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

Abstract

In distributed object systems it is desirable to be able to migrate objects transparently between locations, for instance in order to support load balancing and efficient resource allocation. Existing approaches build complex message routing infrastructures, typically on top of ip, using e.g. message forwarding chains, or centralized object location servers. These solutions are costly in terms of complexity and overhead. In an earlier paper we have shown how location independent routing can be used to implement process overlays in a sound, fully abstract, and efficient manner on top of an abstract network of processing nodes connected by asynchronous point to point channels. The overlays considered in that work allowed only one type of message, with modest requirements on global consistency. In this paper we show how the approach can be generalized to more complex object overlays involving futures, essentially placeholders for method return values that need to be kept consistent and propagated correctly to all objects that eventually need them.

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 in a mobile setting objects location changes dynamically, some form of application level routing is needed for inter-object messages to reach their destination. Various approaches have been considered in the literature (cf. [34] for a survey). One common implementation strategy is to use some form of centralized, replicated, or decentralized object location register, either for forwarding or for address lookup cf. [14, 34, 1, 18]. This type of solution requires some form of synchronization to keep registers consistent with physical location, 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, useful for recovery or for service discovery, but hardly efficient as a general purpose routing device in large systems.

In general one would like a mechanism for object mobility with the following properties:

- 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 size of the network).
- Self-stabilization: Even when started in a transient state, computions should proceed correctly, and converge to a stable state. Observe that this precludes the use of locks.
- Decentralization: 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 (centralizes or decentralized) and pointer forwarding regimes both preclude low stretch, and the use of locks precludes self-stabilization.

In a precursor to this paper [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, OID's), and the information used for routing, namely host identifiers, typically IP numbers. If we were to route messages not according 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 [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 [20, 2, 21] that allows messages (rpc's) to be routed directly to the called object, independently of the physical node on which that object is currently executing. In this way 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 instance for message forwarding.

The language considered in [11] allows to define a dynamically growing or shrinking collection of objects communicating by asynchronous rpc, and thus its functionality is not much different from a core version of Erlang [5], or the nomadic PICT language studied in [34]. The question we raise in [11] is how program behaviour is affected by being run in the networked model, as compared with a more standard (reference) semantics given here using rewrite logic. 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 a location, routing, message passing, and so on. In

[11] we show that, with a maximally nondeterministic networked semantics, and in the sense of barbed equivalence [31] which is a standard equivalence to study in these types of applications [9, 16, 17], programs exhibit the same behaviour in both cases.

Messaging in [11] is very simple. The implicit channel abstraction used in the reference semantics is essentially that of a reliable, unordered communication channel. Messages (calls) are sent according to the program order, but the order in which they are acted upon is arbitrary[1]. Soundness and full abstraction for the networked semantics is therefore an interesting and useful observation, since it allows many conclusions made at the level of abstract program behaviour to transfer to the setting of the networked realization.

The question is how sensitive these results are to the type of communication taking place at the abstract level. The overlays considered in the earlier work allows only one type of message, with modest requirements on global consistency. It is of interest to examine also languages with richer communication structures than asynchronous point to point message passing. To this end we enrich in this paper the language studied earlier, and show that the conclusions of our previous work remain valid, however with more involved constructions. The extension results in much more complex object overlays involving so-called future variables that need to be kept consistent and propagated correctly to all objects that may eventually need them.

Future variables [7, 15, 27, 36, 28, 13] are placeholder variables for values that may be waiting to get instantiated. Futures are used extensively in many concurrent and distributed high-level languages, libraries, and models including Java, .NET, scheme, concurrent LISP, Oz, to name just a few. Many versions of future variables exist in the literature. Our work uses futures as placeholders for return values of remote method calls, as in [6, 13, 8]. Other models exists, such as the transparent first-class futures considered in [3], or the concurrent constraint store model of e.g. Oz [35, 28].

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

1. A remote thread evaluating `obj.m(args)`
2. The assignment of a future to `x`. The future is initially uninstantiated, but is intended to become instantiated after the remote call has returned.

This allows long running tasks to be offloaded to a remote thread with the main thread proceeding to other tasks. When the return value is eventually

[1]This is not strictly speaking true in general, as in the reference semantics, the program order on calls may induce happens-before constraints on external method calls that cannot be realized in the networked semantics because messages are explicitly queued and can always be shuffled. However, barbed equivalence is not sensitive to this type of happens-before constraint in the reference semantics.

needed, the calling thread can request it by executing a **get** on the future. If `x` is uninstantiated, this causes the future to block.

The problem is that futures can be transmitted as parameters between threads as well. If `y` is a future 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 benefit propagation of futures as well, 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 networked 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 [19, 30]. In this work we use eager forward chaining 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 look up futures only as needed. Either approach may benefit from the use of location independent routing.

Our main result is to show that, with a fully nondeterministic semantics, the abstract semantics and the networked semantics with futures implemented by eager forward chaining correspond in the sense of barbed equivalence. This is interesting in itself, as it shows that the networked semantics captures the abstract behaviour 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 uses a normal form construction in two stages. First, it is shown that each well-formed configuration in the networked 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 networked semantics commute with normalization. Normalization brings configurations in the networked 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 semantics of mABS is given in section 4. In section 5 we present type 1 barbed equivalence, the notion of barbed equivalence adapted to the reference semantics. Then, in section 6, we turn to the network semantics and present the runtime syntax and the reduction rules. We proceed by detailing the well-formedness conditions for the network semantics in section 7 and adapt barbed equivalence to the network semantics in section 8. We then present the normal form construction in section 9 and continue by completing the correctness proof in section 10. in section 11 we discuss scheduling, and finally in section 12 we conclude. Longer proofs have been deferred to the appendix.

2 Notation

We use a standard boldface vector notation to abbreviate sequences, for compactness. Thus, $\mathbf{x}$ abbreviates a sequence $x_0, \ldots, x_n$, possibly empty, and $\mathbf{f}(\mathbf{x})$ abbreviates a sequence $f_1\ x_1, \ldots, f_n\ x_n$, etc. Let $\mathbf{x} = x_1, \ldots, x_n$. Then $x_0, \mathbf{x}$ abbreviates $x_0, \ldots, x_n$. Let $g : A \rightarrow B$ 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 $\perp$ for bottom elements, and $A_\perp$ for the lifted set with partial order $\sqsubseteq$ such that $a \sqsubseteq b$ if and only if either $a = b \in A$ or else $a = \perp$. Also, if x is variable ranging over A we often use $x_\perp$ as a variable ranging over $A_\perp$. For g a function $g : A \rightarrow B_\perp$ we write $g(a) \downarrow$ if $g(a) \in B$, and $g(a) \uparrow$ if $g(a) = \perp$. The product of sets (flat cpo's) A and B is $A \times B$ with pairing (a, b) and projections π_1 and π_2.

3 mABS

We define a small concurrent object-based language mABS, short for milli-ABS, with asynchronous calls and futures, as depicted in fig. 1. The mABS language is an extension of the language μABS (micro-ABS) of message-passing processes introduced in [11] with return values as futures, and it corresponds essentially to the asynchronous fragment of the core ABS language extensively studied in the EU FP7 project HATS. The language is

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

Figure 1: mABS abstract syntax

fairly self-explanatory. A program is a sequence of class definitions, along with a set of global variables $\mathbf{x}$, and a "main" statement s. The class hierarchy is flat and fixed. Objects have parameters $\mathbf{x}$, local variable declarations $\mathbf{y}$, and methods $\mathbf{M}$. Methods have parameters $\mathbf{x}$, local variable declarations $\mathbf{y}$ and a statement body. For simplicity we assume that variables have

```
class Server1(){,
  serve(x){,return foo(x)}
}
```

Figure 2: μABS Code Sample 1

```
class Server(){,
  serve(x){s1,s2,f1,f2,r1,r2,
    if small(x){return foo(x)}
    else {
      s1 = new Server() ;
      s2 = new Server() ;
      f1 = s1!serve(upper(x)) ;
      f2 = s2!serve(lower(x)) ;
      r1 = f1.get ;
      r2 = f2.get ;
      return combine(r1,r2)
    }
  }
}
```

Figure 3: μABS Code Sample 2

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 is any significant way and consequently left out.

Statements include standard sequential control structures, and a minimal set of constructs for asynchronous method invocation, object creation, and retrieval of futures (**get** statements).

Example 3.1. A very simple server applying `foo` to its argument is shown in fig. 2.

Example 3.2. Assume that `combine(upper(x),lower(x)) = foo(x)`. The program example in fig. 3 returns immediately with the result, if the argument to `serve` is small. If the argument is not small, two new servers are spawned, and computation of the result on upper and lower tranches is delegated to those servers. The results are then fetched from the two newly spawned servers by evaluating the **get** statements, combined, and returned.

4 Reduction Semantics

We first present an abstract "reference" semantics for mABS using rewriting logic. The presentation follows [11] quite closely. We use the abstract semantics as the point of reference for the concrete network-oriented semantics which we present later. The reduction semantics uses a reduction relation $cn \rightarrow cn'$ where cn, cn' are *configurations*, as determined by the runtime syntax in fig. 4. Later on, we introduce different configurations and transition relations, and so use index 1, or talk of e.g. configurations of "type 1", for this first semantics when we need disambiguate. Terms

$x \in \mathit{Var}$			Variables
$o \in \mathit{OID}$			Object id
$p \in \mathit{PVal}$			Primitive values
$f \in \mathit{Fut}$			Futures
$v \in \mathit{Val}$	$=$	$\mathit{PVal} \cup \mathit{OID} \cup \mathit{Fut}$	Values
$z \in \mathit{Name}$	$=$	$\mathit{Fut} \cup \mathit{OID}$	Names
$l \in \mathit{MEnv}$	$=$	$\mathit{Var} \cup \{\mathbf{ret}\} \rightarrow \mathit{Val}_\perp$	Method environment
$a \in \mathit{OEnv}$	$=$	$\mathit{Var} \rightarrow \mathit{Val}_\perp$	Object environment
$\mathit{tsk} \in \mathit{Tsk}$	$::=$	$\mathsf{t}(o, l, s)$	Task
$\mathit{obj} \in \mathit{Obj}$	$::=$	$\mathsf{o}(o, a)$	Object
$\mathit{fut} \in \mathit{fut}$	$::=$	$\mathsf{f}(f, v_\perp)$	Future
$\mathit{call} \in \mathit{Call}$	$::=$	$\mathsf{c}(o, o', m, \mathbf{v})$	Call
$\mathit{ct} \in \mathit{Ct}$	$::=$	$\mathit{tsk} \mid \mathit{obj} \mid \mathit{call} \mid \mathit{fut}$	Container
$\mathit{cn} \in \mathit{Cn}$	$::=$	$0 \mid \mathit{ct} \mid \mathit{cn}\ \mathit{cn}' \mid \mathsf{bind}\ z.\mathit{cn}$	Configuration

Figure 4: mABS type 1 runtime syntax

of the runtime syntax are ranged over by M, and $\preceq$ is the subterm relation. The runtime syntax uses disjoint, denumerable sets of object identifiers $o \in \mathit{OID}$, futures $f \in \mathit{Fut}$, and primitive values $p \in \mathit{PVal}$. Values are either primitive values, OID's, or futures. Lifted values are ranged over by $v_\perp \in \mathit{Val}_\perp$, and we use $\sqsubseteq$ for the standard, associated partial ordering. OID's and futures are subject to π-calculus like binding. Later, in the type 2 semantics, this type of explicit binding is dropped. Accordingly, names are either OID's or futures, we use z as a generic name variable, and names are bound using the π-like binder bind. We assume throughout that names are uniquely bound. The free names of configuration cn is the set $\mathit{fn}(cn)$, and $\mathit{OID}(cn) = \{o \mid \exists a.\mathsf{o}(o, a) \preceq cn\}$ is the set of OID's of objects occurring in cn. Similarly, $\mathit{Fut}(cn) = \{f \mid \exists v_\perp.\mathsf{f}(f, v_\perp) \preceq cn\}$ is the set of futures 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 $z \notin \mathit{fn}(cn_2)$. We often use a vectorized notation $\mathsf{bind}\ \mathbf{z}.cn$ as abbreviation, letting $\mathsf{bind}\ \varepsilon.cn = cn$ where ε is the empty sequence. The structural identities then allows us to rewrite each configuration into a *standard form* $\mathsf{bind}\ \mathbf{z}.cn$ such that each member of $\mathbf{z}$ 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 environments l and a, respectively, map local variables to values. Task environments are aware of a special variable $\mathbf{ret}$ that the task can use in order to identify its return future. Upon invocation, the task environment is initialized using the operation $locals(o, f, m, \mathbf{v})$ by mapping the formal parameters of m in o to the corresponding actual parameters in $\mathbf{v}$, by initializing the method local variables to suitable null values, by mapping $self$ to o, and by mapping $\mathbf{ret}$ to f, intended as the return future of the task being created. Object environments are initialized using the operation $init(C, \mathbf{v})$, which maps the parameters of C to $\mathbf{v}$, and initializes the object local variables as above.

In addition to $locals$ and $init$, the reduction rules presented below use the following helper functions:

- $body(o, m)$ retrieves the statement of the shape s in the definition body for m in the class of o.
- $\hat{e}(a, l) \in Val$ evaluates e using method environment l and object environment o.

Calls play a special role in defining the external observations of a configuration cn. An observation, or *barb*, is a call expression of the form $o!m(\mathbf{v})$, ranged over by obs. In order to define the observations of a given configuration, we assume a fixed set Ext of external OID's to which outgoing method calls can be directed. Names in Ext are not allowed to be bound. A barb, then, is an external method call, i.e. a method call to an OID in Ext. Calls that are not external are meant to be completed in usual reduction semantics style, by internal reaction with the called object, to spawn a new task. External calls could be represented directly, without introducing a special container type (which is not present in the core ABS semantics of [23]), by saying that a configuration cn has barb $obs = o!m(\mathbf{v})$ if and only if cn has the shape

$$\mathsf{bind}\ \mathbf{o_1}.(cn'\ \mathsf{o}(o_2, a)\ \mathsf{t}(o_2, l, e_1!m(\mathbf{e_2}); s))\ , \qquad (1)$$

where $\hat{e_1}(a, l) = o \in Ext$ and $\hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l}) = \mathbf{v}$. 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 the method bodies $e_1!m_1(\mathbf{e_1'}); e_2!m_2(\mathbf{e_2'})$ and $e_2!m_2(\mathbf{e_2'}); e_1!m_1(\mathbf{e_1'})$. This, however, is difficult to reconcile with the representation (1). To this end call containers are introduced, to allow configurations like (1) to produce a corresponding call, and then proceed to elaborate s.

We next present the reduction rules. For ease of notation the rules assume that sequential composition is associative with unit $\mathbf{skip}$. Figures 5 and 6 present the reduction rules. The rules use the notation $cn \vdash cn' \to cn''$ as shorthand for $cn\ cn' \to cn\ cn''$. We use $\to_1$ when we want to make the reference to the type 1 reduction semantics explicit. Fig. 5 gives the mostly routine rules for assignment, control structures, and contextual reasoning, and fig. 6 gives the more interesting rules that involve inter-object communication, external method invocation, and object creation. A method call causes a new future to be created, along with its future container, initialized to $\perp$. Internal and external calls are treated somewhat asymmetrically, 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 $x \in dom(l)$ then $\mathsf{t}(o, l, x = e; s) \to \mathsf{t}(o, l[\hat{e}(a, l)/x], s)$

wfield: If $x \in dom(a)$ then $\mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[\hat{e}(a, l)/x])\ \mathsf{t}(o, l, s)$

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

if-true: If $\hat{e}(a, l) \neq 0$ 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 $\hat{e}(a, l) = 0$ 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 $\hat{e}(a, l) \neq 0$ 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 $\hat{e}(a, l) = 0$ then $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{while}\ e\{s_1\}; s) \to \mathsf{t}(o, l, s)$

Figure 5: mABS reduction semantics part 1

since external calls are only used to define barbs, and barbs corresponding to the act of receiving a external return value are not very meaningful in a reduction semantics setting[2]. Future instantiation is done when **return** statements are evaluated, and **get** statements cause the evaluating task to hang until the value associated to the future is defined. Wait statements can easily be added; they contribute nothing essential to this presentation. Object creation (**new**) statements cause new objects to be created along with their OID's in the expected manner.

We note some basic properties of the reduction semantics.

Proposition 4.1.

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

[2]They are very meaningful in labelled semantics setting, but that is a different story.

call: Let $o' = \hat{e_1}(a, l)$ in $\mathsf{o}(o, a)\ \mathsf{o}(o', a') \vdash \mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s) \to$
$\mathsf{bind}\ f.\mathsf{t}(o, l[f/x], s)\ \mathsf{t}(o', locals(o', m, f, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l})), body(o', m))\ \mathsf{f}(f, \perp)$

call-ext: If $o' = \hat{e_1}(a, l) \in Ext$ then
$\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s) \to \mathsf{t}(o, l, s)\ \mathsf{c}(o, o', m, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l}))$

ret: Let $l(\mathbf{ret}) = f$ in $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, \mathbf{return}\ e; s)\ \mathsf{f}(f, \perp) \to \mathsf{f}(f, \hat{e}(a, l))$

get: If $v \neq \perp$ then $\mathsf{o}(o, a)\ \mathsf{f}(\hat{e}(a, l), v) \vdash \mathsf{t}(o, l, x = e.\mathbf{get}; s) \to \mathsf{t}(o, l[v/x], s)$

new: $\mathsf{o}(o, a) \vdash \mathsf{t}(o, l, x = \mathbf{new}\ C(\mathbf{e}); s) \to \mathsf{bind}\ o'.\mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', init(C, \hat{\mathbf{e}}(a, l)))$

Figure 6: mABS reduction semantics part 2

Proof. No structural identity nor any reduction rule allows an OID or future to escape its binder. Also no rules allow futures to be re-instantiated. The result follows. □

Consider a program $\boldsymbol{CL}\{\mathbf{x}, s\}$. Assume a reserved OID *main* and a reserved future f_{init}. A *type 1 initial configuration* is any configuration of the shape

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

where $l_{init,1}$ is the initial type 1 method environment assigning suitable default values to the variables in $\mathbf{x}$, and $l(\mathbf{ret}) = f_{init}$. The program can place calls to any of the external objects with OID's in Ext, and in this way produce externally observable output.

We say that a configuration cn_n of type 1 is *reachable* if there is a derivation $cn_{init} = cn_0 \to \cdots \to cn_n$ where cn_{init} is an initial configuration. Reachable configurations satisfy some well-formedness conditions which we make significant use of later in the paper.

Definition 4.2 (Type 1 Well-formedness)**.** A configuration cn is *type 1 well-formed* (WF1) if cn satisfies:

1. *OID Uniqueness*: Suppose $\mathsf{o}(o_1, a_1), \mathsf{o}(o_2, a_2) \preceq cn$ are distinct object occurrences. 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. *Object Existence*: Suppose $o \notin Ext$ occurs in cn. Then $\mathsf{o}(o, a) \preceq cn$ for some object environment a
4. *Object Nonexistence*: Suppose $o \in Ext$. Then $\mathsf{o}(o, a) \not\preceq cn$ for any object environment a
5. *Object Binding*: Suppose $o \notin Ext$. Then $o \notin fn(cn)$
6. *Future Uniqueness*: Suppose $\mathsf{f}(f_1, v_{\bot,1}), \mathsf{f}(f_2, v_{\bot,2}) \preceq cn$ are distinct future occurrences. Then $f_1 \neq f_2$
7. *Single Writer*: If $\mathsf{t}(o, l, s) \preceq cn$ then $\mathsf{f}(l(\mathbf{ret}), \bot) \preceq cn$
8. *External Calls*: If $\mathsf{c}(o, o', m, \mathbf{v}) \preceq cn$ then $o' \in Ext$

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 existence properties are needed to ensure that the partitioning of OID's into external and (by extension) internal is meaningful, in that external references are always routed outside the "current configuration". 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 remain uninstantiated.

Proposition 4.3 (WF1 Preservation)**.** *If cn is WF1 and $cn \to cn'$ then cn' is WF1.*

Proof. By inspection of the rules. $\square$

Theorem 4.4. *If cn is type 1 reachable then cn is WF1.*

Proof. It is sufficient to check that any initial configuration is WF1, and then use proposition 4.3. $\square$

5 Type 1 Barbed Equivalence

Our approach to implementation correctness uses barbed equivalence [31]. The goal is to show that it is possible to remain strongly faithful to the reference semantics, provided all nondeterminism is deferred to be handled by a separate scheduler. This allows to draw strong conclusions also in the case a scheduler is added, as we discuss in section 11. Barbed equivalence requires of a pair of equivalent configurations that the internal transition relation $\to$ is preserved in both directions, while preserving also a set of external observations. Although weaker than corresponding equivalences such as bisimulation equivalence on labelled transition systems, barbed equivalence in nonetheless of interest for the following two reasons:

1. Barbed equivalence offers a reasonable notion of observationally identical behaviour on *closed systems*, i.e. when composition of (in our case) subconfigurations to build larger configurations is not considered because it a) is for some reason not important or relevant, or b) does not offer new observational capabilities.
2. Barbed equivalence can be strengthened in a natural way to *contextual equivalence* [29] by adding to barbed equivalence a natural requirement of closure under context composition. Furthermore, a number of works, cf. [22, 32] 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.

It is, however, far from trivial to devise a natural notion of context that works at the level of the network semantics introduced later, and such that the notions of context correspond at both the abstract, reference semantics level we consider at present, and at the network level. For this reason the account of this paper based on barbed equivalence is also a natural stepping stone towards a deeper study of the notion of context in real-world—or at least not overly artificial—networked software systems.

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

$$\mathsf{bind}\ \mathbf{o}.(cn'\ \mathsf{c}(o, o', m, \mathbf{v}))\ .$$

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

Definition 5.1 (Type 1 Witness Relation, Type 1 Barbed 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. $cn_2 \mathcal{R} cn_1$ (symmetry)
2. If $cn_1 \rightarrow cn_1'$ then $cn_2 \rightarrow^* cn_2'$ for some cn_2' such that $cn_1' \mathcal{R} cn_2'$ (reduction closure)
3. If $cn_1 \downarrow obs$ then $cn_2 \Downarrow obs$ (barb preservation)

The WF1 configurations cn_1 and cn_2 are *type 1 barbed equivalent*, $cn_1 \cong cn_2$, if $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation $\mathcal{R}$.

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

Proposition 5.2. *The identity relation is a type 1 witness relation. Barbed 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. $\mathcal{R}^*$
3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$

Proof. See appendix 1. □

We may conclude that $\cong$ has the expected basic property:

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

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

The following property of barbed equivalence illustrates well its closed system nature:

Proposition 5.4. *Suppose cn is WF1, $o \notin fn(cn)$, and $cn\ \mathsf{o}(o, a)$ is WF1. Then $cn \cong cn\ \mathsf{o}(o, a)$*

Proof. See appendix 1. □

This is reasonable from a closed system perspective, as if $o \notin fn(cn)$ there is no way $\mathsf{o}(o, a)$ can be exercised without outside stimulus, but barbed equivalence lacks a context formation clause that can allow such stimulus to be produced. Thus, from a modeling perspective, all stimulus (= model input) that needs to be considered in a given modeling exercise must be included from the outset.

6 Network 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 the introduction. The approach follows closely the network semantics introduced in [11], with the important difference that return values, as futures, are now included. In addition to the naming, routing, and object migration issues already addressed in [11] the additional challenge is to ensure that futures are correctly assigned and propagated at the network level.

In the network semantics we assume an explicitly given network "underlay": A network of nodes and directional links to 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 messaging becomes possible. Objects may choose to migrate between neighbouring nodes. When this is done is not addressed here; we discuss possible adaptation/scheduling strategies in a separate paper [12]. The propagation of routing information will automatically see to it that routing tables are eventually updated. How and when this is done is again left to a scheduler. Method calls can be issued if the caller task knows the OID of the called object. The call is delivered once a route to the callee is known.

For the language considered in this paper the network semantics must be extended to cover also return values and futures. In this paper we use a form of eager forward chaining [6]. Each object mentioning a future can subscribe to that future at some other object. This may happen in remote method calls where the caller subscribes to the return value later to be provided by the callee. It may also happen when a value containing a future is passed from some sender object to some receiver object. In that case the receiver object becomes subscriber at the sender object for that future. When a future gets instantiated to an actual value at some object, it is the task of that object to forward the instantiation to the subscribing objects. This is the implementation strategy applied in our work as well, and it is the objective of the proof to show that this approach is sound and fully abstract for our network semantics, even when routing is in an unstable state.

Example 6.1. An example illustrating future propagation and the interaction with routing is shown in Figure 7. In configuration 1 an rpc with argument f, a future, is sent from object obj_1 residing on nd_1 is sent to object obj_3 on node nd_3. This causes the following events:

- The forwarding list for f at object obj_1 is augmented to include object obj_3
- A new future g is created at object obj_1, to hold the return value of method $obj_3.m$
- Object obj_3 is augmented with placeholders for f and g, and the forwarding list for g is augmented to point to obj_1

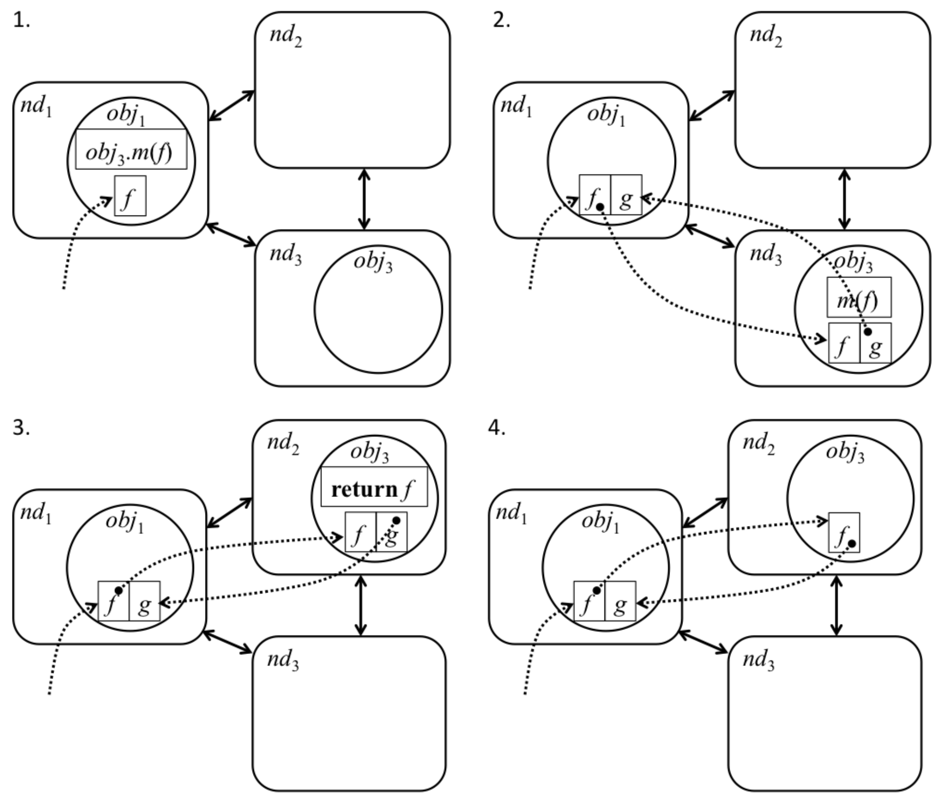

Figure 7: Futures in the mABS network semantics

The scheduler now decides to migrate obj_3 from nd_3 to nd_2. No action is required other than regular routing table updates, as the forward pointers keep pointing to the same objects. Finally, the remote method returns the future f [3]. The forwarding chain for g has now outlived its purpose and can be garbage collected (our semantics does not actually accomplish this quite yet). The forwarding chain for f is augmented to point to the return future g. The resulting loop costs additional messaging in the current rather basic semantics, and could be eliminated. However, it does not cause additional latency as obj_1 is able to discover an assignment to f at obj_1 when it is first made.

6.1 mABS-NET Runtime Syntax

In fig. 8 we present the mABS-NET runtime syntax, i.e. the shape of the runtime state. Recall from section 4 that we reuse symbols as much as possible and use indices to disambiguate. Thus, for instance, Obj_1 is the set Obj of the type 1 semantics in fig. 4, and Obj_2 is the corresponding set in fig. 8. We adopt the same syntactical conventions as in section 4. Tasks are

[3] This might not appear wholly optimal, but for the sake of the example . . .

$u \in \mathit{NID}$			Node identifier
$a \in \mathit{OEnv}_2$	$=$	$(\mathit{Var} \to \mathit{Val}_\perp) \times$ $(\mathit{Fut} \to (\mathit{Val}_\perp \times (\mathit{OID}\ \mathrm{list}))_\perp)$	Object environment
$t \in \mathit{RTable}$	$=$	$\mathit{OID} \to (\mathit{NID} \times \omega)_\perp$	Routing table
$q \in Q$	$=$	Msg^*	Message queue
$\mathit{obj} \in \mathit{Obj}_2$	$::=$	$\mathsf{o}(o, a, u, q_{in}, q_{out})$	Object
$\mathit{nd} \in \mathit{Nd}$	$::=$	$\mathsf{n}(u, t)$	Network node
$\mathit{lnk} \in \mathit{Lnk}$	$::=$	$\mathsf{l}(u, q, u')$	Network link
$\mathit{ct} \in \mathit{Ct}_2$	$::=$	$\mathit{tsk} \mid \mathit{obj} \mid \mathit{nd} \mid \mathit{lnk}$	Runtime container
$\mathit{cn} \in \mathit{Cn}_2$	$::=$	$\mathit{ct}_1 \ldots \mathit{ct}_n$	Configuration
$\mathit{msg} \in \mathit{Msg}$	$::=$	$\mathsf{call}(o, o', f, m, \mathbf{v}) \mid \mathsf{future}(o, f, v) \mid$ $\mathsf{table}(t) \mid \mathsf{object}(\mathit{cn})$	Message

Figure 8: μABS-NET runtime syntax

unchanged from fig. 4. We write $\mathsf{t}(cn)$ for the multiset of tasks in cn, i.e. the multiset $\{tsk \mid \exists cn'.cn = tsk\ cn'\}$, and $\mathsf{o}(cn)$ for the multiset of objects in cn, similarly defined. We also write $\mathsf{m}(cn)$ 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 detailed explanation of other features, in particular routing, we refer to [11].

Network and Routing The network graph contains a vertex u for each node container $\mathsf{n}(u,t)$ and an edge (u,u') for each link $\mathsf{l}(u,q,u')$. The reduction semantics given later does not allow nodes or links to be dynamically 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 impose some constraints on the well-formedness of the network graph, including (i) endpoints of edges exist, (ii) vertices and edges are uniquely determined, (iii) the network graph is reflexive and symmetric, and (iv) the network graph is connected. For routing we adopt a simple Bellman-Ford distance vector (d.v.) discipline. 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 node u with distance n that first visits the node u. We only count hops in this work, for simplicity. A more realistic routing scheme attaches *weights* to the edges, reflecting latency or capacity constraints. Next hop lookup is performed by the operation $nxt(o,t) = \pi_1(t(o))$ where π_1 is the first projection. There is also an operation of updating a routing table t by a routing table t' received from a neighbouring node u, defined by the function

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

Finally, there is an operation $reg(o,u,t)$ that returns the routing table t' obtained by registering o at t's current node u, i.e. such that

$$reg(o,u,t)(o') = \begin{cases} (u,0) & \text{if } o = o' \\ t(o') & \text{otherwise} \end{cases}$$

Message Queues Queue operations are standard: $enq(v,q)$ enqueues v onto the tail of q, $hd(q)$ returns the head of q, and $deq(q)$ returns the tail of the q, i.e. q with $hd(q)$ removed. If q is empty then $hd(q) = deq(q) = \perp$.

Objects and Object Environments Objects $\mathsf{o}(o, a, u, q_{in}, q_{out})$ are now attached to a node u and a pair of 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 fut to pairs $(v_\perp, \mathbf{o})$ where:

- $v_\perp$ is the lifted value currently assigned to fut at the current object, and
- $\mathbf{o}$ is a *forwarding set* of the objects subscribing to updates to fut at the current object.

For instance, if $a(fut) = (\perp, o_1 :: o_2 :: \varepsilon)$ the future fut is as yet uninstantiated (at the object to which a belongs), and, if fut eventually does become instantiated, the instantiation must be forwarded to o_1 and o_2, in random order.

We introduce some syntax to help manipulating object environments:

- $a(x)$ abbreviates $\pi_1(a)(x)$, $a(f)$ 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 $f \notin dom(\pi_2(a))$ then $a[v/f](f) = (v, \varepsilon)$, i.e. the update to value takes effect. Finally we use $a[(v, \mathbf{o})/f]$ for the expected update where both the value and the forwarding list is updated.
- $fw(\mathbf{v}, o, a)$ updates $\pi_2(a)$ by for each future f occurring in $\mathbf{v}$ adding o to the forwarding list of $a(f)$, i.e. by mapping f to the pair either $(\perp, o)$ if $a(f)$ is undefined ($= \perp$), or $(\pi_1(a(f)), o :: \pi_2(a(f)))$ otherwise.

- $init(C, \mathbf{v})$ returns an initial objects environment by mapping the formal parameters of C to $\rightarrow$.
- $init(f, a)$ augments a by mapping f to the pair $(\perp, \varepsilon)$. If $f \notin dom(a)$ then $init(f, a) = a$.
- $init(v, a)$ augments a by mapping each f in v which is uninitialized in a (i.e. such that $f \notin dom(a)$) to $(\perp, \varepsilon)$.

As a consequence of this change, 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 are unchanged, in particular we continue to assume multiset properties of configuration juxtaposition.

Messages The network semantics uses four types of messages. The first is a method call message of the shape $\mathsf{call}(o, o', f, m, \mathbf{v})$, already implicit in the reduction semantics (and explicit for external calls). The remaining three are new: $\mathsf{future}(o, f, v)$ is a future instantiation message, informing object o that f now has been instantiated to value v, $\mathsf{table}(t)$ encodes the routing table t, and $\mathsf{object}(cn)$ is used for object migration. The first two types of messages are said to be *object bound*, and the two latter are *node bound*. We define $dst(msg)$, the *destination* of msg to be o' for msg of the first form above, o for msg of the second form, and $dst(msg) = \perp$ in the remaining two cases. For an object message $\mathsf{object}(cn)$ to be valid, the configuration cn needs to be an *object closure* of the form $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l_1, s_1)\ \ldots\ tsk(o, l_n, s_n)$. Specifically, if cn is any configuration then $clo(cn, o)$, the *closure* of object o with respect to cn, is the multiset of all type 2 containers of the form either $\mathsf{o}(o', a', u', q'_{in}, q'_{out})$ or $\mathsf{t}(o', l', s')$ such that $o' = o$, and $objof(cn)$ is a partial function returning o if all type 2 containers in cn are either objects or tasks, with OID o.

6.2 Reduction Semantics

An important distinction between the reference semantics and the network semantics is the absence of binding. For the standard semantics, name binding plays an important role to avoid clashes between locally generated names. However, in a language with BID's 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 μABS-NET semantics below takes place in the context of a given NID, we can simply assume operations $newf(u)$, resp. $newo(u)$, that return a new future, resp. OID, which is globally fresh for the "current context". We use $new(z)$ for either $newf$ or $newo$ when the nature of z is not known.

We present the mABS-NET reduction rules. First, fig. 5 applies with the following two minor modifications:

- Rule ctxt-2 is dropped as name binding is dropped from the type 2 runtime syntax

- Rule wfield is modified in the obvious way to read: If $x \in dom(a)$ then $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e; s) \to \mathsf{o}(o, a[\hat{e}(a, l)/x], u, q_{in}, q_{out})\ \mathsf{t}(o, l, s)$

The remaining reduction rules are presented in fig. 9.

---

t-send: $\mathsf{n}(u, t) \vdash \mathsf{l}(u, q, u') \to \mathsf{l}(u, enq(\mathsf{table}(t), q), u')$

t-rcv: If $hd(q) = \mathsf{table}(t')$ then $\mathsf{l}(u', q, u)\ \mathsf{n}(u, t) \to \mathsf{l}(u', deq(q), u)\ \mathsf{n}(u, upd(t, u', t'))$

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

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

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

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

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

call-send: Let $o' = \hat{e_1}(a, l)$, $\mathbf{v} = \hat{\mathbf{e_2}}(a, l)$, $f = newf(u)$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s) \to$
$\mathsf{o}(o, fw(\mathbf{v}, o', init(f, a)), u, q_{in}, enq(\mathsf{call}(o, o', f, m, \mathbf{v}), q_{out}))\ \mathsf{t}(o, l[f/x], s)$

call-rcv: If $hd(q_{in}) = \mathsf{call}(o', o, f, m, \mathbf{v})$ then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \to$
$\mathsf{o}(o, fw(f, o', init(\mathbf{v}, init(f, a))), u, deq(q_{in}), q_{out})$
$\mathsf{t}(o, locals(o, m, f, \mathbf{v}), body(o, m))$

fut-send: If $a(f) = (v, o_1 :: \mathbf{o_2})$ then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \to \mathsf{o}(o, fw(v, o_1, a[(v, \mathbf{o_2})/f]), u, q_{in}, enq(\mathsf{future}(o_1, f, v)), q_{out})$

fut-rcv: If $hd(q_{in}) = \mathsf{future}(o, f, v)$ then
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \to \mathsf{o}(o, a[v/f], u, deq(q_{in}), q_{out})$

ret-2: $\mathsf{o}(o, a, u, q_{in}, q_{out})\ \mathsf{t}(o, l, \mathbf{return}\ e; s) \to \mathsf{o}(o, a[\hat{e}(a, l)/l(\mathbf{ret})], u, q_{in}, q_{out})$

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

new-2: Let $o' = newo(u)$ in
$\mathsf{o}(o, a, u, q_{in}, q_{out}) \vdash \mathsf{n}(u, t)\ \mathsf{t}(o, l, x = \mathbf{new}\ C(\vec{e}); s) \to$
$\mathsf{n}(u, reg(o', u, t))\ \mathsf{t}(o, l[o'/x], s)\ \mathsf{o}(o', init(C, \hat{\mathbf{e}}(a, l)), u, \varepsilon, \varepsilon)$

obj-send: Let $cn' = clo(cn, o)$ in
$\mathsf{n}(u, t)\ \mathsf{l}(u, q, u')\ cn \to \mathsf{n}(u, reg(o, u', t))\ \mathsf{l}(u, enq(\mathsf{object}(cn'), q), u')\ (cn - cn')$

obj-rcv: If $hd(q) = \mathsf{object}(cn')$ then
$\mathsf{l}(u', q, u)\ \mathsf{n}(u, t)\ cn\ \to \mathsf{l}(u', deq(q), u)\ \mathsf{n}(u, reg(objof(cn'), u, t))\ cn\ cn'$

Figure 9: mABS-NET reduction rules

Routing The first set of rewrite rules, t-send and t-rcv, are concerned with the exchange of routing tables.

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, will be enforced by the well-formedness condition later, which prohibits output links.

Unstable Routing The two rules msg-delay-1 and msg-delay-2 are used to handle the case where routing tables have not yet stabilized. 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 out-queue without the hosting nodes routing table having information about the message's destination (rule msg-delay-2). Another case is where 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 deadlock. The solution we propose is to use a network self-loop as a buffer for temporarily unroutable messages.

Producing and Consuming Messages The four rules call-send, call-rcv, fut-send, fut-rcv produce and consume messages, method calls and future instantiations. A method call causes a local future to be created and passed with the call message. Upon reception of the call, the callee first initialized those received futures it does not already know about, and then augments the resulting local object environment to forward instantiations of the received future to the caller. Observe that it may be that the callee already knows about the return future of the call. Since message order is not assumed to be preserved a later call referring to the original return future may overtake the earlier call. The eventual return value becomes bound to the return future by the assignment to the constant $\mathbf{ret}$ during initialization of the called methods local environment. The rule fut-send may cause future instantiations to be forwarded to objects in the forwarding list whenever the future is seen to have received a value, and fut-rcv causes the receiving object to update its local environment accordingly. A future may itself be instantiated to a future. The local forwarding table may thus need to be updated.

Language Constructs The three rules ret-2, get-2, new-2 handle the corresponding language constructs. Return statements cause the corresponding future to be instantiated, as explained. 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.

Object Migration The final rules concern object migration. Those rules are *global* in that they are 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. In rule obj-send, $cn - cn'$ is multiset difference.

It is important to notice that all of the above rules are strictly local and appeal only to mechanisms directly implementable at link level: Tests and simple datatype manipulations taking place at a single node, or accesses to a single nodes link layer interface. The “global" property appealed to above for the migration rules 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 nodes self-loop. 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 $cn \to cn'$.*

1. *If $\mathsf{n}(u,t) \preceq cn$ then $\mathsf{n}(u,t') \preceq cn'$ for some t'*
2. *If $lnk = \mathsf{l}(u,q,u') \preceq cn$ then $\mathsf{l}(u,q',u') \preceq cn'$ for some q'*
3. *If $obj = \mathsf{o}(o,a,u,q_{in},q_{out}) \preceq cn$ then there is an object*
$$obj' = \mathsf{o}(o',a',u',q'_{in},q'_{out}) \preceq cn'$$
(the derivative *of obj in cn') such that $o' = o$, $u' = u$, for all x, if $a(x)\downarrow$ then $a'(x)\downarrow$, for all f, if $a'(f)\downarrow$ then $a'(f)\downarrow$, and if $\pi_1(a(f))\downarrow$ then $\pi_1(a'(f))\downarrow$.*
4. *If $tsk = \mathsf{t}(o,l,s) \preceq cn$ then either there is a task $\mathsf{t}(o',l',s') \preceq cn'$ (the* derivative *of tsk in cn) such that $o' = o$, $dom(l) \subseteq dom(l')$, and $l'(\mathbf{ret}) = l(\mathbf{ret})$, or else there is an object $\mathsf{o}(o,a,u,q_{in},q_{out})$ such that $\pi_1(a(l(\mathbf{ret})))\downarrow$*

Proof. By inspecting the rules. □

We then turn to initial configurations. Let a program $\boldsymbol{CL}\{\mathbf{x}, s\}$ be given with reserved OID *main* and a reserved future f_{init}.

Definition 6.3 (Type 2 Initial Configuration)**.** A *type 2 initial configuration* has the shape

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

where:

- $l_{init,2}$ identical to $l_{init,1}$, except that $l_{init,2}(f_{init}) = \perp$,
- $a_{init,2} = \perp[(\perp, \varepsilon)/f_{init}]$,
- cn_{graph} is a configuration consisting only of nodes and links,
- u_{init} names a node $\mathsf{n}(u_{init}, t_{init})$ in cn_{graph},
- $t_{init}(o) = (u_{init}, 0)$, and $t_{init}(o') = \perp$ for $o' \neq o$, and
- $t(o) = \perp$ for all routing tables $t \neq t_{init}$ in cn_{init}.

7 Type 2 Well-formedness

Well-formedness becomes more complex in the case of the network semantics, as account must be taken of e.g. queues, messages in transit, and routing, to ensure that e.g., multiple nodes are never given identical names, 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 some time need the value of a future will also eventually receive it, or else no object does so (in case the method for which the future holds the return value fails to terminate). This is the "future liveness" property in def. 7.4 below. To this end we first define when a future f is active at a given object.

Definition 7.1 (Active Futures)**.** Fix a configuration cn. The future f is *active* in o if one of the following two conditions hold:

1. There is an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ such that $a(f) \downarrow$.
2. A message $\mathsf{call}(o', o, f', m, \mathbf{v})$ is in transit in cn and f occurs in $\mathbf{v}$.

Thus a future f is active in o if either o already has a value for f, or o has received f but not its value, or if o is about to receive a method call which contains f among one of its parameters.

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

Definition 7.2 (Notification Path)**.** Fix a type 2 configuration cn and an object container $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$. Let $n \geq 1$. Inductively, o is on the notification path of f in n steps, if n is minimal such that one of the following conditions hold:

1. $n = 0$, and $\pi_1(a(f)) = v \neq 0$
2. $n = 1$, and there is a task $\mathsf{t}(o, l, s) \preceq cn$ with $l(\mathbf{ret}) = f$
3. $n = 1$, and there is a future message $\mathsf{future}(o, f, v) \preceq cn$
4. $n = n' + 1$ 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 in n' steps
5. $n = 2$, and there is a call message $\mathsf{call}(o, o', f, m, \mathbf{v}) \preceq cn$

Say that o is on the notification list of f if o is on the notification list of f in some number of steps.

Condition 1 is the base case when f has already been instantiated. Condition 2 holds if o is due to receive the return value from one of its pending tasks. Condition 3 holds if a future has been resolved and a future message is in transit to o. Condition 4 holds if o has been inserted into a forwarding list for f at closer distance to the "source", and condition 5 holds if a call has been sent off to o' with return future f. o is then on the notification list of f since the call message is guaranteed to be received at the callee's site and the forwarding list there updated. No clause corresponding to a future being transmitted from o' to o as part of the argument list, as in that case rule call-send sees to it that o':s forwarding list is updated already when the call is made.

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

Proof. See appendix 2. □

We say that cn *assigns v to f* if there is an object container

$$\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$$

such that $a(f) = v \neq \bot$, or else there is a future message $\mathsf{future}(o, f, v) \preceq cn$.

Definition 7.4 (Type 2 Well-formedness)**.** A type 2 configuration cn is *type 2 well-formed* (WF2) if cn satisfies:

1. *OID Uniqueness*: If $\mathsf{o}(o_i, a_i, u_i, q_{in,i}, q_{out,i})$, $i \in \{1, 2\}$, are distinct object occurrences in cn then $o_1 \neq o_2$
2. *Object-Node Existence*: If $\mathsf{o}(o, a, u, q_{in}, q_{out}) \in cn$ then $\mathsf{n}(u, t) \in cn$ for some t
3. *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}, q_{out}
4. *Object Existence*: If $o \notin Ext$ occurs in cn then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \preceq cn$ for some a, u, q_{in}, q_{out}
5. *Object Nonexistence*: Suppose $o \in Ext$. Then $\mathsf{o}(o, a, u, q_{in}, q_{out}) \not\preceq cn$ for any a, u, q_{in}, q_{out}
6. *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. Also, if $msg \preceq q_{in}$ then $dst(msg) = o$
7. *Local Routing Consistency, 1*: If $\mathsf{n}(u, t), \mathsf{o}(o, a, u, q_{in}, q_{out}) \in cn$ then $nxt(o, t) = (u, 0)$
8. *Local Routing Consistency, 2*: If $\mathsf{n}(u, t) \preceq cn$ and $\pi_1(nxt(o, t)) = u'$ then there is a link $\mathsf{l}(u, q, u') \preceq cn$
9. *Future Uniqueness*: If cn assigns v_i to f, $i \in \{1, 2\}$, then $v_1 = v_2$
10. *Single Writer*: If $\mathsf{t}(o, l, s) \preceq cn$ then cn does not assign any v to $l(\mathbf{ret})$ and no message $\mathsf{future}(o', l(\mathbf{ret}), v)$ is in transit for any o', v.

11. *Future Liveness*: If f is pending in o then o is on the notification path from f

Conditions 7.4.1 to 7.4.8 are inherited from [11]. Condition 7.4.11 is the future propagation property discussed above. We use the term "future liveness" not as a guarantee that f will eventually be instantiated, but to indicate that, if eventually f is instantiated somewhere a notification path exists along which the update can be propagated. The rationale behind 7.4.10 is that mABS enforces a single-writer discipline on futures which must be reflected in the well-formedness constraints at the mABS-NET level. Once the future has been assigned through the evaluation of a return statement, the task is "garbage collected". For 7.4.6 observe that only object bound messages (for in-queues, with messages appropriately addressed) enter the object queues. Buffer cleanliness is needed to prevent the formation of contexts that are deadlocked because an in- or out-queue contains messages of the wrong type. For 7.4.7 the requirement should hold only when the object is *not* in transit, as otherwise the object may be on the wire away from node u, and u:s routing table will then have been updated.

Lemma 7.5 (Type 2 Well-formedness Lemma)**.** *If cn is WF2 and $cn \to cn'$ then cn' is WF2.*

Proof. See appendix 2. □

Corollary 7.6. *If cn is type 2 reachable then cn is WF2.*

Proof. First check that initial configurations are WF2 and closed and then use lemma 7.5. □

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

Proposition 7.7. *Suppose that cn is WF2. 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, the result follows. □

8 Type 2 Barbed Equivalence

We adapt the notion of barbed equivalence to the type 2 setting as in [11]. The only difficulty is to define the type 2 correlate of the observation predicate. Say an observation $obs = o!m(\mathbf{v})$ is enabled at a configuration cn if a corresponding call message $\mathsf{call}(o', o, m, \mathbf{v})$ is located at the head of one of the object output queues in cn. More precisely, the type 2 observability predicate is $cn \downarrow obs$, holding if and only if cn has the following shape;

$$cn = cn' \ \mathsf{o}(o', a, u, q_{in}, q_{out}) \tag{2}$$

and $hd(q_{out})$ is defined and equal to $\mathsf{call}(o', o, m, \mathbf{v})$.

It may be thought that the fifo queue discipline goes against the treatment of external calls in the type 1 semantics as there an external call container, once created, will remain as an inert element of all future configurations. Thus, once, say, 5 call containers have been created, all 5 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 the type 2 semantics, external call messages can always be recycled on the reflexive link, allowing available external calls to be shuffled.

There are other ways of defining the observability predicate that may be more natural. For instance one may attach external OID's to specific NID's and restrict observations to those NID's 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, particular in the latter case.

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 symmetry, reduction closure, and barb preservation. Thus:

Definition 8.1. Type 2 Barbed Equivalence Let $cn_1 \cong_2 cn_2$ if and only if $cn_1 \mathcal{R} cn_2$ for some type 2 witness relation $\mathcal{R}$.

In fact, for the purpose of this paper there in no real need to distinguish between the type 1 and type 2 equivalences, and hence we conflate the notions of witness relation and barbed equivalences, by letting the type of the configuration arguments be determined by the context, and use $\cong$ as the generic notion.

9 Normal Forms

An mABS-NET program can be run from an initial state in either the type 1 or the type 2 semantics. We want to show that the behaviour of the programs is preserved, in the sense that the initial states at type 1 and type 2 levels are barbed equivalent.

The key to the proof is a normal form lemma for mABS-NET 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 two steps:

- First we show that routing can be stabilized and inter node links emptied, except for external messages (messages addressed at an external OID). This part if the proof is identical to the corresponding proof in [11]. For this reason we only describe it cursorily here.

Algorithm 1: Stabilize routing and read internal link messages

Input Type 2 well-formed configuration cn on a connected network graph
Output Configuration with stable routing and external link messages only

```
repeat
  Use t-send on each proper link in $cn$ to broadcast routing tables to all
  neighbours ;
  repeat
    Use t-rcv to dequeue one message on a link in $cn$
  until t-rcv no longer enabled ;
  Use msg-rcv, msg-route, msg-delay-1, obj-rcv to dequeue one message
  from each link, if possible
until link queues contain only external messages, and routing is stable
```

Figure 10: Algorithm 1– Stabilize routing and empty link queues of internal messages

- We then complete the construction by emptying object queues, propagating futures, and moving all objects to a single node.

9.1 Stabilization

In the context of a configuration cn call a *proper link* any link $\mathsf{l}(u, q, u')$ for which $u \neq u'$.

Definition 9.1 (Stable Routing, External Queued Messages)**.** Let cn be a type 2 configuration.

1. cn has *stable routing,* if for all $nd(u, t)$, $\mathsf{o}(o, a, u', q_{in}, q_{out}) \preceq cn$, if $nxt(o, t) = u''$ then there is a minimal length path from u to u' which visits u''

2. cn has *external link messages only,* if $\mathsf{l}(u, q, u') \in cn$ and $msg \preceq q$ implies msg is external.

The idea is to first empty link queues as far as possible as we simultaneously exchange routing tables to converge to a configuration with stable routing. This first stage is accomplished using algorithm 1 in fig. 10 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. We then say that cn' is in *stable form*. Stable forms are almost unique, but not quite, since routing may stabilize in different ways.

Proposition 9.2. *Algorithm 1 terminates.*

Proof. See appendix 3. □

Let $\mathsf{t}_1(cn) = \{tsk \mid tsk \preceq cn\}$ and let $\mathsf{o}_1(cn)$ be the multiset of object containers $ct = \mathsf{o}(o, a, u, q_{in}, q_{out})$ in cn such that either $ct \in \mathsf{o}(cn)$, or else $\mathsf{o}(o, a, u', q_{in}, q_{out})$ is in transit in cn from some u' to u (since then, after applying alg. 1, u will host the object). Finally, let $\mathsf{m}_1(cn)$ be the multiset of external messages in transit in cn, or of messages occurring in an object in- or out-queue.

Proposition 9.3. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then*

1. $\mathit{graph}(cn) = \mathit{graph}(cn')$
2. cn' *has stable routing*
3. cn' *has external link messages only*
4. $\mathsf{t}(cn') = \mathsf{t}_1(cn)$
5. $\mathsf{o}(cn') = \mathsf{o}_1(cn)$
6. $\mathsf{m}(cn') = \mathsf{m}_1(cn)$

Proof. Property 1 and 2 are immediate. Property 3 and 4 can be read out of the termination proof. For the remaining three properties observe first that t_1, o_1, and m_1 are all invariant under the transitions used in algorithm 1. The equations follows by noting that only external messages (and so no object closures) are in transit in cn'. □

Prop. 9.3 shows the "almost uniqueness" property alluded to above. The normal form property suggested by prop. 9.3 motivates a notion of equivalence "up to stabilization" defined below.

Definition 9.4 ($\equiv_1$)**.**

1. Let $cn_1 \; \mathcal{R}_1 \; cn_2$ if cn_1 and cn_2 are WF2, $\mathit{graph}(cn_1) = \mathit{graph}(cn_2)$, $\mathsf{t}_1(cn_1) = \mathsf{t}_1(cn_2)$, $\mathsf{o}_1(cn_1) = \mathsf{o}_1(cn_2)$, and $\mathsf{m}_1(cn_1) = \mathsf{m}_1(cn_2)$.
2. Let $cn_1 \equiv_1 cn_2$ if there are cn_1', cn_2' such that

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_1' \; \mathcal{R}_1 \; cn_2' \leftsquigarrow \mathcal{A}_1(\mathbf{z}_2 : cn_2)$$

Prop. 9.3 together with termination of $\mathcal{A}_1$ allows the existential quantifiers in def. 9.4.2 to be exchanged by universal ones.

Corollary 9.5. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then $cn \equiv_1 cn'$*

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

Lemma 9.6. $\equiv_1$ *is reduction closed*

Proof. See appendix 3. □

Proposition 9.7. $\equiv_1$ *is a type 2 witness relation*

Algorithm 2: Normalization

Input Type 2 well-formed configuration $\mathbf{z} : cn$ on a connected network graph **Output** Configuration in type 2 normal form

```
fix a NID $u$ ;
run alg. 1 ;
while some object queue is nonempty {
  use msg-send, msg-delay-2, call-rcv, fut-rcv to dequeue one
    message from each nonempty object queue ;
  while fut-send is enabled { apply fut-send }  } ;
while an object exists not located at $u$ {
  use obj-send to send the object towards $u$ ;
  run alg. 1 }
```

Figure 11: Algorithm 2 – Normalization

Proof. Symmetry is immediate, and reduction closure follows by lemma 9.6. For barb preservation, if

$$cn_1 \quad \equiv_1 \quad cn_2 \tag{3}$$
$$cn_1 \quad \rightarrow^* \quad cn_1' \tag{4}$$

and $cn_1' \downarrow tick(n)$ then by lemma 9.6 we find cn_2' such that

$$cn_1' \equiv_1 cn_2', \tag{5}$$

and by corollary 9.5 and transitivity of $\equiv_1$ we can assume that cn_2' has external link messages only. But then $cn_2' \downarrow tick(n)$ as well. □

Corollary 9.8. *If $\mathcal{A}_1(cn) \rightsquigarrow cn'$ then $cn \cong cn'$*

Proof. By prop. 9.7 and corollary 9.5. □

9.2 Normalization

We then turn to the second normalization step, to empty object queues, propagate futures, and migrate all object closures to a central node. The normalization procedure is algorithm 2 shown in fig. 11. Let $\mathcal{A}_2(cn) \rightsquigarrow cn'$ 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 performed in cycles, each cycle starting and ending in a stable configuration. In each cycle one message is read from the object in- and out-queues. By well-formedness, object queues contain only calls and future messages. Receptions of future messages may cause object environment to instantiate futures. This may cause new future instantiation messages to be enabled. Accordingly, those messages are generated and sent to the objects out-queue. Once this is done, objects not yet at u will be migrated.

Proposition 9.9. *Algorithm 2 terminates*

Proof. See appendix 3. □

We then turn to normal forms and define first a couple of ancillary operations. Let $\mathsf{t}_2(cn)$ be the multiset of method containers $tsk = \mathsf{t}(o, l, s)$ such that one of the following cases apply:

- tsk is a task container in cn.
- There is a message $\mathsf{call}(o', o, f, m, \mathbf{v})$ in transit, o is not external, $l = locals(o, m, f, \mathbf{v})$ and $s = body(o, m)$.

Let $\mathsf{o}_2(cn)$ be the multiset of object containers $\mathsf{o}(o, a, u', \varepsilon, \varepsilon)$ for which the following apply:

- $u' = u$
- There is an object container $obj = \mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$
- $a'(x) = a(x)$ for all variables x
- $a'(f) = (v, \varepsilon)$ with $v \neq \perp$ if and only if for some object container $\mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1}) \preceq cn$, $a_1(f) = (v, \mathbf{o})$, and otherwise $a'(f) = (\perp, \varepsilon)$, if for some such a_1, $a_1(f) = (\perp, \mathbf{o})$.

Also say that cn has *external messages only*, if link queues in cn contain only external messages.

Definition 9.10 (Normal Form)**.** A well-formed configuration cn is in *normal form*, if

1. cn has stable routing
2. cn has external messages only
3. $\mathsf{t}(cn) = \mathsf{t}_2(cn)$
4. $\mathsf{o}(cn) = \mathsf{o}_2(cn)$
5. $\mathsf{m}(cn) = \mathsf{m}_1(cn)$

Proposition 9.11. *Suppose cn is WF2. If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then*

1. *cn' is in normal form*
2. *$graph(cn) = graph(cn')$*
3. *$\mathsf{t}_2(cn) = \mathsf{t}(cn')$*
4. *$\mathsf{o}_2(cn) = \mathsf{o}(cn')$*
5. *$\mathsf{m}_1(cn) = \mathsf{m}(cn')$*

Proof. See appendix 3. □

Proposition 9.11 motivates the following definition of normal form equivalence.

Definition 9.12 ($\equiv_2$). 1. Let $cn_1 \ \mathcal{R}_2 \ cn_2$ if and only if cn_1 and cn_2 are WF2, $graph(cn_1) = graph(cn_2)$, $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$, $\mathsf{o}_2(cn_1) = \mathsf{o}_2(cn_2)$, and $\mathsf{m}_1(cn_1) = \mathsf{m}_1(cn_2)$.

2. Let $cn_1 \equiv_2 cn_2$ if and only if there are cn_1', cn_2' such that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn_1' \ \mathcal{R}_2 \ cn_2' \leftsquigarrow \mathcal{A}_2(cn_2)$$

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

Corollary 9.13. $\equiv_1 \subseteq \equiv_2$

Proof. If $\mathsf{t}_1(cn_1) = \mathsf{t}_1(cn_2)$ then $\mathsf{t}_2(cn_1) = \mathsf{t}_2(cn_2)$ and similar for o_1 and o_2. The result follows. □

We also obtain that normalization respects normal form equivalence.

Corollary 9.14. *If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then $cn \equiv_2 cn'$*

Proof. By prop. 9.11. □

The proof of reduction closure follows that of lemma 9.6 quite closely.

Lemma 9.15. *$\equiv_2$ is reduction closed.*

Proof. See appendix 3. □

Proposition 9.16. *$\equiv_2$ is a type 2 witness relation*

Proof. Similar to the proof of prop. 9.7. □

It follows that if $cn_1 \equiv_2 cn_2$ then $cn_1 \cong cn_2$.

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

Proof. None of the rules used in alg. 2 affects the shape of the normal form. Thus, if $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then $cn \equiv_2 cn'$. But then $cn \cong cn'$, by prop. 9.16. □

10 Correctness

In this section we prove correctness of the network semantics by mapping each well-formed type 1 configuration $\mathsf{bind}\ \mathbf{z}.cn$ in standard form to a well-formed type 2 configuration $down(cn)$ on an arbitrary underlying network graph. We then prove that the two configurations are barbed equivalent, i.e. that $\mathsf{bind}\ \mathbf{z}.cn \cong down(cn)$.

Defining the Underlying Network Graph We first fix a graph represented as a well-formed type 2 configuration cn_{graph} with a distinguished UID u_0, similar to the way initial configurations are defined in section 6. Thus, cn_{graph} consists of nodes and links only, each node u in cn_{graph} has the form (u, t), and each link has the form (u, ε, 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 names in the type 2 semantics, which does not include the binding construct of the type 1 semantics, but on the other hand has different generator functions (the functions $newf$ and $newo$). This prevents the name relation from being modeled using the identity relation. To address this we assume that names and futures in the type 1 semantics are really symbolic, connected to concrete names/futures 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 $rep(f)$, $rep(o)$ in the type 2 semantics. For convenience we extend the name representation map rep to arbitrary values and task environments as follows:

- $rep(o) = o$, if $o \in Ext$,
- $rep(p) = p$, if $p \in PVal$,
- $rep(l)(x) = rep(l(x))$, $rep(l)(\mathbf{ret}) = rep(l(\mathbf{ret}))$

Representing Object Environments To extend rep also to object environments a complication 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 auxillary function $oenvmap(cn, p, rep) \in Fut \to Val_\perp$ on triples of type 1 configurations, a pool of OID/future constants, and a name representation map, as a function which gathers together assignments to futures as determined by the future containers in cn:

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

Fix now a root type 1 configuration cn_0 and a large enough pool p_0 of names (proportional to the size of cn_0, and computed using $newf(u_0)$ and $newo(u_0)$ to conform with our naming policy). Assume that cn_0 is in standard form, i.e. $cn_0 = \mathsf{bind}\ \mathbf{z_0}.cn_0'$ where cn_0' does not have binders. Fix $g = oenvmap(cn_0, p_0, \perp)$ and cn_{graph} as above. We can now extend rep to object environments by:

- $\pi_1(rep(a))(x) = rep(\pi_1(a)(x))$

- $\pi_2(rep(a))(f) = \begin{cases} (g(f), \varepsilon) & \text{if } g(f) \neq \bot \\ (\bot, OID(cn_0)) & \text{otherwise} \end{cases}$

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

$$rep(\hat{e}(a, l)) = \hat{e}(rep(a), rep(l)) \tag{6}$$

Proposition 10.1. *Fix a type 1 well-formed root configuration cn_0 in standard form and pool p_0 as above. Then $rep(a)(f) = (v, \varepsilon)$ if and only if $\mathsf{f}(f, v) \preceq cn$.*

Proof. By well-formedness the future, if it exists, is unique. Pick a name representation map rep. Then $oenvmap(cn_0, p_0, rep)(f)$ 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. This is done in the obvious way by the operation $send$ as follows:

$$\begin{aligned} & send(\mathsf{c}(o, o', f, m, \mathbf{v}), \mathsf{o}(o, a, u, q_{in}, q_{out})\ cn) \\ & \quad = \quad \mathsf{o}(o, a, u, q_{in}, enq(\mathsf{call}(o, o', f, m, \mathbf{v}), q_{out}))\ cn \end{aligned} \tag{7}$$

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, initially the underlying network graph, as the mapping $down$ as follows:

- $down(0, rep)(cn) = cn$
- $down(cn_1\ cn_2, rep) = down(cn_1, rep) \circ down(cn_2, rep)$
- $down(\mathsf{t}(o, l, s), rep)(cn) = \mathsf{t}(rep(o), rep(l), s)\ cn$
- $down(\mathsf{o}(o, a), rep)(cn) = \mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\ cn$
- $down(\mathsf{f}(f, v_\bot), rep)(cn) = cn$
- $down(\mathsf{c}(o, o', f, m, \mathbf{v}), rep)(cn) = send(\mathsf{c}(o, o', f, m, \mathbf{v}), cn, u_0)$

In other words, we represent type 1 configurations by first assuming some underlying network graph, secondly 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 type 2 level.

Defining Routing Tables The only detail remaining to be fixed above is the routing tables. For u_0 the initial routing table, t_0, needs to register all objects in cn_0, i.e.

$$t_0 = reg(g(o_0), u_0, reg(g(o_1), u_0, reg(\cdots, reg(g(o_m), u_0, \bot))\cdots))$$

where $o_0, \ldots, o_m$ are the OID's in cn_0'. 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 alg. 1, and we leave out the details.

Definition 10.2 (Representation Map $down$)**.** Let a network graph cn_{graph} and a name representation map rep be given. For each well-formed type 1 configuration cn_0, the type 2 representation of cn_0 is the configuration $down(cn_0) = down(cn_0, rep)(cn_{graph})$.

In this definition, forwarding lists are overapproximated as compared to the type 2 semantics, which forward futures only to objects that have actually received them. To handle this slight complication we need a little lemma saying that for well-formed type 2 configurations, forwarding lists can be extended without affecting observable behaviour. To make this precise say that $\mathsf{o}(o, a, u, q_{in}, q_{out})$ *extends* $\mathsf{o}(o', a', u', q_{in}', q_{out}')$, if $o = o'$, $u = u'$, $q_{in} = q_{in}'$, $q_{out} = q_{out}'$, $a(x) = a'(x)$ for all x, and $\pi_1(a(f)) = \pi_1(a'(f))$ and $\pi_2(a(f)) \supseteq \pi_2(a'(f))$ for all f.

Lemma 10.3. *Suppose that cn is WF2, and cn' differs from cn only by replacing each object obj by an object obj' such that obj' extends obj. Then cn' is WF2 as well, and $cn \cong cn'$.*

Proof. The check that cn is WF2 only if cn' is, is straightforward. Relate cn and cn' by the relation R such that each object obj in cn is replaced by objects obj' in cn' such that obj' extends obj, or vice versa. We show that R is a type 2 witness relation. Evidently R is symmetric. For the remaining properties (reduction closure and barb preservation) it suffices to note that $\{cn_1 \mid \mathcal{A}_2(cn) \rightsquigarrow cn_1\} = \{cn_2 \mid \mathcal{A}_2(cn') \rightsquigarrow cn_2\}$. This property relies heavily on Future Liveness, def. 7.4. This establishes the result. □

Following [11] we can now show a key lemma allowing us to relate transitions in the two semantics under barbed equivalence.

Lemma 10.4. *Let $\mathsf{bind}\ \mathbf{z}.cn$ be WF1 in standard form.*

1. *If $\mathsf{bind}\ \mathbf{z}.cn \to \mathsf{bind}\ \mathbf{z'}.cn'$ then $down(cn) \to^* \circ \cong down(cn')$*
2. *If $down(cn) \to cn''$ then for some $\mathbf{z'}$, cn', $\mathsf{bind}\ \mathbf{z}.cn \to^* \mathsf{bind}\ \mathbf{z'}.cn'$ and $cn'' \cong down(cn')$*

Proof. See appendix 4. □

We can now state the main result.

Theorem 10.5 (Correctness of the Type 2 Semantics)**.** *For all well-formed type 1 configurations cn on a connected network graph,*

$$cn \cong down(cn)$$

Proof. We exhibit a barbed bisimulation relation $\mathcal{R}$ as follows:

$$\mathcal{R} = \{(cn, cn') \mid down(cn) \cong cn'\} \tag{8}$$

We show that $\mathcal{R}$ is a witness relation.

First for reduction-closure: If $cn_1 \mathcal{R} cn_2$ then $down(cn_1) \cong cn_2$. If $cn_1 \rightarrow cn_1'$ then by lemma 10.4.1, $down(cn_1) \rightarrow^* cn' \cong down(cn_1')$. It follows that $cn_2 \rightarrow^* cn_2'$ such that $cn' \cong cn_2'$. But then $cn_1' \mathcal{R} cn_2'$. Conversely, if $cn_2 \rightarrow cn_2'$ then $down(cn_1) \rightarrow^* cn'$ and $cn' \cong cn_2'$. By lemmma 10.4.2, $cn_1 \rightarrow^* cn_1'$ and $cn' \cong down(cn_1')$. But then $cn_1' \mathcal{R} cn_2'$ as desired.

For Barb Preservation assume $cn_1 \mathcal{R} cn_2$. Then $cn_1 \downarrow o!m(\mathbf{v})$ if and only if $down(cn_1) \Downarrow o!m(\mathbf{v})$ if and only if $cn_2 \Downarrow o!m(\mathbf{v})$. □

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, and 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 overhead. This raises the question of how to determine these parameters, something which we address in more detail in [12].

Regardless how this is done, 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 10.5 achieves more than is called for, as soundness and full abstraction a priori applies only to the type 2 semantics with all transitions included.

A scheduler can be viewed abstractly as a predicate on histories in the following way. Let a *scheduled execution* be any sequence $\rho = cn_0 cn_1 \cdots cn_n$ such that $cn_i \rightarrow cn_{i+1}$ for all $i : 0 \leq i < n$ where the 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 any cn where $\langle cn \rangle$ is the one element execution consisting of cn (a scheduler kicks in only when execution is started), and
2. if $\mathcal{S}(cn_0 \cdots cn_n)$ and $cn_n \rightarrow cn_{n+1}$ for some cn_{n+1} then $\mathcal{S}(cn_0 \cdots cn_n cn_{n+1})$ for exactly one cn_{n+1}.

Then a *scheduled type 2 semantics* is a transition system on executions $\rho = cn_0 \cdots cn_n$ such that $\rho \rightarrow \rho'$ if and only if $\rho' = cn_0 \cdots cn_n cn_{n+1}$ 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 and barb preservation (when $cn_0 \cdots cn_n \downarrow obs$ if and only if $cn_n \downarrow obs$), but not necessarily symmetry. We immediately obtain from theorem 10.5 the following corollary:

Corollary 11.1. *For all well-formed type 1 configurations cn on a connected network graph,*

$$\langle down(cn)\rangle \ \sqsubseteq \ cn \ .$$

Proof. It suffices to note that if $\rho = cn_0 \cdots cn_n \to \rho' = cn_0 \cdots cn_{n+1}$ then $cn_n \to cn_{n+1}$, and if $\rho \downarrow obs$ then $cn_n \downarrow obs$ as well. $\square$

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. The present paper focuses on correctness. In other ongoing work [12] we study the use of the model presented here adapted to the full asynchronous fragment of the core ABS language [23] studied in the EU FP7 HATS project. In that paper we report on experimental results on autonomous performance adaptation for load balancing and latency management, with very promising results.

The correctness analysis is based on barbed equivalence, similar to other past work mostly belonging to the π-calculus school of process algebra [16, 9, 17]. A closely related precursor is Nomadic Pict [34]. 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 were 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 language [35] 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 Ambient calculus [4]). Past correctness analyses for languages with futures include [8] which 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 [33]. In a sequence of papers, Johnsen and coworkers, cf. [24], have studied an extension of ABS with deployment components that can be used as a reflection mechanism for explicit performance management. We are mainly interested in automating the performance adaptation process, and so reflection mechanisms are not top priority.

Scalability is not fully resolved in the present work. We use a rather naive 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 network node. 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 wellknown drawbacks that arise in the case of intermittent network partitioning.

Besides routing we see two main directions for future work. The first is to examine richer language semantics, specifically towards more dynamicity. In ongoing work we are studying power control: Adding an explicit knob to the network 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 [12] 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 sizes, 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. D. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing: programming Environments, Languages, Security and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer-Verlag, 2003.

[2] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. Rofl: routing on flat labels. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 363–374, New York, NY, USA, 2006. ACM.

[3] A. Cansado, L. Henrio, and E. Madelaine. Transparent first-class futures and distributed components. *Electr. Notes Theor. Comput. Sci.*, 260:155–171, 2010.

[4] L. Cardelli and A. D. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.

[5] R. Carlsson. An introduction to core erlang. In *In Proceedings of the PLI 2001 Erlang Workshop*, 2001.

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

[7] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Principles of Programming Languages (POPL'04)*, pages 123–134. Association of Computing Machinery, 2004.

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

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

[10] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. In *Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, ASAMA '99, pages 22–, Washington, DC, USA, 1999. IEEE Computer Society.

[11] M. Dam. Location independent routing in process network overlays. Manuscript, to be submitted.

[12] M. Dam, A. Jafari, A. Lundblad, and K. Palmskog. Abs-net: Fully decentralized runtime adaptation for distributed objects. Manuscript, to be submitted.

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

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

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

[16] 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.

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

[18] D. Havelka, C. Schulte, P. Brand, and S. Haridi. Thread-based mobility in oz. In *Proceedings of the Second international conference on Multiparadigm Programming in Mozart/Oz*, MOZ'04, pages 137–148, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] L. Henrio, M. U. Khan, N. Ranaldo, and E. Zimeo. First class futures: Specification and implementation of update strategies. In *Euro-Par Workshops*, pages 295–303, 2010.

[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.

[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. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[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. Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–. Springer-Verlag, 2012.

[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.

[26] M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/K: A kernel language for component-based open programming. In *GPCE’07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

[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.

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

[29] 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.

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

[31] D. Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[32] 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.

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

[34] P. Sewell, P. T. Wojciechowski, and A. Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Transactions on Programming Languages and Systems*, 34, April 2010.

[35] G. Smolka. The definition of kernel oz. In Andreas Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, 1995.

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

Appendix 1: Proofs for Section 5

Proposition 5.2 *The identity relation is a type 1 witness relation. Barbed 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. $\mathcal{R}^*$
3. $\mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1$

Proof. The identity relation is trivially symmetric, reduction closed and barb preserving. Suppose $cn_1 \cong cn_2$. Then $cn_1 \mathcal{R} cn_2$ for some type 1 witness relation. Then $cn_2 \mathcal{R} cn_1$ by symmetry, such that $cn_2 \cong cn_1$, so symmetry holds. If $cn_1 \to cn_1'$ then we find cn_2' such that $cn_2 \to^* cn_2'$ and $cn_1' \mathcal{R} cn_2'$. But then $cn_1' \cong cn_2'$, and we have shown reduction closure. For barb preservation, if $cn_1 \downarrow obs$ then $cn_2 \Downarrow obs$, as $\mathcal{R}$ is barb preserving. Inverse closure follows from symmetry, and reflexive, transitive closure 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 $cn_1 \mathcal{R}_1 cn_{1,1} \mathcal{R}_2 cn_{1,2} \mathcal{R}_1 cn_2$ for some $cn_{1,1}$, $cn_{1,2}$ so $cn_2 \mathcal{R}_1 cn_{1,2} \mathcal{R}_2 cn_{1,1} \mathcal{R}_1 cn_1$ by symmetry of $\mathcal{R}_1$ and $\mathcal{R}_2$. Second, if $cn_1 \to cn_1'$ we find $cn_{1,1}'$, $cn_{1,2}'$, cn_2' such that $cn_{1,1} \to^* cn_{1,1}'$ etc. such that, by stepwise iteration, $cn_1' \mathcal{R}_1 \circ \mathcal{R}_2 \circ \mathcal{R}_1 cn_2'$, showing reduction closure. Barb preservation is similar, but simpler. $\square$

Proposition 5.4 *Suppose cn is WF1, $o \notin fn(cn)$, and $cn\ \mathsf{o}(o,a)$ is WF1. Then $cn \cong cn\ \mathsf{o}(o,a)$*

Proof. Let $cn \mathcal{R} cn'$ if and only if $cn' = cn\ \mathsf{o}(o,a)$ as described in the the statement of the proposition. We show that $\mathcal{R} \cup \mathcal{R}^{-1}$ is a type 1 witness relation:

Symmetry: Trivial

Reduction closure: First if $cn \to cn_1$ then $cn\ \mathsf{o}(o,a) \to cn_1\ \mathsf{o}(o,a)$, and $cn_1 \mathcal{R} cn_1\ \mathsf{o}(o,a)$. If $cn\ \mathsf{o}(o,a) \to cn_1$ and cn_1 does not have the required shape $cn_1'\ \mathsf{o}(o,a)$ such that $cn \to cn_1'$ then the transition must be an instance of call, and o must be the callee. This is so, as in any other case, a transition involving o requires a task at o to be present in cn. But then o is free in cn as well, contrary to assumptions.

Barb preservation: If $cn \downarrow obs$ then $cn\ \mathsf{o}(o,a) \downarrow obs$, and it $cn\ \mathsf{o}(o,a) \downarrow obs$ then $cn \downarrow obs$, as $o \notin fn(cn)$. $\square$

Appendix 2: Proofs for Section 7

Lemma 7.3 *Fix cn and an object $obj = \mathsf{o}(o,a,u,q_{in},q_{out}) \preceq cn$. If o is on the notification path of f in n steps in the configuration cn, $cn \to cn'$, then o is on the notification path of f in at most n steps in cn'.*

Proof. The proof is by induction on n. We follow the case analysis in def. 7.2. In case 1 we obtain that $\pi_1(a(f)) \neq \bot$ and then by prop. 6.2.3 we find obj' such that $\pi_1(a'(f)) \downarrow$. In case 2, by prop. 6.2.4 there are two options: Either we find a task $\mathsf{t}(o, l', s') \preceq cn'$ with $l'(\mathbf{ret}) = f$ or else we find obj' such that $\pi_1(a'(l(\mathbf{ret}))) \downarrow$. In case 3 we obtain that either $\mathsf{future}(o, f, v) \preceq cn'$, or else $\pi_1(a'(f)) = v$. In case 4 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 in $n{-}1$ steps in cn. By prop. 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 in cn', now in some $n'' \leq n{-}1$ 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. Finally, in case 5, by inspection of the rules we either find that the call message $\mathsf{call}(o, o'', f, m, \mathbf{v}) \prec cn'$, or else there is a task $\mathsf{t}(o'', l'', s'') \preceq cn'$ such that $l''(\mathbf{ret}) = f$, and we are done. $\square$

Lemma 7.5 *If cn is WF2 and $cn \to cn'$ then cn' is WF2.*

Proof. We consider each transition rule in turn. For well-formedness, NID and link uniqueness is trivial, as $graph(cn) = graph(cn')$.

OID uniqueness: In all rules except new-2 there is a 1-1 correspondence between object occurrences in cn and object occurrences in cn'. This is sufficient to conclude. For new-2 it is sufficient to note that o' is a freshly generated OID.

Object-Node Existence, Task-Object Existence: The properties follow since nodes and objects are never 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.

Object Existence, Object Nonexistence: SImple invariant checks.

Buffer cleanliness: We check that only object bound messages enter in- and out-queues. This concerns the rules msg-rcv, call-send, fut-send, msg-delay-1 and msg-delay-2 only. The check is routine.

Local Routing Consistency, 1, 2: Easily proved by case analysis.

Future uniqueness: We only need to consider rules which assign a non-$\bot$ 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 7.4.5.

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.

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 pending in o for the configuration cn'. If o is a newly created OID, no future is pending in o. Either $a'(f) \downarrow$, or else a call message $\mathsf{call} o', o, f', m, \mathbf{v}$ is in transit in cn' and f occurs in $\mathbf{v}$. We proceed by cases on the transition rule leading to cn'. Any rule that does not directly affect any of the conditions in def. 7.1 or def. 7.2 immediately allows to conclude that f is pending in o also in cn. By the induction hypothesis we can conclude that o is on the notification path from f in cn and then o is on the notification path from f also in cn', as the only exception in lemma 7.3 is when o:s environment is updated. For the remaining rules there are the following cases to consider:

- call-send: 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 7.2.5, since a call message is in transit from o to o' with return future f. If f is another future which is pending in o in cn' then f is also pending in 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 7.3. 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 7.3 to conclude that this also applies to cn'.

- call-rcv: 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 def. 7.2.2. Another option is that f is a future in $\mathbf{v}$ (referring to the transition rule in fig. 9). Then f is pending in o in cn as well, by def. 7.1. If f is some other future the case is completed by the IH as above.

- fut-send: Follows from the induction hypothesis and lemma 7.3, as in the case for call-send.

- fut-rcv: If f is pending in o for the configuration cn' then f is pending in o also for cn, and f is not the received future. But then the result follows by the IH and lemma 7.3.

- ret-2: If f is pending in o for cn' then f is not the return future and we again complete by the induction hypothesis.

□

Appendix 3: Proofs for Section 9

Proposition 9.2 *Algorithm 1 terminates.*

Proof. In each iteration of the outermost loop of alg. 1, exactly one message is enqueued on each proper link, and at least one message is dequeued (from all link queues), so the sum of messages in transit in link queues does not exceed its initial value. The rules msg-rcv, msg-delay-1, obj-rcv cause messages to leave the link queues, except for external messages. Moreover, if the link queues have only routing table messages the algorithm terminates in that iteration. So if the algorithm fails to terminate it must be because msg-route is from some point n_0 onwards applied in each iteration of the outermost loop. From n_0 onwards, no messages other than table updates are delivered (to the receiving node, or to the receiving object). In particular, no object messages can be in transit on a link from that point onwards.

We show that then routing tables must at some point stabilize. At point n_0 (as all other points) each node u has $t(o) = (u, 0)$ whenever o:s host is u, by def. 7.4.9. Let m_0 be the length of the largest link queue at the point from which no messages are delivered. After $n_0 + m_0 + 1$ iterations, each node u has received at least one table update from each of its neighbours u', and the last table update applied to u has $t(o) = 0$. As result, at point $n_0 + m_0 + 1$ each node u has $t(o) = (u', 1)$ whenever the host of o is u' and the minimal length path from u to u' has length 1. The entry of u:s routing table for o will not change from that point onwards. We say that those entries are *stable*. Proceeding, let m_1 be the length of the largest link queue at 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 the message will be delivered.

The only detail remaining to be checked is that a message can always be read from a link, but table and object messages can always be delivered, and call and future messages can also always be delivered, if nothing else to the self loop, in case the routing table has not yet been updated, or if the message is external and the destination object is not known to the routing table. This is the only case where msg-delay-1 is used, in fact. □

Lemma 9.6 $\equiv_1$ *is reduction closed.*

Proof. Assume that

$$cn_1 \to cn_1' \tag{9}$$

and

$$cn_1 \equiv_1 cn_2 \ , \tag{10}$$

and show that

$$cn_2 \to^* cn_2' \tag{11}$$

such that

$$cn_1' \equiv_1 cn_2' \ . \tag{12}$$

The proof is by cases on the rewriting rule applied. Most cases are straightforward. For rules not among call-send-in, call-send-out, call-rcv, fut-send, fut-rcv, if

$$\mathcal{A}_1(cn_1) \rightsquigarrow cn_{1,1} \ \mathcal{R}_1 \ cn_{2,1} \leftsquigarrow \mathcal{A}_1(cn_2) \tag{13}$$

and $\mathcal{A}_1(cn_1') \rightsquigarrow cn_{1,1}'$ then we obtain

$$cn_{1,1}' \ \mathcal{R}_1 \ cn_{1,1} \ , \tag{14}$$

by prop. 9.3 and since the correspondences between tasks, objects, and call and future messages are maintained in pre- and poststates. For the remaining rules:

call-send: From (9) we may assume that

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) &(15)\\ cn_1' &= cn_{1,1}\ \mathsf{o}(o,fw(\mathbf{v},o',init(f,a)),u,q_{in},enq(msg,q_{out}))\\ &\quad \mathsf{t}(o,l[f(x)],s) &(16)\end{aligned}$$

where $\mathbf{v}=\hat{\mathbf{e_2}}(a,l)$, $f=newf(u)$, $msg=\mathsf{call}(o,o',f,m,\mathbf{v})$, $o'=\hat{e_1}(a,l)$. From (10) we get

$$\mathcal{A}_1(cn_1)\rightsquigarrow cn_1''\ \mathcal{R}_1\ cn_2''\leftsquigarrow \mathcal{A}_1(cn_2)\ . \tag{17}$$

By prop. 9.3, (15) and (17),

$$\mathsf{o}(o,a,u,q_{in},q_{in},q_{out}),\mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s)\in cn_1'' \tag{18}$$

and hence, by the definition of $\mathcal{R}_1$,

$$\mathsf{o}(o,a,u,q_{in},q_{in},q_{out}),\mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s)\in cn_2'' \tag{19}$$

as well. But then it follows that the configuration cn_2 can mimick the call-send step by cn_1 by first stabilizing to cn_2'' and then performing the call-send step, obtaining cn_2'. It is also clear by prop. 9.3 that (12) holds, completing the case.

The remaining cases are involve a similar amount of detail but are, as this, essentially straightforward. □

Proposition 9.9 *Algorithm 2 terminates.*

Proof. Routing is stable after each run of alg. 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 out-queue is nonempty, causing output queue size to decrease by one. By Buffer Cleanliness, one of call-rcv 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 outqueue 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 and call-rcv. Of these, call-send is never used in either alg. 1 or alg. 2. Each application of call-rcv consumes one call message, and none of the rules cause new call messages to be created. Thus, eventually, call-rcv 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, 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 alg. 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. $\square$

Proposition 9.11 *Suppose cn is WF2. If $\mathcal{A}_2(cn) \rightsquigarrow cn'$ then*

1. *cn' is in normal form*
2. *$graph(cn) = graph(cn')$*
3. *$\mathsf{t}_2(cn) = \mathsf{t}(cn')$*
4. *$\mathsf{o}_2(cn) = \mathsf{o}(cn')$*
5. *$\mathsf{m}_1(cn) = \mathsf{m}(cn')$*

Proof. Property 9.11.2 follows from prop. 9.3.1.

For property 9.11.3 observe first that the function t_2 is invariant under transitions used in alg. 2. On termination of alg. 2 only external messages are in transit, and since no rule causes a task to be modified, 9.11.3 follows.

For 9.11.4 let $\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \in \mathsf{o}(cn')$. We show

$$\mathsf{o}(o_2, a_2, u_2, q_{in,2}, q_{out,2}) \in \mathsf{o}_2(cn) .$$

By definition, $q_{in,2} = q_{out,2} = \varepsilon$. Also, $u_2 = u$. We know that there is an object container $\mathsf{o}(o, a', u'', q_{in}, q_{out}) \preceq cn$, as there is a 1-1 correspondence between object containers in pre- and poststate for each transition used in alg. 2. We also know that $a'(x) = a_2(x)$ for all x. Suppose finally that an object container $ct = \mathsf{o}(o_1, a_1, u_1, q_{in,1}, q_{out,1})$ exists in cn with $a_1(f) = (v, \mathbf{o})$. Let $\mathsf{o}(o_1, a_1', u_1', q_{in,1}', q_{out,1}')$ be the derivative of ct in cn'. Then $\pi_1(a_1'(f)) = v$ as well, by prop. 7.7. We know by Future Uniqueness that $a_2(f) = (v', \mathbf{o'})$ implies $v' = v$. It remains to show that $\pi_1(a_2(f)) \neq \bot$. Assume not. The future f is then pending in o_2. By the Type 2 Subject Reduction Lemma, 7.5, cn' is WF2, so by Future Liveness, o_2 is on the notification path from f in cn' in some n steps. We proceed by induction on n:

- $n = 0$ and $\pi_1(a_2(f)) = v \neq \bot$. This is a contradiction.
- $n = 1$ and there is a task $\mathsf{t}(o_2, l_2, s_2) \preceq cn'$ with $l_2(\mathbf{ret}) = f$. By Future Binding, $\pi_1(a_1'(f)) = \bot$, a contradiction

- $n = 1$ and there is a future message $\mathsf{future}(o_2, f, v') \preceq cn'$, which is a contradiction, since the only queued messages in cn' are external
- $n = n' + 1$ and there is an object $\mathsf{o}(o, a, u, qin, q_{out}) \preceq cn'$ such that $o_2 \in \pi_2(a(f))$, and o is on the notification path from f in n' steps. Either $\pi_1(a(f)) = v \neq \bot$, but then cn' is not in normal form, contradiction. Alternatively, we conclude by the IH.
- There is a call message $\mathsf{call}(o_2, o, f, m, \mathbf{v}) \preceq cn'$. This again contradicts the external messages only property.

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

For 9.11.5 the property holds as it does so already for alg. 1.

We finally need to prove 9.11.1. Property 9.10.1 is trivial, as each run of alg. 2 ends with a run of alg. 1, and alg. 1 ensures that cn' has stable routing. Property 9.10.2 holds since alg. 1 ensures almost empty link queues, and since on termination, alg. 2 ensures empty object queues. For 9.10.3 the result follows since only external messages are in transit in cn'. For 9.10.4, 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 $obj \in cn'$ as needed to be shown. Finally, 9.10.5 holds since it does so already for alg. 1. □

Lemma 9.15 *$\equiv_2$ is reduction closed.*

Proof. Assume that

$$cn_1 \to cn_1' \tag{20}$$

and

$$cn_1 \equiv_2 cn_2\ , \tag{21}$$

and we show that

$$cn_2 \to^* cn_2' \tag{22}$$

for some cn_2' such that

$$cn_1' \equiv_2 cn_2' \tag{23}$$

As above the proof is by cases on the rewrite rule justifying 20. We can assume that cn_2 is in normal form, by 9.14 and transitivity of $\equiv_2$. For rules that do not affect $\mathsf{t}_2(cn_1)$, $\mathsf{o}_2(cn_1)$, or $\mathsf{m}_1(cn_1)$ the result is trivial. Rules in fig. 5 commute directly, i.e. the same rule applied to cn_1 can be applied to cn_2, in the same way. This follows since cn_2 is in normal form. Rules such as msg-send, msg-rcv that ship around messages between object and link queues are also very easy to prove, by reference to prop. 21.

call-send: From (20) we may assume that

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out})\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) && (24)\\ cn_1' &= cn_{1,1}\ \mathsf{o}(o,\mathit{fw}(\mathbf{v},o',\mathit{init}(f,a)),u,q_{in},\mathit{enq}(msg,q_{out}))\\ &\quad \mathsf{t}(o,l[f(x),s) && (25)\end{aligned}$$

where $\mathbf{v}=\hat{\mathbf{e_2}}(a,l)$, $f=\mathit{newf}(u)$, $msg=\mathsf{call}(o,o',f,m,\mathbf{v})$, $o'=\hat{e_1}(a,l)$. By 21, since cn_2 is in normal form, we obtain

$$\mathcal{A}_2(cn_1)\rightsquigarrow cn_1''\ \mathcal{R}_2\ cn_2 \qquad (26)$$

for some choice of cn_1''. By prop. 9.11 we obtain that

$$cn_2=cn_{2,1}\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \qquad (27)$$

where $a'(x)=a(x)$ for variables x, and $a'(f)=(v',\varepsilon)$ if and only if $a_1(f)=(v',\mathbf{o})$ for some $\mathbf{o}$, where a_1 is an object environment in cn_1. It follows that cn_2' can be chosen so that

$$cn_2'=cn_{2,1}\ \mathsf{o}(o,\mathit{fw}(\mathbf{v},o',\mathit{init}(f,a')),u,\varepsilon,\mathit{enq}(msg,\varepsilon))\ \mathsf{t}(o,l[f(x),s), \qquad (28)$$

and

$$\mathcal{A}_2(cn_1')\rightsquigarrow cn_{1,3} \qquad (29)$$

with $cn_{1,3}=cn_{1,4}\ \mathsf{o}(o,\mathit{fw}(\mathbf{v},o',\mathit{init}(f,a')),u,\varepsilon,\mathit{enq}(msg,\varepsilon))\ \mathsf{t}(o,l[f(x),s)$ such that $cn_{1,4}\ \mathcal{R}_2\ cn_{2,1}$. It follows that $cn_2\rightarrow cn_2'$ and $cn_1'\equiv_2 cn_2'$, as desired.

fut-send: From 20 we can assume

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out}) && (30)\\ cn_1' &= cn_{1,1}\ \mathsf{o}(o,a[(v,\mathbf{o_2})/f],u,q_{in},\mathit{enq}(\mathsf{future}(o_1,f,v),q_{out})) && (31)\end{aligned}$$

where $a(f)=(v,o_1::\mathbf{o_2})$. Using 21, since cn_2 is in normal form, we obtain

$$\mathcal{A}_2(cn_1)\rightsquigarrow cn_1''\ \mathcal{R}_2\ cn_2 \qquad (32)$$

for some choice of cn_1'', and we can write cn_1'' in the form

$$cn_1''=cn_1'''\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon) \qquad (33)$$

where $a'(x)=a(x)$ and $a'(f')=(v',\varepsilon)$ if and only if some object environment in cn_1 assigns v' to f'. By prop. 9.11 we get

$$cn_2=cn_{2,1}\ \mathsf{o}(o,a',u,\varepsilon,\varepsilon) \qquad (34)$$

and then we can choose cn_2' as

$$cn_2'=cn_{2,1}\ \mathsf{o}(o,a'[(v,\varepsilon)/f],u,\varepsilon,\mathit{enq}(\mathsf{future}(o_1,f,v),\varepsilon)) \qquad (35)$$

noting that then (23) holds.

call-rcv: In this case we get

$$\begin{aligned} cn_1 &= cn_{1,1}\ \mathsf{o}(o,a,u,q_{in},q_{out}) && (36)\\ cn_1' &= cn_{1,1}\ \mathsf{o}(o,\mathit{fw}(f,o',\mathit{init}(\mathbf{v},\mathit{init}(f,a))),u,\mathit{deq}(q_{in}),q_{out})\\ &\quad \mathsf{t}(o,\mathit{locals}(o,m,f,\mathbf{v}),\mathit{body}(o,m)) && (37)\end{aligned}$$

where $hd(q_{in}) = \mathsf{call}(o', o, f, m, \mathbf{v})$. Again using prop. 21 with cn_2 in normal form we get that

$$\mathcal{A}_2(cn_1) \rightsquigarrow cn_1'' \; \mathcal{R}_2 \; cn_2 \tag{38}$$

where cn_1'' can be written as

$$cn_1'' = cn_1''' \; \mathsf{o}(o, a', u, \varepsilon, \varepsilon) \; \mathsf{t}(o, \mathit{locals}(o, m, f, \mathbf{v}), \mathit{body}(o, m)) \tag{39}$$

where a' is as in the previous case. Now using prop. 21, since $hd(q_{in}) = \mathsf{call}(o', o, f, m, \mathbf{v})$, we obtain

$$cn_2 = cn_{2,1} \; \mathsf{o}(o, a', u, \varepsilon, \varepsilon) \; \mathsf{t}(o, \mathit{locals}(o, m, f, \mathbf{v}), \mathit{body}(o, m)) \; , \tag{40}$$

and $cn_1'' \; \mathcal{R}_2 \; cn_2$, completing the case.

The remaining cases fut-rcv, ret-2, get-2, new-2, and the object migration rules are proved in a similar fashion as the above.

□

Appendix 4: Proofs for Section 10

Lemma 10.4 *Let* $\mathsf{bind}\ \mathbf{z}.cn$ *be WF1 in standard form.*

1. *If* $\mathsf{bind}\ \mathbf{z}.cn \to \mathsf{bind}\ \mathbf{z'}.cn'$ *then* $down(cn) \to^* \circ \cong down(cn')$
2. *If* $down(cn) \to cn''$ *then for some* $\mathbf{z'}$, cn', $\mathsf{bind}\ \mathbf{z}.cn \to^* \mathsf{bind}\ \mathbf{z'}.cn'$ *and* $cn'' \cong down(cn')$

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

$$\mathsf{bind}\ \mathbf{z}.cn \to \mathsf{bind}\ \mathbf{z'}.cn' \; . \tag{41}$$

Fix cn_{graph} and name representation map 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 fig. 5 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

$$\mathsf{bind}\ \mathbf{z}.cn \; \mathsf{o}(o, a) \; \mathsf{t}(o, l, x = e; s) \to \mathsf{bind}\ \mathbf{z}.cn \; \mathsf{o}(o, a[\hat{e}(a, l)/x]) \; \mathsf{t}(o, l, s) \tag{42}$$

where $x \in dom(a)$. We obtain:

$$
\begin{aligned}
& down(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x=e;s)) \\
&\quad = \quad (down(cn,rep) \circ down(\mathsf{o}(o,a),rep) \circ down(\mathsf{t}(o,l,x=e;s),rep))(cn_{graph}) \\
&\quad = \quad down(cn,rep)(down(\mathsf{o}(o,a),rep)(down(\mathsf{t}(o,l,x=e;s),rep)(cn_{graph}))) \\
&\quad = \quad down(cn,rep)(down(\mathsf{o}(o,a),rep)(\mathsf{t}(rep(o),rep(l),x=e;s)\ cn_{graph})) \\
&\quad = \quad down(cn,rep)(\mathsf{o}(rep(o),rep(a),u_0,\varepsilon,\varepsilon)\ \mathsf{t}(rep(o),rep(l),x=e;s)\ cn_{graph}) \\
&\quad \to \quad down(cn,rep)(\mathsf{o}(rep(o),rep(a)[\hat{e}(rep(a),rep(l))/x],u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep(o),rep(l),s)\ cn_{graph}) \\
&\quad = \quad down(cn,rep)(\mathsf{o}(rep(o),rep(a[\hat{e}(a,l)/x]),u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep(o),rep(l),s)\ cn_{graph}) \\
&\quad = \quad down(cn\ \mathsf{o}(o,a[\hat{e}(a,l)/x])\ \mathsf{t}(o,l,s))
\end{aligned}
$$

using (6) to justify the second but last step and the type 2 wfield rule to derive the transition.

The remaining rules in fig. 5 are proved in the same manner, so we proceed to the rules in fig. 6.

call: Consider the following type 1 transition:

$$
\begin{aligned}
& \mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s) \\
&\quad \to \quad \mathsf{bind}\ \mathbf{z},f.cn\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l[f/x],s) \\
&\qquad\qquad \mathsf{t}(o',locals(o',m,f,\hat{\mathbf{e_2}}(\mathbf{a},\mathbf{l})),body(o',m))\ \mathsf{f}(f,\bot)
\end{aligned} \qquad (43)
$$

where $o' = \hat{e_1}(a,l)$. We calculate:

$$
\begin{aligned}
& down(cn\ \mathsf{o}(o,a)\ \mathsf{o}(o',a')\ \mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s)) \\
&\quad = \quad down(cn,rep)(down(\mathsf{o}(o,a),rep)(down(\mathsf{o}(o',a'),rep) \\
&\qquad\qquad (down(\mathsf{t}(o,l,x=e_1!m(\mathbf{e_2});s),rep)(cn_{graph})))) \\
&\quad = \quad down(cn,rep)(\mathsf{o}(rep(o),rep(a),u_0,\varepsilon,\varepsilon)\ \mathsf{o}(rep(o'),rep(a'),u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep(o),rep(l),x=e_1!m(\mathbf{e_2});s)\ cn_{graph}) \\
&\quad \to \quad down(cn,rep)(\mathsf{o}(rep(o),fw(rep(\mathbf{v}),rep(o'),init(f',rep(a))),u_0,\varepsilon, \\
&\qquad\qquad enq(\mathsf{call}(rep(o),rep(o'),f',m,rep(\mathbf{v})),\varepsilon))\ \mathsf{o}(rep(o'),rep(a'),u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep(o),rep(l)[f'/x],s)\ cn_{graph}) \\
&\quad = \quad down(cn,rep')(\mathsf{o}(rep'(o),fw(rep'(\mathbf{v}),rep'(o'),init(rep'(f),rep'(a))),u_0,\varepsilon, \\
&\qquad\qquad enq(\mathsf{call}(rep'(o),rep'(o'),rep'(f),m,rep'(\mathbf{v})),\varepsilon))\ \mathsf{o}(rep'(o'),rep'(a'),u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep'(o),rep'(l)[rep'(f)/x],s)\ cn_{graph}) \\
&\quad = \quad down(cn,rep')(\mathsf{o}(rep'(o),fw(rep'(\mathbf{v}),rep'(o'),init(rep'(f),rep'(a))),u_0,\varepsilon, \\
&\qquad\qquad enq(\mathsf{call}(rep'(o),rep'(o'),rep'(f),m,rep'(\mathbf{v})),\varepsilon))\ \mathsf{o}(rep'(o'),rep'(a'),u_0,\varepsilon,\varepsilon) \\
&\qquad\qquad \mathsf{t}(rep'(o),rep'(l[f/x]),s)\ cn_{graph}) \\
&\quad = \quad cn'
\end{aligned}
$$

where by (6), $\mathbf{v} = \hat{\mathbf{e_2}}(\mathbf{a},\mathbf{l})$, $o' = \hat{e_1}(a,l)$, $f' = newf(u_0)$, and where $rep' =$

$\mathit{rep}[f'/f]$. Using corollary 9.17 and the definition of $\equiv_2$ we obtain

$$\begin{aligned} cn' \;&\cong\; \mathit{down}(cn, \mathit{rep}')(\mathsf{o}(\mathit{rep}'(o), a_1, u_0, \varepsilon, \varepsilon)\ \mathsf{o}(\mathit{rep}'(o'), a_2, u_0, \varepsilon, \varepsilon) \\ &\quad \mathsf{t}(\mathit{rep}'(o), \mathit{rep}'(l)[f'/x], s) \\ &\quad \mathsf{t}(\mathit{rep}'(o'), \mathit{locals}(\mathit{rep}'(o'), m, f', \mathbf{v}), \mathit{body}(\mathit{rep}(o'), m))\ cn_{\mathit{graph}}) \\ &= cn'' \end{aligned}$$

where $a_1(x) = \mathit{rep}(a)(x)$ and $a_2(x) = \mathit{rep}(a')(x)$ for all x in $\mathit{dom}(a)$, $\mathit{dom}(a')$, respectively, and where $a_1(f) = a_2(f) = (v, \varepsilon)$ if $a'''(f) = (v, \mathbf{o})$ for some a''' in cn' and $\mathbf{o}$, and $a_1(f) = a_2(f) = (\bot, \varepsilon)$ $a'''(f)$ has the shape $(\bot, \mathbf{o})$ for all a''' in cn' and moreover some such a''' exists. In this case $a_1 = \mathit{rep}'(a)$ and $a_2 = \mathit{rep}'(a')$ by prop. 10.1. Then

$$\begin{aligned} cn'' \;&=\; \mathit{down}(cn, \mathit{rep}')(\mathsf{o}(\mathit{rep}'(o), \mathit{rep}'(a), u_0, \varepsilon, \varepsilon)\ \mathsf{o}(\mathit{rep}'(o'), \mathit{rep}'(a'), u_0, \varepsilon, \varepsilon) \\ &\quad \mathsf{t}(\mathit{rep}'(o), \mathit{rep}'(l)[\mathit{rep}'(f)/x], s) \\ &\quad \mathsf{t}(\mathit{rep}'(o'), \mathit{locals}(\mathit{rep}'(o'), m, \mathit{rep}'(f), \mathbf{v}), \mathit{body}(\mathit{rep}'(o'), m))\ cn_{\mathit{graph}}) \\ &= \mathit{down}(cn\ \mathsf{o}(o, a)\ \mathsf{o}(o', a')\ \mathsf{t}(o, l[f/x], s) \\ &\quad \mathsf{t}(o', \mathit{locals}(o', m, f, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l})), \mathit{body}(o', m))\ \mathsf{f}(f, \bot)) \\ &= \mathit{down}(cn\ \mathsf{o}(o, a)\ \mathsf{o}(o', a')\ \mathsf{t}(o, l[f/x], s) \\ &\quad \mathsf{t}(o', \mathit{locals}(o', m, f, \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l})), \mathit{body}(o', m))\ \mathsf{f}(f, \bot)) \end{aligned}$$

as desired.
call-ext: Consider the following type 1 transition:

$$\begin{aligned} &\mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s) \\ &\quad \to\ \mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, s)\ \mathsf{c}(o, o', m, \mathbf{v}) \end{aligned} \tag{44}$$

where $o' = \hat{e_1}(a, l) \in \mathit{Ext}$ and $\mathbf{v} = \hat{\mathbf{e_2}}(\mathbf{a}, \mathbf{l})$. We calculate:

$$\begin{aligned} &\mathit{down}(cn\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s)) \\ &\quad= \mathit{down}(cn, \mathit{rep})(\mathit{down}(\mathsf{o}(o, a), \mathit{rep})(\mathit{down}(\mathsf{t}(o, l, x = e_1!m(\mathbf{e_2}); s), \mathit{rep}) \\ &\qquad (cn_{\mathit{graph}}))) \\ &\quad= \mathit{down}(cn, \mathit{rep})(\mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon) \\ &\qquad \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l), x = e_1!m(\mathbf{e_2}); s)\ cn_{\mathit{graph}}) \\ &\quad\to \mathit{down}(cn, \mathit{rep})(\mathsf{o}(\mathit{rep}(o), \mathit{fw}(\mathbf{v}, \mathit{rep}(o'), \mathit{init}(f', \mathit{rep}(a))), u_0, \varepsilon, \\ &\qquad \mathit{enq}(\mathsf{call}(\mathit{rep}(o), \mathit{rep}(o'), f', m, \mathit{rep}(\mathbf{v})), \varepsilon))\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l)[f'/x], s) \\ &\qquad cn_{\mathit{graph}}) \\ &\quad= \mathit{down}(cn, \mathit{rep})(\mathit{send}(\mathsf{call}(\mathit{rep}(o), \mathit{rep}(o'), f', m, \mathit{rep}(\mathbf{v})), \varepsilon), \\ &\qquad \mathsf{o}(\mathit{rep}(o), \mathit{fw}(\mathbf{v}, \mathit{rep}(o'), \mathit{init}(f', \mathit{rep}(a))), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l)[f'/x], s) \\ &\qquad cn_{\mathit{graph}}) \\ &\quad= \mathit{down}(cn, \mathit{rep})(\mathit{send}(\mathsf{call}(\mathit{rep}(o), \mathit{rep}(o'), f', m, \mathit{rep}(\mathbf{v})), \varepsilon), \\ &\qquad \mathsf{o}(\mathit{rep}(o), \mathit{rep}(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(\mathit{rep}(o), \mathit{rep}(l)[f'/x], s) \\ &\qquad cn_{\mathit{graph}}) \\ &\quad= \mathit{down}(cn\ \mathsf{o}(o, a)\ \mathsf{t}(o, l, s)\ \mathsf{c}(o, o', m, \mathbf{v}))\ . \end{aligned}$$

ret: Consider next the type 1 transition below:

$$\mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\perp) \rightarrow \mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v) \qquad (45)$$

where $l(\mathbf{ret}) = f$ and $v = \hat{e}(a,l)$. Again we calculate:

$$\begin{aligned}
&down(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,\mathbf{return}\ e;s)\ \mathsf{f}(f,\perp))\\
&\quad = \quad down(cn, rep)(\mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l), \mathbf{return}\ e;s)\\
&\qquad\quad cn_{graph})\\
&\quad \rightarrow \quad down(cn, rep')(\mathsf{o}(rep(o), rep(a)[\hat{e}(rep(a), rep(l))/l(\mathbf{ret})], u_0, \varepsilon, \varepsilon)\\
&\qquad\quad cn_{graph})\\
&\quad = \quad down(cn, rep')(\mathsf{o}(rep'(o), rep'(a), u_0, \varepsilon, \varepsilon)\ cn_{graph})\\
&\quad = \quad down(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v))
\end{aligned}$$

where in the second but last step we use the fact that

$$rep' = rep[\hat{e}(rep(a), rep(l))/l(\mathbf{ret})]\ .$$

get: Now consider the following type 1 transition:

$$\begin{aligned}
&\mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x = e.\mathbf{get};s)\\
&\quad \rightarrow \quad \mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s) \qquad (46)
\end{aligned}$$

where $f = \hat{e}(a,l)$. Again we calculate:

$$\begin{aligned}
&cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l,x = e.\mathbf{get};s)\\
&\quad = \quad down(cn, rep)(\mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l), x = e.\mathbf{get};s)\\
&\qquad\quad cn_{graph})\\
&\quad \rightarrow \quad down(cn, rep)(\mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep(o), rep(l)[v/x]s)\ cn_{graph})\\
&\quad = \quad down(cn\ \mathsf{o}(o,a)\ \mathsf{f}(f,v)\ \mathsf{t}(o,l[v/x],s))
\end{aligned}$$

new: The final case is for object creation:

$$\begin{aligned}
&\mathsf{bind}\ \mathbf{z}.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\mathbf{e});s)\\
&\quad \rightarrow \quad \mathsf{bind}\ \mathbf{z}, o'.cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\mathsf{o}(o', init(C, \hat{\mathbf{e}}(\mathbf{a},\mathbf{l}))) \qquad (47)
\end{aligned}$$

We obtain:

$$\begin{aligned}
&down(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l,x = \mathbf{new}\ C(\mathbf{e});s))\\
&\quad = \quad down(cn, rep)(\mathsf{o}(rep(o), rep(a), u_0, \varepsilon, \varepsilon)\\
&\qquad\quad \mathsf{t}(rep(o), rep(l), x = \mathbf{new}\ C(\mathbf{e});s)\ cn_{graph})\\
&\quad \rightarrow \quad down(cn, rep')(\mathsf{o}(rep'(o), rep'(a), u_0, \varepsilon, \varepsilon)\\
&\qquad\quad \mathsf{t}(rep'(o), rep'(l)[rep'(o')/x], s)\\
&\qquad\quad \mathsf{o}(rep'(o'), init(C, \hat{\mathbf{e}}(\boldsymbol{rep'}(\mathbf{a}), \boldsymbol{rep'}(\mathbf{l}))), u_0, \varepsilon, \varepsilon)\ cn_{graph})\\
&\quad = \quad down(cn, rep')(\mathsf{o}(rep'(o), rep'(a), u_0, \varepsilon, \varepsilon)\ \mathsf{t}(rep'(o), rep'(l[o'/x]), s)\\
&\qquad\quad \mathsf{o}(rep'(o'), rep'(init(C, \hat{\mathbf{e}}(\mathbf{a},\mathbf{l}))), u_0, \varepsilon, \varepsilon)\ cn_{graph})\\
&\quad = \quad down(cn\ \mathsf{o}(o,a)\ \mathsf{t}(o,l[o'/x],s)\mathsf{o}(o', init(C, \hat{\mathbf{e}}(\mathbf{a},\mathbf{l}))))
\end{aligned}$$

where we switch from rep to rep' and use (6) as usual. This completes the proof of statement 1.

2. We proceed now by cases on the type 2 transition. Suppose $down(cn) \to cn''$, and we find $\mathbf{z'}$, 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 fig 5 are straightforward and left to the reader. For the rules in fig. 9 excepting call-send, ret-2, get-2, and new-2 we can choose $\mathbf{z'} = \mathbf{z}$ and $cn' = cn$, since by def. 9.12 and corollary 9.17, $down(cn) \cong cn''$. For the four remaining cases (five, since call-send splits in two dependent on whether the called OID is internal or not), 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 type 2 transition holds. This completes the argument. □