henrio. *A theory of distributed objects - asynchrony, mobility, groups, components*. Springer, 2005.

[6] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *POPL '04*, pages 123–134. ACM, 2004.

[7] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, Apr. 2009. ISSN 0890-5401.

[8] G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal calculus. *Inf. Comput.*, 201(1), 2005.

[9] 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(2):187–243, 2002.

[10] S. Conchon and F. Le Fessant. Jocaml: mobile agents for objective-caml. In *Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on*, pages 22–29, 1999.

[11] M. Dam and K. Palmskog. Location independent routing in process network overlays. Technical report, KTH Royal Institute of Technology, 2013. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-131378`.

[12] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[13] F. Douglis and J. K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Softw., Pract. Exper.*, 21 (8):757–785, 1991.

[14] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, Jan. 1999. ISSN 0956-7968.

[15] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proc. 23rd ACM Symp. Principles of Programming Languages (POPL)*, pages 372–385. ACM Press, 1996.

[16] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. *Mathematical Structures in Computer Science*, 13(3):371–408, 2003.

[17] D. Havelka, C. Schulte, P. Brand, and S. Haridi. Thread-based mobility in Oz. In P. Roy, editor, *Multiparadigm Programming in Mozart/Oz*, volume 3389 of *Lecture Notes in Computer Science*, pages 137–148. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25079-1.

[18] L. Henrio, M. U. Khan, N. Ranaldo, and E. Zimeo. First class futures: Specification and implementation of update strategies. In *Post-Proceedings Selected Papers From The Coregrid Workshop On Grids, Clouds and P2P Computing*, Aug 2010.

[19] L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In C. Julien and R. De Nicola, editors, *COORDINATION 2013*, LNCS. IFIP International Federation for Information Processing, Springer, June 2013.

[20] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 41–52, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7.

[21] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[22] A. Jeffrey and J. Rathke. Contextual equivalence for higher-order pi-calculus revisited. *Logical Methods in Computer Science*, 1(1), 2005.

[23] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[24] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In F. Arbab and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–204. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35742-8. doi: 10.1007/978-3-642-35743-5_12.

[25] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, Feb. 1988. ISSN 0734-2071.

[26] M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/K: a kernel language for component-based open programming. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 43–52, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8.

[27] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 260–267, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1.

[28] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, Nov. 2006. ISSN 0304-3975.

[29] K. Palmskog, M. Dam, A. Lundblad, and A. Jafari. ABS-NET: Fully decentralized runtime adaptation for distributed objects. In M. Carbone, I. Lanese, A. Lluch Lafuente, and A. Sokolova, editors, Proceedings 6th *Interaction and Concurrency Experience,* Florence, Italy, 6th June 2013, volume 131 of *Electronic Proceedings in Theoretical Computer Science*, pages 85–100. Open Publishing Association, 2013.

[30] A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. Cambridge University Press, Nov. 2011.

[31] N. Ranaldo and E. Zimeo. Analysis of different future objects update strategies in proactive. In *IEEE International Parallel and Distributed Processing Symposium, 2007, IPDPS '07*, pages 1–7, march 2007.

[32] D. Sangiorgi and D. Walker. *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521781779.

[33] D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1): 5:1–5:69, Jan. 2011.

[34] I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

[35] J. Schäfer. *A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems*. PhD thesis, University of Kaiserslautern, 2010.

[36] P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic Pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32 (4):12:1–12:63, Apr. 2010.

[37] G. Smolka. The definition of Kernel Oz. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-59155-9.

[38] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[39] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming abcl/1. *SIGPLAN Not.*, 21(11):258–268, June 1986. ISSN 0362-1340.

Appendix 1: Proofs for Section 5

Proposition 5.2. *The identity relation is a type 1 witness relation. Contextual equivalence is a type 1 witness relation. If $\mathcal{R}$, $\mathcal{R}_1$, $\mathcal{R}_2$ are type 1 witness relations then so is*

1. $\mathcal{R}^{-1}$,
2. $\mathcal{R}^*$, *and*
3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$.

Proof. The identity relation is trivially reduction closed, context closed, and barb preserving, and is its own converse. To prove $\simeq_1$ is a type 1 witness relation, suppose $cn_1 \simeq_1 cn_2$. Then, $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation. If $cn_1 \to cn_1'$, we have cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \mathcal{R} cn_2'$. But then $cn_1' \simeq cn_2'$, and we have shown reduction closure. For context closure, assume $cn_1 cn$ is WF1; then $cn_1 cn \; \mathcal{R} \; cn_2 cn$ and consequently $cn_1 cn \simeq_1 cn_2 cn$. For barb preservation, if $cn_1 \downarrow obs$, then $cn_2 \Downarrow obs$, since $\mathcal{R}$ is barb preserving. The converse arguments are completely symmetric.

For property 1, it suffices to note that $(\mathcal{R}^{-1})^{-1} = \mathcal{R}$. Reflexive, transitive closure in property 2 follows by a straightforward inductive argument. For property 3, if $cn_1 \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2$ then $cn_1 \mathcal{R}_1 cn_{1,2} \mathcal{R}_2 cn_{2,1} \mathcal{R}_1 cn_2$ for some $cn_{1,2}$, $cn_{2,1}$. For reduction closure, assume $cn_1 \to cn_1'$. Then, $cn_{1,2} \to^* cn_{1,2}'$ and $cn_1' \mathcal{R}_1 cn_{1,2}'$. But then, $cn_{2,1} \to^* cn_{2,1}'$ and $cn_{1,2}' \mathcal{R}_2 cn_{2,1}'$. Consequently, $cn_2 \to^* cn_2'$ and $cn_{2,1}' \mathcal{R}_1 cn_2'$, whereby $cn_1' \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2'$. For context closure, assume $cn_1 cn$ is WF1. Then, $cn_{1,2} cn$ is WF1 and $cn_1 cn \; \mathcal{R} \; cn_{1,2} cn$, and so on, yielding that $cn_2 cn$ is WF1 and $cn_1 cn \; \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 \; cn_2 cn$. For barb preservation, assume $cn_1 \downarrow obs$. Then, $cn_{1,2} \to^* cn_{1,2}' \downarrow obs$, allowing us to find $cn_{2,2}'$ such that $cn_{1,2}' \mathcal{R}_1 cn_{2,2}'$, whereby $cn' \to^* cn_{2,2}'' \downarrow obs$, and so on, yielding that $cn_2 \Downarrow obs$, as needed. Again, the converse arguments are symmetric. □

Appendix 2: Proofs for Section 7

Lemma 7.4. *Fix a configuration cn and an object $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$. If o is on the notification path of f by o' in n steps in the configuration cn and $cn \to cn'$, then o is on the notification path of f by o' in at most n steps in cn'.*

Proof. The proof is by induction on n. We follow the case analysis in Definition 7.3.

Case 1: We obtain that $\pi_1(a(f)) \downarrow$, and then, by Proposition 6.2.3, we find $\mathsf{o}(o, a', u', q_{in}', q_{out}') \preceq cn'$ such that $\pi_1(a'(f)) \downarrow$.

Case 2: Either $\mathsf{future}(o, f, v) \preceq cn'$ or we find $\mathsf{o}(o, a', u', q_{in}', q_{out}') \preceq cn'$ such that $\pi_1(a'(f)) \downarrow$.

Case 3: There are two options by Proposition 6.2.4: Either we find a task $\mathsf{t}(o, l', s') \preceq cn'$ with $l'(\mathbf{ret}) = f$, or else we find $\mathsf{o}(o, a', u', q_{in}', q_{out}')$ such that $\pi_1(a'(l(\mathbf{ret}))) \downarrow$.

Case 4: Either $\mathsf{call}(o, o'', f, m, \overline{v}) \preceq cn'$ or we find $\mathsf{t}(o, l, s) \preceq cn'$ with $l(\mathbf{ret}) = f$.

Case 5: Either $\mathsf{call}(o', o, f, m, \overline{v}) \preceq cn'$, or we find $\mathsf{o}(o', a', u', q'_{in}, q'_{out}) \preceq cn'$ such that $\mathsf{t}(o', l', s') \preceq cn'$ with $l'(\mathbf{ret}) = f$, and $o \in \pi_2(a'(f))$.

Case 6: We find an object $obj'' = \mathsf{o}(o'', a'', u'', q''_{in}, q''_{out}) \preceq cn$ such that o'' is on the notification path of f by o' in $n-1$ steps in cn. By Proposition 6.2.3, we find the derivative $obj''' = \mathsf{o}(o'', a''', u'', q'''_{in}, q'''_{out}) \preceq cn'$ of obj'', and by the induction hypothesis o'' is also on the notification path of f by o' in cn', now in some $n'' \leq n-2$ steps. By inspection of the rules we see that either $\pi_2(a'''(f))$ is a suffix of $\pi_2(a''(f))$, or else there is a message $\mathsf{future}(o, f, \pi_1(a''(f))) \preceq cn'$. In either case we can conclude.

Case 7: We find an object $obj_1 = \mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn$ such that o_1 is on the notification path of f' by o'' in n' steps in cn, and an object $obj'' = \mathsf{o}(o'', a'', u'', q''_{in}, q''_{out}) \preceq cn$ such that o'' is on the notification path of f by o' in n'' steps, with $n = 2n' + n''$, where f' is assigned to f in cn. In the next step, by the induction hypothesis, o'' can be on the notification path of of f by o' in the same or fewer steps, and the length of the notification path of f' for o_1 does not increase as in the previous case. In either case we can conclude. □

Lemma 7.6. *Let cn be a configuration. Then, the following holds:*

1. *If cn is a type 2 initial configuration, then cn is WF2.*
2. *If cn is WF2 and $cn \to cn'$, then cn' is WF2.*
3. *If cn is type 2 reachable, then cn is WF2.*

Proof. Property 1 follows straightforwardly from Definition 6.3 and Definition 7.5. For property 2, we consider each transition rule in turn.

OID Uniqueness: In all rules except NEW-2 there is a one-to-one correspondence between object container occurrences in cn and object container occurrences in cn'. This is sufficient to conclude. For NEW-2, it is sufficient to note that o' is a freshly generated OID.

Task-Object Existence, Object-Node Existence: The properties follow since neither nodes nor objects are ever removed. In the first instance, objects can only be created when the node is present, and in the second instance tasks can only be created when the object is present.

Buffer Cleanliness: We check that only object-bound messages enter in- and out-queues. This concerns the rules MSG-RCV, CALL-SEND-2, FUT-SEND, MSG-DELAY-1, and MSG-DELAY-2 only. The check is routine.

Local Routing Consistency: Easily proved by case analysis on rules.

Call Uniqueness: In all rules except CALL-SEND-2, call messages are the same in cn and cn', or present in cn and removed in cn'. This is sufficient to conclude. For CALL-SEND-2, it is sufficient to note that f is a freshly generated FID.

Future Uniqueness: We only need to consider rules which assign a non-$\perp$ value to futures. This happens in rules FUT-SEND, FUT-RCV and RET-2. The

former two rules are immediate, and for RET-2 we use the assumption that cn satisfies 7.5.5.

Single Writer: Again, we only need to consider the rules FUT-SEND, FUT-RCV and RET-2. Since for the former two rules, cn assigns v to f if and only if cn' does so, only RET-2 remains, which is immediate.

External OID: Assuming fresh identifiers for objects are never equal to ext, the check is routine.

Future Liveness: Let $\mathsf{o}(o, a', u', q'_{in}, q'_{out})$ be the derivative of $\mathsf{o}(o, a, u, q_{in}, q_{out})$ in cn, and assume that f is active for o for the configuration cn'. Either $a'(f) \downarrow$, or else a call message $\mathsf{call}o', o, f', m, \overline{v}$ is in transit in cn' and f occurs in $\overline{v}$. We proceed by cases on the transition rule leading to cn'. Any rule that does not directly affect any of the conditions in Definition 7.2 or Definition 7.3 immediately allows to conclude that f is active for o also in cn. By the induction hypothesis, we can conclude that o is on the notification path of f in cn, and then o is on the notification path of f also in cn', since the only exception in Lemma 7.4 is when the environment of o is updated. For the remaining rules, there are the following cases to consider:

CALL-SEND-2: Assume first that o is the sending object. Either f is the newly introduced future in which case o is on the notification path from f according to 7.3.5, since a call message is in transit from o to o' with return future f. If f is another future which is active for o in cn' then f is also active for o in cn. By the induction hypothesis, o is on the notification path from f in cn. Then o is also on the notification path from f in cn' by Lemma 7.4. On the other hand if o is not the sending object the case is immediately closed by the induction hypothesis, as the "pending" relation transfers from cn' to cn. We then apply the IH to conclude that o is on the notification path from f in cn, and then we use Lemma 7.4 to conclude that this also applies to cn'.

CALL-RCV-2: Assume first that o is the object receiving the call, and that f is the future of the call. Then o is on the notification path from f by Definition 7.3.2. Another option is that f is a future in $\overline{v}$ (referring to the transition rule in Figure 9). Then f is active for o in cn as well, by Definition 7.2. If f is some other future the case is completed by the IH as above.

FUT-SEND: Follows from the induction hypothesis and Lemma 7.4, as in the case for CALL-SEND-2.

FUT-RCV: If f is active for o for the configuration cn' then f is active for o also for cn, and f is not the received future. But then, the result follows by the IH and Lemma 7.4.

RET-2: If f is active for o for cn', then f is not the return future and we again complete by the induction hypothesis. Finally, property 3 holds by property 1 and repeated application of property 2. □

Appendix 3: Proofs for Section 9

Proposition 9.3. *Algorithm 1 terminates.*

Proof. In each iteration of the outermost loop of Algorithm 1, exactly one message is enqueued on each proper link, and at least one message is dequeued from all link queues. MSG-RCV, MSG-DELAY-1, and OBJ-RCV cause messages to leave the link queues, except for external messages, which are moved to the self-loop queues. If the link queues contain only routing table messages, the algorithm terminates in that iteration. If not, there must be object messages or routable call messages in some link queue. Since no new object messages are enqueued, there must some number of iterations n_0 after which all object messages have been received via OBJ-RCV and the associated object OIDs o registered on some node u so that $t(o) = (u, 0)$.

Let m_0 be the size of the largest link queue at the point which there are no object messages in transit. After $n_0 + m_0 + 1$ iterations, each node u has received at least one table update from each of its neighbors u', and the last table update applied to u has $t(o) = 0$. As a result, at point $n_0 + m_0 + 1$, each node u has $t(o) = (u', 1)$ whenever the u' is the host of o and the minimal length path from u to u' has length 1. The entry of the routing table of u for o will not change from that point onwards. We say that those entries are *stable*. Proceeding, let m_1 be the size of the largest link queue at point $n_0 + m_0 + 1$. After $n_0 + m_0 + 1 + m_1 + 1$ iterations, each routing table entry with length 2 (or less) will be stable. In the limit, each entry will be stable. It follows that Algorithm 1 must terminate, since, once routing has stabilized, rule MSG-ROUTE can only be applied a finite number of times before a routable message will be delivered. There is no chance of routable messages getting stuck in self-loop queues, since they are continuously shuffled using MSG-DELAY-3.

The only detail remaining to be checked is that a message can always be read from a link. Table and object messages can always be delivered, and call messages can also always be delivered, if nothing else to the self-loop link, in which case the routing table is not up-to-date or the message is external. This is the only case where MSG-DELAY-1 is used. This completes the argument. □

Lemma 9.8. $\equiv_1$ *is reduction closed.*

Proof. Suppose $cn_1 \equiv_1 cn_2$, where cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_1 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, OBJ-RCV, MSG-DELAY-3, and OBJ-REG, we take $cn_2' = cn_2$, since then, the stable form is unaffected, i.e., $cn_1 \equiv_1 cn_1'$, by Proposition 9.5.

The remaining cases include the rules for sequential control, MSG-SEND, MSG-DELAY-2, CALL-SEND-2, CALL-RCV-2, FUT-SEND, FUT-RCV, RET-2, GET-2, NEW-2, and OBJ-SEND. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_1(cn_2) \rightsquigarrow cn_2''$. By

Proposition 9.5, there is a task $\mathsf{t}(o, l, x = e; s)$ and an object $\mathsf{o}(o, a, u, q'_{in}, q_{out})$ in cn''_2. Hence, it is possible to perform a transition

$$cn' \ \mathsf{o}(o, a, u, q'_{in}, q_{out}) \ \mathsf{t}(o, l, x = e; s) \to cn' \ \mathsf{o}(o, a[v/x], u, q'_{in}, q_{out}) \ \mathsf{t}(o, l, s)$$

and we have

$$cn \ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out}) \ \mathsf{t}(o, l, s) \equiv_1 cn' \ \mathsf{o}(o, a[v/x], u, q'_{in}, q_{out}) \ \mathsf{t}(o, l, s)$$

as needed, setting cn'_2 to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} & cn \ \mathsf{o}(o, a, u, q_{in}, q_{out}) \ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ \to \quad & cn \ \mathsf{o}(o, a', u, q_{in}, \mathtt{enq}(msg, q_{out})) \ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$, $f = \mathtt{newf}(u)$, $msg = \mathsf{call}(o', o, f, m, \overline{v})$, $o' = [\![e_1]\!]_{(a,l)}$, and $a' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a))$. Consider cn''_2 such that $\mathcal{A}_1(cn_2) \rightsquigarrow cn''_2$. By Proposition 9.5, there is a task $\mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s)$ and an object $\mathsf{o}(o, a, u, q'_{in}, q_{out})$ in cn''_2. Hence, it is possible to perform a transition

$$\begin{aligned} & cn' \ \mathsf{o}(o, a, u, q'_{in}, q_{out}) \ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ \to \quad & cn' \ \mathsf{o}(o, a', u, q'_{in}, \mathtt{enq}(msg, q_{out})) \ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

and we have

$$\begin{aligned} & cn \ \mathsf{o}(o, a', u, q_{in}, \mathtt{enq}(msg, q_{out})) \ \mathsf{t}(o, l[f/x], s) \\ \equiv_1 \quad & cn' \ \mathsf{o}(o, a', u, q'_{in}, \mathtt{enq}(msg, q_{out})) \ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

setting cn'_2 to the right-hand side. The remaining cases are proved in a similar straightforward way, by mimicking the transition after applying Algorithm 1. □

Lemma 9.9. *$\equiv_1$ is context closed.*

Proof. Assume $cn_1 \equiv_1 cn_2$ and $cn_1 \ cn$ is WF2. We first show that $cn_2 \ cn$ is WF2 as well.

OID Uniqueness: If $obj_1, obj_2 \preceq cn_2 \ cn$ either $obj_1, obj_2 \preceq cn_2$, $obj_1, obj_2 \preceq cn$, or (wlog) $obj_1 \preceq cn_2$ and $obj_2 \preceq cn$. In either case, since $OID(cn_1) = OID(cn_2)$ by the definition of $\equiv_1$, the result follows.

Task-Object Existence: If $tsk \preceq cn_2 \ cn$ either $tsk \preceq cn_2$ or $tsk \preceq cn$. In the former case, if $tsk = Tsk(o, l, s)$ then, since $cn_1 \equiv_1 cn_2$ and so cn_2 is WF2, we find $obj \preceq cn_2$ with $OID(obj) = \{o\}$. Otherwise, since $cn_1 \ cn \equiv_1 cn_2 \ cn$ we find $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1 \ cn$ and hence by definition of $\equiv_1$, $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2 \ cn$ as well.

Object-Node Existence: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2 \ cn$, then either the container is in cn_2, which is WF2 and thus has a node u, or it is in cn, which means node u is in cn_1, which has the same network as cn_2.

Buffer Cleanliness: If $obj \preceq cn_2 \ cn$ either $obj \preceq cn_2$ or $obj \preceq cn$. In the former case we are done since cn_2 is WF2. In the latter case, we get $obj \preceq cn_1 \ cn$ and $cn_1 \ cn$ is WF2, which is sufficient.

Local Routing Consistency: This is immediate since cn contains only object and task containers.

Call Uniqueness: If $call \preceq cn_2$ and $call' \preceq cn$, and there is a future clash, there must be a clash between $call'$ and some message in cn_1. But this is ruled out since $cn_1\ cn$ is WF2.

Future Uniqueness: Assume cn_2 assigns v_1 to f and cn assigns v_2 to f and $v_1 \neq v_2$. Then, since $cn_1 \equiv_2 cn_2$, cn_1 assigns v_1 to f as well, violating WF2.

Single Writer: Assume $tsk_1, tsk_2 \preceq cn_2\ cn$, with associated future identifiers f_1 and f_2. Clearly, $f_1 \neq f_2$, and both are unassigned, or else WF2 would be violated for cn_2 or $cn_1\ cn$. Assume $call \preceq cn_2\ cn$ with f; then f is distinct from f_1 and f_2, or there would have been a clash in cn_2 or $cn_1\ cn$, ruled out by WF2.

External OID: cn_2 is WF2, so $ext \notin OID(cn_2)$. Also, $cn_1\ cn$ is WF2, so $ext \notin OID(cn)$. Hence $ext \notin OID(cn_2\ cn)$. Also, if t is a routing table in $cn_2\ cn$ it is a routing table in cn_2 and cn_2 is WF2. Then $ext \notin \mathtt{dom}(t)$, as required.

Future Liveness: Assume f is active for o in $cn_2\ cn$; then f is active for o in $cn_1\ cn$, and thus on the notification path of f there. Hence, it is also on the notification path of o in $cn_2\ cn$.

We next need to show that $cn_1\ cn \equiv_1 cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_1(cn_1\ cn) \rightsquigarrow cn_1'$ and $\mathcal{A}_1(cn_2\ cn) \rightsquigarrow cn_2'$. It suffices to prove $cn_1'\ \mathcal{R}_1\ cn_2'$. We check the requirements:

$\mathtt{graph}(cn_1') = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn_2')$.

$\mathsf{t}(cn_1') = \mathsf{t}(cn_1\ cn) = \mathsf{t}(cn_1) \cup \mathsf{t}(cn) = \mathsf{t}(cn_2) \cup \mathsf{t}(cn) = \mathsf{t}(cn_2\ cn) = \mathsf{t}(cn_2')$.

$\mathsf{o}(cn_1') \cong_1 \mathsf{o}_1(cn_1\ cn) = \mathsf{o}_1(cn_1) \cup \mathsf{o}_1(cn) \cong_1 \mathsf{o}_1(cn_2) \cup \mathsf{o}_1(cn) = \mathsf{o}_1(cn_1\ cn) \cong_1 \mathsf{o}(cn_2')$.

$\mathsf{m}(cn_1') = \mathsf{m}_1(cn_1\ cn) = \mathsf{m}_1(cn_1) \cup \mathsf{m}_1(cn) = \mathsf{m}_1(cn_2) \cup \mathsf{m}_1(cn) = \mathsf{m}_1(cn_2\ cn) = \mathsf{m}(cn_2')$. □

Proposition 9.10. *$\equiv_1$ is a type 2 witness relation.*

Proof. Reduction closure and its converse follows from Lemma 9.8, and context closure and its converse follows from Lemma 9.9. Hence, it suffices to show barb preservation in both directions. Suppose $cn_1 \equiv_1 cn_2$ and $cn_1 \downarrow obs$. Then, there is a call message msg with destination ext at the head of some self-loop queue in cn_1. After running Algorithm 1 on cn_2, we will have external link messages, with msg in some link queue. Using MSG-DELAY-1 and MSG-DELAY-3, msg can then be brought to the head of some self-loop queue. Hence, $cn_2 \Downarrow obs$. The proof of converse barb preservation is symmetric. □

Proposition 9.12. *Algorithm 2 terminates.*

Proof. Routing is stable after each run of Algorithm 1, and none of the rules applied in the outermost loop in the first outermost loop affect routing.

Thus, one of MSG-SEND or MSG-DELAY-2 will be enabled whenever the output outqueue is nonempty, causing output queue size to decrease by one. By Buffer Cleanliness, one of CALL-RCV-2 or FUT-RCV will be applicable if the object in-queue is nonempty, decreasing in-queue size by one. Thus, when the inner while loop is reached, each nonempty inqueue has decreased in size by one, and each out-queue may have increased in size by one if the in-queue head position contains a delayed message.

Sending future messages may cause out-queues to increase in size. Each application of FUT-SEND causes a forwarding list to decrease in length by one. Thus, termination of the inner while loop is clear. We need to argue that the outer while loop also terminates.

We first show that, eventually, no forwarding list is incremented. Only two rules can cause forwarding lists to increase in size, namely CALL-SEND-2 and CALL-RCV-2. Of these, CALL-SEND-2 is never used in either Algorithm 1 or Algorithm 2. Each application of CALL-RCV-2 consumes one call message, and none of the rules cause new call messages to be created. Thus, eventually, CALL-RCV-2 is never applied, and from that point onward forwarding lists are either emptied completely by the inner loop, or they remain untouched, since their corresponding future is undefined. Futures can become instantiated by FUT-RCV, but again, this can only happen a bounded number of times. Moreover, the only rule causing futures to be created is MSG-SEND, so the supply of futures to consider is fixed. Consequently, eventually, each future either remains uninstantiated forever, or else the corresponding forwarding list is empty. From that point onward, no FUT-SEND is enabled, and the innermost loop terminates trivially in all future iterations. In this situation, since MSG-SEND, CALL-RCV-2, and FUT-RCV all consume messages from a bounded resource (the set of messages in transit), if the outermost loop fails to terminate the only option is that, from some point onwards, only MSG-DELAY-2 is applied. From this point onward, since routing is stable, all messages will eventually be delivered.

Termination of the final loop is trivial. Observe that Algorithm 2 does not rely on routing to move the object towards u. For the algorithm, it is sufficient to establish that *some* good direction exists, and this is clearly the case as the network is stable and connected. □

Proposition 9.14. *Suppose cn is WF2. If $\mathcal{A}_2(cn) \rightsquigarrow cn'$, then the following holds:*

1. *cn' is reachable from cn.*
2. *cn' is in normal form.*
3. $\mathtt{graph}(cn) = \mathtt{graph}(cn')$.
4. $\mathsf{t}_2(cn) = \mathsf{t}(cn')$.
5. $\mathsf{o}_2(cn) = \mathsf{o}(cn')$.
6. $\mathsf{m}_2(cn) = \mathsf{m}(cn')$.

Proof. Property 1 is immediate. Property 3 follows from property 2 of Proposition 9.5.

For property 4, observe first that the function t_2 is invariant under transitions used in Algorithm 2. On termination of Algorithm 2, only external messages are in transit, and since no rule cause a task to be modified, property 4 follows.

For property 5, let $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}(cn')$. We show that it is then the case that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. From the definition of Algorithm 2, $q_{in,2} = q_{out,2} = \varepsilon$. Also, $u_2 = u$. We know that there is an object container $\mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$, since there is a one-to-one correspondence between object containers in pre- and poststate for each transition used in Algorithm 2. We also know that $a'(x) = a_2(x)$ for all x. Suppose finally that an object container $obj = \mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ exists in cn with $a_1(f) = (v, \overline{o})$. Let $\mathsf{o}(o_1, a_1', u_1', q_{in,1}', q_{out,1}')$ be the derivative of obj in cn'. Then $\pi_1(a_1'(f)) = v$ as well, by Proposition 7.7. We know by Future Uniqueness that $a_2(f) = (v', \overline{o'})$ implies $v' = v$. It remains to show that $\pi_1(a_2(f)) \neq \bot$. Assume not. We have that o_2 is on the notification path of f in n steps for some nonnegative integer n. We proceed by induction on the notification path relation:

Case 1: $n = 0$, and $\pi_1(a_2(f)) = v \neq \bot$. This yields a contradiction.

Case 2: $n = 1$, and there is future message $\mathsf{future}(o_2, f, v') \preceq cn$. This yields a contradiction, since the only queued messages in cn' are external.

Case 3: $n = 1$, and there is a task $\mathsf{t}(o_2, l_2, s_2) \preceq cn'$ with $l_2(\textbf{ret}) = f$. Then, $\pi_1(a_1'(f)) = \bot$ by well-formedness, which is a contradiction.

Case 4: $n = 2$, and there is a call message $\mathsf{call}(o_2, o'', f, m, \overline{v}) \preceq cn'$. This again contradicts the assumption of having only external messages.

Case 5: $n = 4$, and there is a call message $\mathsf{call}(o', o_2, f, m, \overline{v}) \preceq cn'$ such that $o' \in OID(cn')$. This again contradicts the assumption of having only external messages.

Case 6: $n = n' + 2$ and there is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn'$ such that $o_2 \in \pi_2(a(f))$, and o is on the notification path from f in n' steps. Then, if we have $\pi_1(a(f)) = v \neq \bot$, cn' is not in normal form, a contradiction; alternatively, we conclude by the IH.

Case 7: $n = 2n' + n''$ and there is an object container $\mathsf{o}(o'', a, u, q_{in}, q_{out}) \preceq cn'$ such that o is on the notification path of f' by o'' in n' steps, f' is assigned to f, and there is an object container $\mathsf{o}(o', a', u', q_{in}, q_{out}) \preceq cn'$ such that o' is on the notification path of f in n'' steps. By the IH, we have $\pi_2(a'(f)) = v \neq \bot$ and thus $\pi_2(a(f)) = v$. Since f will have been forwarded down to o_2, v will have been forwarded down the same chain.

We can thus conclude that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. Conversely, assume that $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq \mathsf{o}_2(cn)$. Object o_2 has exactly one derivative in cn', by well-formedness. That object has empty queues, the same NID as in cn, preserves assignments to variables, and has $\pi_1(a_2(f))$ assigned to a non-$\bot$ value if and only if some object in cn' has so, by the above argument.

For 9.14.6, the property holds as it does so already for Algorithm 1.

We finally need to prove property 2. Property 9.13.1 is trivial, as each run of Algorithm 2 ends with a run of Algorithm 1, and Algorithm 1 ensures that cn' has stable routing. Property 9.13.2 holds since Algorithm 1 ensures almost empty link queues, and since on termination, Algorithm 2 ensures empty object queues. For 9.13.3, if $obj = \mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ satisfies the properties defining o_2 above then, referring to those conditions, $u' = u'' = u$, $a' = a$, $q_{in} = \varepsilon = q_{out}$, and $obj \preceq cn'$ as needed to be shown. For 9.13.4, the result follows since only external messages are in transit in cn'. □

Lemma 9.18. *$\equiv_2$ is reduction closed.*

Proof. Suppose $cn_1 \equiv_2 cn_2$, where cn_1 and cn_2 are WF2. Assume $cn_1 \to cn_1'$; we need to find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \equiv_2 cn_2'$. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1.

For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, we take $cn_2' = cn_2$, since then, the normal form is unaffected, i.e., $cn_1 \equiv_2 cn_1'$, by Proposition 9.14.

The remaining cases include the rules for sequential control, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$$

where $[\![e]\!]_{(a,l)} = v$ and $x \in \mathtt{dom}(a)$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 9.14, there is a task $\mathsf{t}(o, l, x = e; s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2''. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e; s) \to cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

and we have

$$cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s) \equiv_2 cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

as needed, setting cn_2' to the right-hand side.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad \to \quad cn\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a)), u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$, $f = \mathtt{newf}(u)$, $msg = \mathsf{call}(o', o, f, m, \overline{v})$, and $o' = [\![e_1]\!]_{(a,l)}$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 9.14, there is a task $\mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' with $[\![\overline{e_2}]\!]_{(a',l)} = \overline{v}$ and $[\![e_1]\!]_{(a',l)} = o'$. Hence, it is possible to perform a transition

$$\begin{aligned} &cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad \to \quad cn'\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a')), u', \varepsilon, \mathtt{enq}(msg, \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

and we have

$$\begin{aligned} & cn\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a)), u, q_{in}, \mathtt{enq}(msg, q_{out}))\ \mathsf{t}(o, l[f/x], s) \\ \equiv_2 \quad & cn'\ \mathsf{o}(o, \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a')), u', \varepsilon, \mathtt{enq}(msg, \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

as needed, setting cn_2' to the right-hand side.

RET-2: Consider a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \to cn\ \mathsf{o}(o, a[v/f], u, q_{in}, q_{out})$$

where $v = [\![e]\!]_{(a,l)}$ and $f = l(\mathbf{ret})$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 9.14, there is a task $\mathsf{t}(o, l, \mathbf{return}\ e; s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a',l)} = v$. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \to cn'\ \mathsf{o}(o, a'[v/f], u', \varepsilon, \varepsilon)$$

and we have

$$cn\ \mathsf{o}(o, a[v/f], u, q_{in}, q_{out}) \equiv_2 cn'\ \mathsf{o}(o, a'[v/f], u', \varepsilon, \varepsilon)$$

as needed, setting cn_2' to the right-hand side.

GET-2: Consider a transition of the form

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e.\mathbf{get}; s) \to cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s)$$

where $[\![e]\!]_{(a,l)} = f$ and $\pi_1(a(f)) = v$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 9.14, there is a task $\mathsf{t}(o, l, x = e.\mathbf{get}; s)$ and an object $\mathsf{o}(o, a', u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a',l)} = f$ and $\pi_1(a'(f)) = v$. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e.\mathbf{get}; s) \to cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

and we have

$$cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s) \equiv_2 cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

as needed, setting cn_2' to the right-hand side.

NEW-2: Consider a transition of the form

$$\begin{aligned} & cn\ \mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ \to \quad & cn\ \mathsf{o}(o, a', u, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u, \varepsilon, \varepsilon) \end{aligned}$$

where $o' = \mathtt{newo}(u)$, $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$, $a' = \mathtt{fw}(\overline{v}, o', a)$, and $a'' = \mathtt{init}(\overline{v}, \mathtt{init}(C, \overline{v}, o'))$. Consider cn_2'' such that $\mathcal{A}_2(cn_2) \rightsquigarrow cn_2''$. By Proposition 9.14, there is a task $\mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s)$ and an object $\mathsf{o}(o, a_1, u', \varepsilon, \varepsilon)$ in cn_2'' where $[\![e]\!]_{(a_1,l)} = \overline{v}$ and $\pi_1(a_1(f)) = v$. Hence, it is possible to perform a transition

$$\begin{aligned} & cn'\ \mathsf{o}(o, a_1, u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ \to \quad & cn'\ \mathsf{o}(o, a_1', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u', \varepsilon, \varepsilon) \end{aligned}$$

and we have

$$\begin{aligned} & cn\ \mathsf{o}(o, a', u, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u, \varepsilon, \varepsilon) \\ \equiv_2 \quad & cn'\ \mathsf{o}(o, a'_1, u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a'', u', \varepsilon, \varepsilon) \end{aligned}$$

as needed, setting cn'_2 to the right-hand side. □

Lemma 9.19. *$\equiv_2$ is context closed.*

Proof. The proof follows the proof of Lemma 9.9. Assume $cn_1 \equiv_2 cn_2$ and $cn_1\ cn$ is WF2. We first show that $cn_2\ cn$ is WF2 as well.

OID Uniqueness: If $obj_1, obj_2 \preceq cn_2\ cn$ either $obj_1, obj_2 \preceq cn_2$, $obj_1, obj_2 \preceq cn$, or (wlog) $obj_1 \preceq cn_2$ and $obj_2 \preceq cn$. In either case, since $OID(cn_1) = OID(cn_2)$ by the definition of $\equiv_2$, the result follows.

Task-Object Existence: If $tsk \preceq cn_2\ cn$ either $tsk \preceq cn_2$ or $tsk \preceq cn$. In the former case, if $tsk = \mathsf{t}(o, l, s)$ then, since $cn_1 \equiv_2 cn_2$ and so cn_2 is WF2, we find $obj \preceq cn_2$ with OID o. Otherwise, since $cn_1\ cn \equiv_2 cn_2\ cn$ we find $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1\ cn$ and hence by definition of $\equiv_2$, $\mathsf{o}(o, a', u', q'_{in}, q'_{out}) \preceq cn_2\ cn$ as well.

Object-Node Existence: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2 cn$, then either the container is in cn_2, which is WF2 and thus has a node u, or it is in cn, which means node u is in cn_1, which has the same network as cn_2.

Buffer Cleanliness: If $obj \preceq cn_2\ cn$ either $obj \preceq cn_2$ or $obj \preceq cn$. In the former case we are done since cn_2 is WF2. In the latter case, we get $obj \preceq cn_1\ cn$ and $cn_1\ cn$ is WF2, which is sufficient.

Local Routing Consistency: This is immediate since cn contains only object and task containers.

Call Uniqueness: If $call \preceq cn_2$ and $call' \preceq cn$, and there is a future clash, there must be a clash between $call'$ and some message in cn_1 or a task in cn_1. But this is ruled out since $cn_1\ cn$ is WF2.

Future Uniqueness: Assume cn_2 assigns v_1 to f and cn assigns v_2 to f and $v_1 \neq v_2$. Then, since $cn_1 \equiv_2 cn_2$, cn_1 assigns v_1 to f as well, violating WF2.

Single Writer: Assume $tsk_1, tsk_2 \preceq cn_2\ cn$, with associated future identifiers f_1 and f_2. Clearly, $f_1 \neq f_2$, and both are unassigned, or else WF2 would be violated for cn_2 or $cn_1\ cn$. Assume $call \preceq cn_2\ cn$ with f; then f is distinct from f_1 and f_2, or there would have been a clash in cn_2 or $cn_1\ cn$, ruled out by WF2.

External OID: If $ext \in OID(cn_2\ cn)$, then $ext \in OID(cn)$, which is ruled out by WF2; the second requirement is immediate since cn_2 is WF2 and cn contains no nodes.

Future Liveness: Assume f is active for o in $cn_2\ cn$; then f is active for o in $cn_1\ cn$, and thus on the notification path of f there. Hence, it is also on the notification path of o in $cn_2\ cn$.

We next need to show that $cn_1\ cn\ \equiv_2 cn_2\ cn$. The WF2 property is immediate. Suppose $\mathcal{A}_2(cn_1\ cn) \rightsquigarrow cn'_1$ and $\mathcal{A}_2(cn_2\ cn) \rightsquigarrow cn'_2$. It suffices to prove $cn'_1\ \mathcal{R}_2\ cn'_2$. We check the requirements:

$\mathtt{graph}(cn'_1) = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn'_2)$.

Since $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, we have $\mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn)$. Hence, $\mathsf{t}(cn'_1) = \mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn) = \mathsf{t}(cn'_2)$.

Since $\mathsf{o}_2(cn_1) = \mathsf{o}_2(cn_2)$, we have $\mathsf{o}_2(cn_1\ cn) = \mathsf{o}_2(cn_2\ cn)$. Hence, $\mathsf{o}(cn'_1) = \mathsf{o}_2(cn_1\ cn) = \mathsf{o}_2(cn_2\ cn) = \mathsf{o}(cn'_2)$.

$\mathsf{m}(cn'_1) = \mathsf{m}_2(cn_1\ cn) = \mathsf{m}_2(cn_1) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2\ cn) = \mathsf{m}(cn'_2)$. □

Appendix 4: Proofs for Section 10

Proposition 10.3. *If* bind $\overline{z}.cn$ *is a WF1 configuration in standard form, then* $\mathtt{net}(cn)$ *is WF2.*

Proof. We consider the WF2 conditions in turn. OID Uniqueness and Task-Object Existence follows from the respective WF1 conditions and from how the name representation map is defined. Object-Node Existence holds since all objects are placed on the node u_0, which exists by the definition of cn_{graph}. Buffer Cleanliness holds since all object containers in $\mathtt{net}(cn)$ have empty queues. Local Routing Consistency follows from how routing tables in cn_{graph} are defined. Call Uniqueness follows from the corresponding WF1 condition and from how the name representation map is defined. Future Uniqueness follows the definition of `oenvmap`, how object environments are represented, and from Proposition 10.1. Single Writer follows the corresponding WF1 condition and from how the name mapping representation map is defined. External OID follows from the corresponding WF1 condition and from how routing tables are defined.

Consider finally Future Liveness. Assume $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq \mathtt{net}(cn)$. Note that by the way we define future mappings in object environments, we have that, whenever f is active for o in $\mathtt{net}(cn)$, $a(f) \downarrow$ holds. Suppose therefore that $a(f) \downarrow$. Then, there must be some future container for the FID corresponding to f in bind $\overline{z}.cn$ with lifted value $v_\perp$. If $v_\perp \neq \perp$, then $a(f) = (v, \varepsilon)$ for some value v, which means that o is trivially on the notification path of f in $\mathtt{net}(cn)$. If $v_\perp = \perp$, there is, by Future Existence, some task or some call message associated with f for some object o' in $\mathtt{net}(cn)$, and, additionally, o' will have o in its forwarding list for f; hence o is on the notification path of f in this case also.

Suppose finally that $\mathtt{net}(cn)$ assigns f to f', and o is on the notification path of f'. There must then be a corresponding assignment in bind $\overline{z}.cn$, which by Future Existence yields that there is future container for the future corresponding to f with lifted value $v_\perp$. If $v_\perp \neq \perp$, then $a(f) = (v, \varepsilon)$ for some value v, which means that o is trivially on the notification path of f. If $v_\perp = \perp$, there is again, by Future Existence, some task or some call message associated with f for some object o' in $\mathtt{net}(cn)$, and, additionally, o' will have o in its forwarding list for f; hence o is on the notification path of f in this case also. □

Proposition 10.8. *Let cn_1 and cn_2 be WF2 configurations such that $cn_1 \geqslant cn_2$. Then, if o is on the notification path of f by o_2 in n steps in cn_2, o is on the notification path of f by some o_1 in at most n steps in cn_1, such that the notification path from o to o_2 in cn_2 traverses o_1.*

Proof. Suppose $\mathsf{o}(o, a_1, u, q_{in}, q_{out}) \preceq cn_1$ and $\mathsf{o}(o, a_2, u, q_{in}, q_{out}) \preceq cn_2$. The proof is by induction on the notification path relation.

Case 1: $n = 0$, $o_2 = o$, and $\pi_1(a_2(f)) = v$. Then, we have $\pi_1(a_1(f)) = v$, and we set $n = 0$ and $o_1 = o$.

Case 2: $n = 1$, $o_2 = o$, and there is a message $\mathsf{future}(o, f, v) \preceq cn_2$. Since messages are unaffected by extension, the same message is in cn_1, and we set $n = 1$ and $o_1 = o$.

Case 3: $n = 1$, $o_2 = o$, and there is a task $\mathsf{t}(o, l, s) \preceq cn_2$, such that $l(\mathbf{ret}) = f$. Since task containers are unaffected by extension, the same container is in cn_1, and we set $n = 1$ and $o_1 = o$.

Case 4: $n = 2$, $o_2 = o$, and there is a call message $call(o, o'', f, m, \overline{v}) \preceq cn_2$. Then, the same message is in cn_1, and we set $n = 2$ and $o_1 = o$.
Case 5: $n = 4$ and there is a message $\mathsf{call}(o_2, o, f, m, \overline{v}) \preceq cn_2$. Then, the same message is in cn_1, and we set $n = 3$ and $o_1 = o_2$.

Case 6: $n = n'_2 + 2$, and there is an object $\mathsf{o}(o', a'_2, u', q'_{in}, q'_{out}) \preceq cn_2$ such that $o \in \pi_2(a'_2(f))$, and o' is on the notification path of f by o_2 in n'_2 steps in cn_2. As induction hypothesis, we have that o' is on the notification path of f by some o'_1 in some number n'_1 steps ($n'_1 \leq n'_2$) in cn_1. Consider the object $\mathsf{o}(o', a'_1, u', q'_{in}, q'_{out}) \preceq cn_1$; by the definition of extended object container, we have that o is forwarding resolved for f in cn_1 at o'. This means that either $o \in \pi_2(a'_1(f))$ or $\pi_1(a_1(f)) = v$, where cn_1 assigns v to f. In the first case, we set $n = n'_1 + 2$ and $o_1 = o'_1$. In the second case, we set $n = 0$ and $o_1 = o$.

Case 7: $n = 2n'_2 + n''_2$, and o is on the notification path of f' by o'_2 in n'_2 steps in cn_2, cn_2 assigns f to f', and o'_2 is on the notification path of f by o_2 in n''_2 steps in cn_2. Since no configuration future assignments are changed by extension, we have that cn_1 assigns f to f'. As induction hypotheses, we have first that o is on the notification path of f' by some o'_1 in some n'_1 steps ($n'_1 \leq n'_2$) in cn_1, such that the notification path from o to o'_2 in cn_2 traverses o'_1. Second, we have that o'_2 is on the notification path of f by some o''_1 in some n''_1 steps ($n''_1 \leq n''_2$) in cn_1. Since the path from o to o'_2 in cn_2 traverses o'_1, and because of forward resolution in cn_1, o'_1 is on the notification path of f by o''_1 in at most $n = 2n'_2 + n''_1$ steps, setting $o_1 = o''_1$. □

Proposition 10.9. *If cn_1 and cn_2 are WF2 configurations such that $cn_1 \geqslant cn_2$ and there are cn'_1 and cn'_2 such that $\mathcal{A}_2(cn_1) \rightsquigarrow cn'_1$, and $\mathcal{A}_2(cn_2) \rightsquigarrow cn'_2$, then $cn'_1 \geqslant cn'_2$.*

Proof. Let $obj = \mathsf{o}(o, a'_1, u, \varepsilon, \varepsilon)$ be an object container in cn'_1 and let $obj' = \mathsf{o}(o, a'_2, u, \varepsilon, \varepsilon)$ be the corresponding container in cn'_2. There must then be a container $\mathsf{o}(o, a_1, u', q_{in}, q_{in}) \preceq cn_1$ and a container $\mathsf{o}(o, a_2, u', q_{in}, q_{in}) \preceq cn_2$.

Case 1: We have $a_2(x) = a_1(x)$ for all x, and $a_2(\mathbf{self}) = a_1(\mathbf{self})$. Since normalization does not affect object-environment variable-value mappings, we get that $a'_2(x) = a'_1(x)$, for all x, and $a'_2(\mathbf{self}) = a'_1(\mathbf{self})$.

Case 2: Suppose $\pi_1(a_2'(f)) = v$. Then, cn_2 must assign v to f, and o must be on the notification path of f in cn_2. Consequently, by Proposition 10.8, o is then on the notification path of f in cn_1, while also assigning v to f. Hence, it is the case that $\pi_1(a_1'(f)) = v$.

Case 3: Suppose $o'' \in \pi_2(a_2'(f))$. Then, f is unresolved at o in cn_2', i.e. $\pi_1(a_2'(f)) = \bot$, and thus f is unassigned in cn_2'. Consequently, f is unassigned at o in cn_2, and hence also in cn_1. We now have two cases. Assume first $o'' \in \pi_2(a_2(f))$; then, $o'' \in \pi_2(a_1(f))$ and consequently $o'' \in \pi_2(a_1'(f))$, as needed. Assume next there is some sequence of futures $f_1, f_2, \ldots, f_n$ such that cn_2 assigns f_{i+1} to f_i for $1 \leq i < n$, cn_2 assigns f to f_n, and $o'' \in \pi_2(a_2(f_1))$. By well-formedness, o is on the notification path of $f_1, f_2, \ldots, f_n$ in cn_2, and hence also in cn_1. By assumption, we have that o'' is forwarding resolved for f_1 in cn_1 at o, which means that either $o'' \in \pi_2(a_1(f_1))$ or that o'' is forwarding resolved for f_2 in cn_1 at o, etc. Hence, it will be the case that $o'' \in \pi_2(a_1'(f))$, as needed.

Case 4: Suppose $\pi_1(a_1'(f)) = v$. Then, cn_1' assigns v to f, and consequently, so does cn_2', and hence, since normalization does not affect future assignments, so does cn_1 and cn_2. If $v \in PVal$, we can then conclude. Assume instead $v = f' \in FID$. Assume $\pi_1(a_1(f)) = f'$; this means that o is on the notification path of f' in cn_1 and thus in cn_1'. Otherwise, o must be on the notification path of f in cn_1 (in some number of steps greater than 0), which means that o is also on the notification path of f in cn_1', as needed.

Case 5: Suppose $o'' \in \pi_2(a_1'(f))$. Then, f is unresolved at o in cn_1', i.e. $\pi_1(a_1'(f)) = \bot$, and thus f is unassigned in cn_1'. Consequently, f is unassigned at o in cn_1, and hence also in cn_2. Suppose $o'' \in \pi_2(a_1(f))$. Then either $o'' \in \pi_2(a_2(f))$ or both $o'' \in OID(cn_2)$ and o is on the notification of f in cn_1. Assume the former; we then have $o'' \in \pi_2(a_2'(f))$ as required. Assume the latter; then we have $o'' \in OID(cn_2')$ and o is still on the notification path of f in cn_1', as needed. □

Lemma 10.11. *$\cong_2$ is reduction closed.*

Proof. Suppose that $cn_1 \cong_2 cn_2$ and that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'' \geqslant cn_2'' \reflectbox{$\rightsquigarrow$} \mathcal{A}_2(cn_2)$. Then cn_1 and cn_2 are WF2. We proceed by case analysis on the transition $cn_1 \to cn_1'$, eliding uses of CTXT-1. For the cases T-SEND, T-RCV, MSG-SEND, MSG-RCV, MSG-ROUTE, MSG-DELAY-1, MSG-DELAY-2, MSG-DELAY-3, CALL-RCV-2, FUT-SEND, FUT-RCV, OBJ-REG, OBJ-SEND, and OBJ-RCV, the configuration cn_1' is fundamentally unchanged, so we set $cn_2' = cn_2$ and have $\mathcal{A}_2(cn_1') \rightsquigarrow cn_1'' \geqslant cn_2'' \reflectbox{$\rightsquigarrow$} \mathcal{A}_2(cn_2')$.

The remaining cases include the rules for assignment and sequential control in Figure 8, CALL-SEND-2, RET-2, GET-2, and NEW-2. The rules for sequential control are handled in a structurally similar way; take WFIELD-2 as an example, with a transition from cn_1 to cn_1' of the form

$$cn\ \mathtt{o}(o, a, u, q_{in}, q_{out})\ \mathtt{t}(o, l, x = e; s) \to cn\ \mathtt{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathtt{t}(o, l, s)$$

where $x \in \mathtt{dom}(a)$ and $v = [\![e]\!]_{(a,l)}$. By the definition of $\geqslant$, there is a task container $\mathtt{t}(o, l, x = e; s)$ and an object container $\mathtt{o}(o, a', u', \varepsilon, \varepsilon)$, where a'

has the same variable-value mappings as a, in cn_2''. Hence, it is possible to perform a transition

$$cn'\ \mathsf{o}(o, a', u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e; s) \to cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)$$

and we have that

$$cn\ \mathsf{o}(o, a[v/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s) \cong_2 cn'\ \mathsf{o}(o, a'[v/x], u', \varepsilon, \varepsilon)\ \mathsf{t}(o, l, s)\ ,$$

setting cn_2' to the right-hand side. This is the case since the transition does not change future-value mappings in any object container.

CALL-SEND-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s)) \\ &\quad\to\quad cn\ \mathsf{o}(o, a_1', u_1, q_{in}, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $o' = [\![e_1]\!]_{(a_1,l)}$, $\overline{v} = [\![\overline{e_2}]\!]_{(a_1,l)}$, $f = \mathtt{newf}(u_1)$, and $a_1' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a_1))$. By the definition of extension, we have that $\mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \preceq cn_2''$ and that there is some container $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$, with the same variable-value mappings as o in cn_1. Hence, we can perform a transition

$$\begin{aligned} &cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e_1!m(\overline{e_2}); s) \\ &\quad\to\quad cn'\ \mathsf{o}(o, a_2', u, \varepsilon, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

where $a_2' = \mathtt{fw}(\overline{v}, o', \mathtt{init}(f, a_2))$. It now suffices to show that

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1', u_1, q_{in}, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s) \\ &\quad\cong_2\quad cn'\ \mathsf{o}(o, a_2', u, \varepsilon, \mathsf{enq}(\mathsf{call}(o', o, f, m, \overline{v}), \varepsilon))\ \mathsf{t}(o, l[f/x], s) \end{aligned}$$

referring to the left-hand side as cn_1' and the right-hand side as cn_2'. Compared to the corresponding configurations before the transition, the only changes are that a task has progressed, a call message has been generated, and forwardings added to o' for the futures in $\overline{v}$. Since the configurations are WF2, o is on the notification path of the futures in $\overline{v}$. Hence, the changes effected by the transitions are the same for both configurations. Consequently, the equivalence holds.

RET-2: Consider a transition of the form

$$cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \to cn\ \mathsf{o}(o, a_1[v/f], u_1, q_{in}, q_{out})$$

where $v = [\![e]\!]_{(a_1,l)}$ and $f = l(\mathbf{ret})$. By the definition of extension, we have that $\mathsf{t}(o, l, \mathbf{return}\ e; s) \preceq cn_2''$, and that there is some container $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$ such that $v = [\![e]\!]_{(a_2,l)}$. Hence, we can perform a transition

$$cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \to cn'\ \mathsf{o}(o, a_2[v/f], u, \varepsilon, \varepsilon)$$

and we wish to prove that

$$cn\ \mathsf{o}(o, a_1[v/f], u_1, q_{in}, q_{out}) \cong_2 cn'\ \mathsf{o}(o, a_2[v/f], u, \varepsilon, \varepsilon)$$

referring to the left-hand side as cn'_1 and the right-hand side as cn'_2. Note that f must be unresolved in cn_1 and cn_2, and hence cn''_2, by well-formedness. After running Algorithm 2, v will be resolved at all objects that are on the notification path of f in either configuration. Additionally, if $v \in \mathit{FID}$, the assigned value of that future, if available, will also have been distributed, and so on. The external messages resulting from both cn'_1 and cn'_2 will be the same, since the definition of extension does not permit the addition of forwardings to external OIDs.

GET-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e.\mathbf{get}; s) \\ \to\quad &cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s) \end{aligned}$$

where $[\![e]\!]_{(a_1,l)} = f$ and $\pi_1(a_1(f)) = v$. By the definition of extension and well-formedness, we have that $\mathsf{t}(o, l, x = e.\mathbf{get}; s) \preceq cn''_2$ and that there is some $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn''_2$ such that $[\![e]\!]_{(a_2,l)} = f$ and $\pi(a_2(f)) = v$. Hence, we can perform a transition

$$cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = e.\mathbf{get}; s) \to cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

and we have that

$$cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[v/x], s) \cong_2 cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[v/x], s)$$

setting cn'_2 to the right-hand side. In the same way as for WFIELD-2, this is the case since the transition does not change the future-value mappings in any object container.

NEW-2: Consider a transition of the form

$$\begin{aligned} &cn\ \mathsf{o}(o, a_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ \to\quad &cn\ \mathsf{o}(o, a'_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a', u_1, \varepsilon, \varepsilon) \end{aligned}$$

where we have $o' = \mathtt{newo}(u_1)$, $\overline{v} = [\![\overline{e}]\!]_{(a_1,l)}$, $a'_1 = \mathtt{fw}(\overline{v}, o', a_1)$, and $a' = \mathtt{init}(\overline{v}, \mathtt{init}(C, \overline{v}, o'))$. By the definition of extension, we have that $\mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \preceq cn''_2$ and that there is some $\mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn''_2$ such that $[\![\overline{e}]\!]_{(a_2,l)} = \overline{v}$. We can then perform a transition

$$\begin{aligned} &cn'\ \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\overline{e}); s) \\ \to\quad &cn'\ \mathsf{o}(o, a'_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a', u, \varepsilon, \varepsilon) \end{aligned}$$

where $a'_2 = \mathtt{fw}(\overline{v}, o', a_2)$. We wish to prove that

$$\begin{aligned} &cn\ \mathsf{o}(o, a'_1, u_1, q_{in}, q_{out})\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a', u_1, \varepsilon, \varepsilon) \\ \cong_2\quad &cn'\ \mathsf{o}(o, a'_2, u, \varepsilon, \varepsilon)\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', a', u, \varepsilon, \varepsilon) \end{aligned}$$

referring to the left-hand side as cn'_1 and the right-hand side as cn'_2. Note that the only added forwardings through the transition are from o to o', and that they will be for the same futures in both cn'_1 and cn'_2, by well-formedness. Hence, after running Algorithm 2, there will be an object container $\mathsf{o}(o', a'', u, \varepsilon, \varepsilon)$ in both resulting configurations, where all futures that

are assigned in the original configurations are resolved. In addition both resulting configurations will contain the task $\mathsf{t}(o, l[o'/x], s)$. It remains to argue that the object container with OID o in the configuration resulting from cn_1' extends the corresponding object container resulting from cn_2'. The only difference between the containers compared to their counterparts in cn_1'' and cn_2'' is the possible addition of o' in some forwarding lists; however, these additions will be the same for both containers, so extension still holds.

The same arguments are valid with minor changes for the case when $cn_2'' \geqslant cn_1''$. □

Lemma 10.12. *$\cong_2$ is context closed.*

Proof. Suppose $cn_1 \cong_2 cn_2$, and let cn be a context configuration. We then have that cn_1 and cn_2 are WF2.

Assume $cn_1 \geqslant cn_2$ and that $cn_1\ cn$ is WF2. We first show that $cn_2\ cn$ is WF2. For OID Uniqueness, note that $OID(cn_1) = OID(cn_2)$ by the definition of extension; hence, any OID clash violates the well-formedness of $cn_1\ cn$. For Task-Object Existence, it suffices to consider the case $\mathsf{t}(o, l, s) \preceq cn$ and $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_1$); by the definition of $\cong_2$, there is then also an object container with OID o. Object-Node Existence follows from $\mathtt{graph}(cn_1) = \mathtt{graph}(cn_2)$, which holds by the definition of $\cong_2$. Buffer Cleanliness distributes over composition and thus holds for cn_1, cn and cn_2 separately. Local Routing Consistency only concerns nodes, which are unaffected by adding the context cn. Call Uniqueness holds since messages in $cn_2\ cn$ are either from cn, correspond to some message or task (which is a single writer) in cn_1; Single Writer holds for similar reasons. External OID follows again by noting that nodes are unchanged when applying cn.

For Future Liveness, assume that $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn_2\ cn$, and either f is active for o in $cn_2\ cn$, $a(f) \downarrow$, or $cn_2\ cn$ assigns f to f' and o is on the notification path of f'. Assume without loss of generalization that $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$, since the property already holds in cn_2 separately. We consider the cases in turn. Suppose f is active for o in $cn_2\ cn$; then f is also active for o in $cn_1\ cn$, so o is on the notification path of f in $cn_1\ cn$. If the path in $cn_1\ cn$ from which o can receive f traverses some $o' \in OID(cn_1)$, there must be an equivalent path in $cn_2\ cn$, since extension does not change forwardings to configuration-external OIDs. The case where the path does not go outside of cn is even simples. The other cases hold for similar reasons.

The converse of this well-formedness argument, that $cn_1\ cn$ is WF2 whenever $cn_2\ cn$ is WF2, holds with only minor changes to the above reasoning.

Assume that $\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \geqslant cn_2' \reflectbox{$\rightsquigarrow$} \mathcal{A}_2(cn_2)$. Let $\mathcal{A}_2(cn_1\ cn) \rightsquigarrow cn_1''$ and $\mathcal{A}_2(cn_2\ cn) \rightsquigarrow cn_2''$. Well-formedness holds from the above reasoning. We wish to prove that $cn_1'' \geqslant cn_2''$. We consider the conditions in turn.

$\mathtt{graph}(cn_1'') = \mathtt{graph}(cn_1\ cn) = \mathtt{graph}(cn_1) = \mathtt{graph}(cn_2) = \mathtt{graph}(cn_2\ cn) = \mathtt{graph}(cn_2'')$.

$\mathsf{m}(cn_1'') = \mathsf{m}_2(cn_1\ cn) = \mathsf{m}_2(cn_1) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2) \cup \mathsf{m}_2(cn) = \mathsf{m}_2(cn_2\ cn) = \mathsf{m}(cn_2'')$.

Since $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, we have $\mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn)$. Hence, $\mathsf{t}(cn_1'') = \mathsf{t}_2(cn_1\ cn) = \mathsf{t}_2(cn_2\ cn) = \mathsf{t}(cn_2'')$.

Consider $obj = \mathsf{o}(o, a_1, u, \varepsilon, \varepsilon) \preceq cn_1''$ and $obj' = \mathsf{o}(o, a_2, u, \varepsilon, \varepsilon) \preceq cn_2''$. We want to argue that obj extends obj'. Suppose the containers are derived from an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$.

Case 1: Since Algorithm 2 does not affect **self** or stored values, we have $a_2(\mathbf{self}) = a_1(\mathbf{self})$ and $a_2(x) = a_1(x)$ for all x.

Case 2: Assume $\pi_1(a_2(f)) = v$. Then, if $\pi_1(a(f)) = v$, we also have $\pi_1(a_1(f)) = v$. Suppose $\pi_1(a(f)) = \bot$; then, o is on the notification path in $cn_2\ cn$, and $cn_2\ cn$ assigns v to f. If the notification path traverses $o' \in OID(cn)$, a corresponding notification path will traverse o' in $cn_1\ cn$, since the forwardings and external future messages of cn_1 and cn_2 to external objects coincide after running Algorithm 2. Consequently, o is also on the notification path of f in $cn_1\ cn$, and we have $\pi(a_1(f)) = v$ by well-formedness and the definition of Algorithm 2.

Case 3: Assume $o'' \in \pi_2(a_2(f))$. Then, f is unresolved in $cn_1\ cn$ and $cn_2\ cn$. We need to argue that o'' is forwarding resolved for o in cn_1''. Suppose that $o'' \in \pi_2(a(f))$; then clearly $o'' \in \pi_2(a_1(f))$. Suppose $o'' \notin \pi_2(a(f))$; then there must some sequence of futures $f_1, \ldots, f_n$, such that $cn_1\ cn$ and $cn_2\ cn$ assign f_i to f_{i+1} for $1 \leq i < n$, assign f to f_n, and additionally $o'' \in \pi_2(a(f_1))$ and o is on the notification path of f_1. After running Algorithm 2, we will clearly have $o'' \in \pi_2(a_1(f))$.

Case 4: Assume $\pi_1(a_1(f)) = v$. Then, either cn_1 or cn assigns v to f, which implies that either cn_2 or cn assigns v to f. Suppose $v = f' \in FID$. If $\pi_1(a(f)) = f'$, o is on the notification path of f' in cn_1'' by well-formedness.

Case 5: Assume $o'' \in \pi_2(a_1(f))$. Then f is unassigned and on the notification path of o in $cn_1\ cn$ and consequently also in $cn_2\ cn$. Since external forwardings and messages are the same in cn_1 and cn_2, the forwarding for o'' was added either by some internal message in cn or by a message or forwarding present in both cn_1 and cn_2. Consequently, $o'' \in \pi_2(a_2(f))$. □

Proposition 10.13. $\cong_2$ *is a type 2 witness relation.*

Proof. By Lemma 10.11 and Lemma 10.12, it suffices to prove barb preservation in both directions. Assume without loss of generalization that $\mathcal{A}(cn_1) \rightsquigarrow cn_1' \geqslant cn_2' \leftsquigarrow \mathcal{A}(cn_2)$. Suppose $cn_1 \downarrow obs$, where $obs = ext!m(\overline{v})$; then $msg \preceq cn_1$ with $msg = \mathsf{call}(ext, o, f, m, \overline{v})$, whence $msg \preceq cn_1'$. By the definition of extension, we then have $msg \preceq cn_2'$, such that msg is in some link queue. It is then possible to repeatedly use the rule MSG-DELAY-3 to reach a configuration cn_2'' where msg is at the top of that link queue. We thus have $cn_2 \rightarrow^* cn_2' \rightarrow^* cn_2'' \downarrow obs$, and conclude that $cn_2 \Downarrow obs$. The proof of the converse property is symmetric. □

Lemma 10.14. *Suppose that cn' is WF2, and $cn \geqslant cn'$. Then, cn is WF2 as well, and $cn \simeq_2 cn'$.*

Proof. It suffices to prove that cn is WF2, since we then have that $cn \cong_2 cn'$ by Proposition 10.9, which by Proposition 10.13 implies that $cn \simeq_2 cn'$.

Note that every WF2 clause except Single Writer and Future Liveness holds straightforwardly in cn from the assumption that cn' is WF2, since only future maps in object containers are different. For Single Writer, note that cn does not introduce mappings for unused futures, and does not introduce new messages. For Future Liveness, assume some f is active for o in cn; then f is active for o in cn', and therefore, o is on the notification path of f in cn'. Then, since cn does not remove any OIDs from forwarding lists or any instantiations of futures, o must still be on the notification path of in cn (possibly with fewer steps). □

Lemma 10.15. *Let* $\mathsf{bind}\ \overline{z}.cn$ *be a WF1 configuration in standard form.*

1. *If* $\mathsf{bind}\ \overline{z}.cn \rightarrow \mathsf{bind}\ \overline{z}'.cn'$*, then for some* cn''*,* $\mathtt{net}(cn) \rightarrow^* cn'' \cong_2 \mathtt{net}(cn')$.
2. *If* $\mathtt{net}(cn) \rightarrow cn''$*, then for some* $\overline{z}'$ *and* cn'*,* $\mathsf{bind}\ \overline{z}.cn \rightarrow^* \mathsf{bind}\ \overline{z}'.cn'$ *and* $cn'' \cong_2 \mathtt{net}(cn')$.

Proof. 1. We proceed by cases on the nature of the given type 1 transition. Let

$$\mathsf{bind}\ \overline{z}.cn \rightarrow \mathsf{bind}\ \overline{z}'.cn'\ . \tag{3}$$

Fix cn_{graph} and name representation map $\mathtt{rep}$. As above we elide uses of CTXT-1 in both semantics by applying the rules to arbitrary configuration subsets, and we elide uses of CTXT-2 in the type 1 semantics, by considering transitions in arbitrary binding contexts. Each of the remaining transitions in Figure 4 immediately translates into a corresponding transition at type 2 level, and moreover, the resulting type 2 configuration is in normal form.

Consider for instance rule WFIELD. We obtain a type 1 transition of the form

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s) \rightarrow \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s)$$

where $x \in \mathtt{dom}(a)$ and $v = [\![e]\!]_{(a,l)}$. Let $v' = [\![e]\!]_{(\mathtt{rep}(a),\mathtt{rep}(l))}$; then $\mathtt{rep}(v) = v'$ by (2). We obtain:

$$\begin{aligned}
&\mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s)) \\
&\quad = \quad (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ \mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep}))(cn_{graph}) \\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathtt{net}(\mathsf{t}(o,l,x=e;s),\mathtt{rep})(cn_{graph}))) \\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})(\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph})) \\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x=e;s)\ cn_{graph}) \\
&\quad \rightarrow \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a)[v'/x],u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
&\quad = \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a[v/x]),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph}) \\
&\quad = \quad \mathtt{net}(cn\ \mathsf{o}(o,a[v/x])\ \mathsf{t}(o,l,s))
\end{aligned}$$

using the rule WFIELD-2 to derive the transition.

The remaining rules in Figure 4 are proved in the same manner, so we proceed to the rules in Figure 5.

CALL-SEND: Consider the following type 1 transition:

$$\begin{array}{l}\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e_1!m(\overline{e_2});s)\\ \quad\rightarrow\quad \mathsf{bind}\ f\,\overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[f/x],s)\ \mathsf{f}(f,\bot)\ \mathsf{c}(o',f,m,\overline{v})\end{array}$$

where $o' = [\![e_1]\!]_{(a,l)}$ and $\overline{v} = [\![\overline{e_2}]\!]_{(a,l)}$. We calculate:

$$\begin{array}{rl}
& \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = e_1!m(\overline{e_2});s))\\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ\\
& \mathtt{net}(\mathsf{t}(o,l,x = e_1!m(\overline{e_2});s),\mathtt{rep}))(cn_{graph})\\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})\\
& (\mathtt{net}(\mathsf{t}(o,l,x = e_1!m(\overline{e_2});s),\mathtt{rep})(cn_{graph})))\\
= & \mathtt{net}(cn,\mathtt{rep})(\mathtt{net}(\mathsf{o}(o,a),\mathtt{rep})\\
& (\mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = e_1!m(\overline{e_2});s)\ cn_{graph}))\\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\\
& \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = e_1!m(\overline{e_2});s)\ cn_{graph})\\
\rightarrow \circ \cong_2 & \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{fw}(\mathtt{rep}'(\overline{v}),\mathtt{rep}'(o'),\mathtt{init}(f',\mathtt{rep}'(a))),\\
& u_0,\varepsilon,\mathtt{enq}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o),f',m,\mathtt{rep}'(\overline{v})),\varepsilon))\\
& \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph})\\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep}')(\mathtt{send}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o'),f',m,\mathtt{rep}'(\overline{v})),\\
& \mathsf{o}(\mathtt{rep}'(o),\mathtt{fw}(\mathtt{rep}'(\overline{v}),\mathtt{rep}'(o'),\mathtt{init}(f',\mathtt{rep}'(a))),u_0,\varepsilon,\varepsilon)\\
& \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph}))\\
= & \mathtt{net}(cn,\mathtt{rep}')(\mathtt{send}(\mathsf{call}(\mathtt{rep}'(o'),\mathtt{rep}'(o'),f',m,\mathtt{rep}'(\overline{v})),\\
& \mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[f'/x],s)\ cn_{graph}))\\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[f/x],s)\ \mathsf{f}(f,\bot)\ \mathsf{c}(o',f,m,\overline{v}))
\end{array}$$

using CALL-SEND-2, and where $f' = \mathtt{newf}(u_0)$ and $\mathtt{rep}' = \mathtt{rep}[f'/f]$.

CALL-RCV: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,f,m,\overline{v}) \rightarrow \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s)$$

where $l = \mathtt{locals}(o,f,m,\overline{v})$ and $s = \mathsf{body}(o,m)$. We calculate:

$$\begin{array}{rl}
& \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{c}(o,f,m,\overline{v}))\\
= & (\mathtt{net}(cn,\mathtt{rep}) \circ \mathtt{net}(\mathsf{o}(o,a),\mathtt{rep}) \circ\\
& \mathtt{net}(\mathsf{c}(o,f,m,\overline{v}),\mathtt{rep}))(cn_{graph})\\
= & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\\
& \mathtt{send}(\mathsf{call}(\mathtt{rep}(o),\mathtt{rep}(o),\mathtt{rep}(f),m,\mathtt{rep}(\overline{v})),cn_{graph}))\\
\equiv_2 & \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon)\\
& \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),s)\ cn_{graph})\\
= & \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,s))
\end{array}$$

RET: Consider the following type 1 transition:

$$\mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\bot) \rightarrow \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)$$

where $l(\mathbf{ret}) = f$ and $v = [\![e]\!]_{(a,l)}$. We calculate:

$$\begin{array}{rll}
& \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\bot)) & \\
= & \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) & \\
& \quad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),\mathbf{return}\ e;s)\ cn_{graph}) & \\
\to \circ \cong_2 & \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a)[\mathtt{rep}(v)/\mathtt{rep}(f)],u_0,\varepsilon,\varepsilon)\ cn_{graph}) & \\
= & \quad \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)) &
\end{array}$$

using RET-2.

GET: Consider the following type 1 transition:

$$\begin{array}{rl}
& \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x = e.\mathbf{get};s) \\
\to & \quad \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s)
\end{array}$$

where $f = [\![e]\!]_{(a,l)}$. We calculate:

$$\begin{array}{rll}
& \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x = e.\mathbf{get};s)) & \\
= & \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) & \\
& \quad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = e.\mathbf{get};s)\ cn_{graph}) & \\
\to & \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) & \\
& \quad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l)[\mathtt{rep}(v)/x],s)\ cn_{graph}) & \\
= & \quad \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s)) &
\end{array}$$

using GET-2.

NEW: Consider the following type 1 transition:

$$\begin{array}{rl}
& \mathsf{bind}\ \overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\overline{e});s) \\
\to & \quad \mathsf{bind}\ o'\,\overline{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')
\end{array}$$

where $\overline{v} = [\![\overline{e}]\!]_{(a,l)}$ and $a' = \mathtt{init}(C,\overline{v},o')$. We calculate:

$$\begin{array}{rll}
& \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\overline{e});s)) & \\
= & \quad \mathtt{net}(cn,\mathtt{rep})(\mathsf{o}(\mathtt{rep}(o),\mathtt{rep}(a),u_0,\varepsilon,\varepsilon) & \\
& \quad \mathsf{t}(\mathtt{rep}(o),\mathtt{rep}(l),x = \mathbf{new}\ C(\overline{e});s)\ cn_{graph}) & \\
\to \circ \cong_2 & \quad \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon) & \\
& \quad \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l)[o''/x],s) & \\
& \quad \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph}) & \\
= & \quad \mathtt{net}(cn,\mathtt{rep}')(\mathsf{o}(\mathtt{rep}'(o),\mathtt{rep}'(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(\mathtt{rep}'(o),\mathtt{rep}'(l[o''/x]),s) & \\
& \quad \mathsf{o}(o'',\mathtt{rep}'(a'),u_0,\varepsilon,\varepsilon)\ cn_{graph}) & \\
= & \quad \mathtt{net}(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\ \mathsf{o}(o',a')) &
\end{array}$$

where $\mathtt{rep}' = \mathtt{rep}[o''/o']$ and we use (2) as usual. This completes the proof of property 1.

2. We proceed now by cases on the type 2 transition. Suppose $\mathtt{net}(cn) \to cn''$, and we find $\overline{z}'$, cn' to complete the diagram as stated in the lemma. As

above, we apply the rules to configuration subsets, to elide uses of CTXT-1. Rules in the type 2 instance of Figure 4 are straightforward and omitted. For the rules in Figure 9 excepting CALL-SEND-2, RET-2, GET-2, and NEW-2, we can choose $\overline{z}' = \overline{z}$ and $cn' = cn$, since by Corollary 9.17, $\texttt{net}(cn) \equiv_2 cn''$.

For the four remaining cases, each case is obtained by reversing the arguments, i.e., proving that if the type 2 transition holds, depending on the rule application and shape of configurations, then also the corresponding type 1 transition holds, and the resulting pair of configurations are in $\cong_2$. This completes the argument. □

Theorem 10.16. *For all well-formed type 1 configurations* bind $\overline{z}.cn$ *in standard form,* bind $\overline{z}.cn \simeq \texttt{net}(cn)$.

Proof. We exhibit a bisimulation relation $\mathcal{R}$ defined as

$$\mathcal{R} = \{(\mathsf{bind}\ \overline{z}.cn, cn') \mid \texttt{net}(cn) \cong_2 cn'\}\ ,$$

where bind $\overline{z}.cn$ is a WF1 configuration in standard form, and cn' is a WF2 configuration. Note that $(\mathsf{bind}\ \overline{z}.cn, \texttt{net}(cn)) \in \mathcal{R}$, since the identity relation is included in $\cong_2$. We show that $\mathcal{R}$ is a witness relation. Clearly, $\cong_2$ is reduction closed.

Suppose bind $\overline{z}.cn_1\ \mathcal{R}\ cn_2$ (or the converse for $\mathcal{R}^{-1}$); then bind $\overline{z}.cn_1$ is WF1 and in standard form, cn_2 is WF2, and $\texttt{net}(cn_1) \cong_2 cn_2$.

For reduction closure, assume bind $\overline{z}.cn_1 \to$ bind $\overline{z}'.cn_1'$, where bind $\overline{z}'.cn_1'$ is in standard form. Then, by property 1 of Lemma 10.15, $\texttt{net}(cn_1) \to^* cn_1'' \cong_2 \texttt{net}(cn_1')$. This means that, for some cn_2', $cn_2 \to^* cn_2'$ and $cn_2' \cong_2 cn_1''$. Hence, by transitivity of $\cong_2$, bind $\overline{z}'.cn_1'\ \mathcal{R}\ cn_2'$. For converse reduction closure, assume $cn_2 \to cn_2'$. Then, $\texttt{net}(cn_1) \to^* cn_2''$ and $cn_2' \cong_2 cn_2''$. By property 2 of Lemma 10.15, this means that bind $\overline{z}.cn \to^*$ bind $\overline{z}'.cn_1'$ and $cn_2'' \cong_2 \texttt{net}(cn_1')$. Hence, by the transitivity of $\cong_2$, $cn_2'\ \mathcal{R}^{-1}$ bind $\overline{z}'.cn_1'$.

For context closure, assume bind$\overline{z}.cn_1 cn$ is WF1, and compose the type 2 interpretation of cn with cn_2 to produce $cn_2 cn'$, using the representation map. We need to show that this configuration is WF2; to do this we assume that the interpretation and composition is done so that Object-Node Existence holds, i.e., the type 2 versions of objects in cn get a NID for a node that exists in $\texttt{graph}(cn_2)$. We also assume that Buffer Cleanliness is respected for messages added from cn. For OID Uniqueness, it suffices to consider the case where $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn_2$ and $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \preceq cn'$; if $o_1 = o_2$, then there is a corresponding clash in $cn_1 cn$, violating WF1. For Task-Object Existence, assume $\mathsf{t}(o, l, s) \preceq cn'$; then, if there is no $\mathsf{o}(o, a, u, q_{in}, q_{out}) cn_2 cn'$, there is no corresponding object container in $cn_1 cn$, violating WF1. Local Routing Consistency holds since the composition does not add nodes or links and does not change routing tables. For Call Uniqueness, it suffices to consider the case where $\mathsf{call}(o_1, o_1', f_1, m_1, \overline{v}_1) \preceq cn_2$ and $\mathsf{call}(o_2, o_2', f_2, m_2, \overline{v}_2) \preceq cn'$; if $f_1 = f_2$, then there must be a clash between the corresponding messages in cn_1 and cn, which WF1 rules out. For Single Writer, a clash of identifiers in return futures for tasks and calls can again be traced back to a clash in $cn_1 cn$, contradicting WF1. External OID holds since ext cannot be defined in WF1 configurations, and routing tables are

intact from cn_2. For Future Liveness, if f is active for o in $cn_2 cn'$, a corresponding future is active in $cn_1 cn$, which either has future container with a value, or a task that is producing it; in the former case, we assume an entry is added to the future map of o during composition, and in the latter case, we assume forwarding lists are extended when necessary, meaning that o is on the notification path of f.

It remains to show that $\mathsf{bind}\ \overline{z}.cn_1\ cn\ \mathcal{R}\ cn_2\ cn'$. The network graphs of $\mathtt{net}(cn_1\ cn)$ and $cn_2\ cn'$ coincide since cn' does not introduce any new nodes or links. Clearly, if and only if a task or a message is in cn, a corresponding task or message is in cn'. We already have that the remaining tasks and tasks corresponding to messages in cn_1 and cn_2 coincide. As for external messages, either they are newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration, and thus have a corresponding message. With respect to objects, the future maps of objects in the normal form of $\mathtt{net}(cn_1\ cn)$ are possibly extended when compared to future maps for objects in the normal form of $cn_2\ cn'$, but in all other ways, objects in the normal forms are equal. Hence, $\mathtt{net}(cn_1\ cn) \cong_2 cn_2\ cn'$.

For converse context closure, assume $cn_2\ cn$ is WF2, and compose the type 1 interpretation of cn, cn', with cn_1 to produce $\mathsf{bind}\ \overline{z}.cn_1\ cn'$ in standard form. We need to show that this configuration is WF1. For OID Uniqueness, it suffices to consider the case where $\mathsf{o}(o_1, a_1) \preceq cn_1$ and $\mathsf{o}(o_2, a_2) \preceq cn'$; note that if $o_1 = o_2$, there would have been an OID clash in $cn_2 cn$, which cannot be the case. For Task-Object Existence, assume $\mathsf{t}(o, l, s) \preceq cn'$; then, if there is no $\mathsf{o}(o, a) cn_1 cn'$, there is no corresponding object container in $cn_2 cn$, violating the WF2 assumption. For Call Uniqueness, it suffices to consider the case where $\mathsf{c}(o_1, f_1, m_1, \overline{v}_1) \preceq cn_1$ and $\mathsf{c}(o_2, f_2, m_2, \overline{v}_2) \preceq cn'$; note that if $f_1 = f_2$, there would have been a WF2 Call Uniqueness violation in $cn_2\ cn$. For Single Writer, clashes in future identifiers can be traced back similarly to clashes in $cn_2\ cn$, violating WF2. For Future Existence, if f is active for o in $cn_1\ cn'$, a corresponding future is active in $cn_2\ cn$, meaning that by the WF2 property, there is a notification path to an assignment, task or message, which translates to there being a future container, and a call container or task container when appropriate in $cn_1\ cn'$.

It remains to show that $cn_2\ cn\ \mathcal{R}^{-1}\ \mathsf{bind}\ \overline{z}.cn_1\ cn'$. The network graphs of $\mathtt{net}(cn_1\ cn')$ and $cn_2\ cn$ coincide since cn does not contain any new nodes or links. Clearly, if and only if a task or a task for a message is in cn, a corresponding task or message is in cn'. We already have that the remaining tasks and tasks corresponding to messages in cn_1 and cn_2 coincide. As for external messages, either they are newly introduced via the context, and are thus in both composed configurations, or they come from the original configuration, and thus coincide. With respect to objects, the future maps of objects in the normal form of $\mathtt{net}(cn_1\ cn')$ are possibly extended when compared to future maps for objects in the normal form of $cn_2\ cn$, but in all other ways, objects in the normal forms are equal. Hence, $\mathtt{net}(cn_1\ cn') \cong_2 cn_2\ cn$.

For barb preservation, assume $\mathsf{bind}\ \overline{z}.cn_1 \downarrow obs$. Then $\mathtt{net}(cn_1) \Downarrow obs$,

which by normal form equivalance yields $cn_2 \Downarrow obs$, as needed. For converse barb preservation, assume $cn_2 \downarrow obs$. Then, by normal form equivalence, $\texttt{net}(cn_1) \Downarrow obs$, and consequently $cn_1 \downarrow obs$, whereby $cn_1 \Downarrow obs$. $\square$