Formalising Java RMI with Explicit Code Mobility

Alexander Ahern, Nobuko Yoshida *

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK.

Abstract

This paper presents an object-oriented, Java-like core language with primitives for distributed programming and explicit code mobility. We apply our formulation to prove the correctness of several optimisations for distributed programs. Our language captures crucial but often hidden aspects of distributed object-oriented programming, including object serialisation, dynamic class downloading and remote method invocation. It is defined in terms of an operational semantics that concisely models the behaviour of distributed programs using machinery from calculi of mobile processes. Type safety is established using invariant properties for distributed runtime configurations. We argue that primitives for explicit code mobility offer a programmer fine-grained control of type-safe code distribution, which is crucial for improving the performance and safety of distributed object-oriented applications.

Key words: Distribution, Java, RMI, Types, Optimisation, Runtime, Code mobility

1 Introduction

Language features for distributed computing form an important part of modern object-oriented programming. It is now common for different portions of an application to be geographically separated, relying on communication via a network. Distributing an application in this way confers many advantages to a programmer such as resource sharing, load balancing, and fault tolerance.

* Corresponding author.
Email addresses: aja@doc.ic.ac.uk (Alexander Ahern), yoshida@doc.ic.ac.uk (Nobuko Yoshida).

However this comes at the expense of increased complexity for that programmer, who must now deal with concerns—such as network failure—that did not occur in centralised programs.

Remote procedure call mechanisms attempt to simplify distributed programming by providing a seamless integration of network resource access and local procedure calls, offering the developer a programming abstraction familiar to them. Java Remote Method Invocation [34] (RMI) is a widely adopted remote procedure call implementation for the Java platform, building on the customisable class loading system of the underlying language to hide code distribution from the programmer. When objects are passed as parameters to remote methods, if the provider of that method does not have the corresponding class file, it may attempt to obtain it automatically from the sender. Such code mobility is important as it eliminates the need for communicating parties to share implementation details, yet preserves the type safety of the system as a whole.

The implicit code mobility in RMI allows almost transparent use of remote objects and services. However when rigorously analysing the dynamics of distributed programs, or when providing developers with source-level control over code distribution [13], it becomes essential to model program behaviour explicitly. This is because elements such as distribution, network delay and partition crucially affect the behaviour and performance of programs and systems. As an example, communication-oriented RMI optimisations, often called *batched futures* [10] or *aggregation* [51,50], use code distribution as their central element. To analyse these optimisations formally, or to make the most of them in applications, explicit primitives for code mobility are essential.

This paper proposes a Java-like, object-oriented core language with communication primitives (RMI) and runtime features for distributed computation. The formalism exposes hidden runtime concerns such as code mobility, class downloading, object serialisation and communication. The operational semantics concisely models this behaviour using machinery from calculi of mobile processes [35,43,21]. One highlight is the use of a *linear* type discipline [29,20,54] to ensure correctness of remote method calls. Another is the application of several invariant properties. These are conditions that hold during execution of distributed programs, and they allow type safety to be established.

Our language supports *explicit code mobility* by providing primitives that allow programs to communicate fragments of code—closely related to closures in functional languages—for later execution. This subsumes the standard serialisation mechanism by sending not only data but also executable code. Code passing offers a programmer fine-grained control of type-safe code distribution, improving the flexibility and performance of their distributed applica-

tions without sacrificing safety. For example, a program fragment accessing a resource remotely could be expressed as a closure. This code could then be passed to the remote site, co-locating it with that resource. This effectively turns remote accesses into local accesses, reducing latency and increasing available bandwidth [13,10,51,50].

As an application of our formalism, we show that the RMI *aggregation* optimisations proposed in [51,50] are type- and semantics-preserving. The generality of the primitives we introduce plays an essential role in this analysis: one optimisation relies on the use of second-order code passing, i.e. passing code that in turn passes code itself. In distributed systems, optimisations like those we have explained naturally arise whenever latency and bandwidth are a limiting factor in the performance of programs, suggesting a wide applicability of this primitive in similar endeavours.

We summarise our major technical contributions below.

(1) Introduction of a core calculus for a class based typed object-oriented programming language with primitives for concurrency and distribution, including RMI, explicit code mobility, thread synchronisation and dynamic class downloading.
(2) An explicit characterisation of standard techniques for proving type preservation and progress theorems. We gather together all properties that must remain invariant during execution to establish these results. Not only are they essential for this purpose but are a useful analytical tool for for developing typing rules consistent with the actual implementation of Java RMI.
(3) Justification of several inter-node RMI optimisations employing explicit code mobility, using semantically sound syntactic transformations of the language and runtime. The analysis also demonstrates the greater control that explicit code mobility offers to programmers.

In the remainder, Section 2 informally motivates the present work through concrete examples of RMI optimisations. Section 3 introduces the syntax of the language. Sections 4 and 5 respectively discuss the dynamic semantics (reduction) and static semantics (typing) of the language. Section 6 establishes type preservation and progress properties. Section 7 studies contextual congruence of the core language and applies the theory to justify the optimisations in Section 2. Section 8 discusses related work. Section 9 concludes the paper with further topics.

This paper is the full version of [5], with complete definitions and detailed proofs. Its emphasis is on the formalisation of Java RMI, the proof of type safety for this formalisation using distributed runtime invariants, and the development of a theory of observational congruence for it. The present paper

also gives more examples on dynamic semantics of RMI and comparisons with related work.

2 Motivation: representing and justifying RMI optimisation

The RMI optimisations introduced in this section are used as running examples, culminating in their justification by the behavioural theory in § 7. These are (arguably) typical inter-node optimisations of distributed object-oriented programs. Just as inter-procedure or inter-module optimisations are hard to analyse, RMI optimisation poses a new challenge to the semantic analysis of distribution. They also motivate the use of explicit code mobility for fine-grained control of distributed behaviour and to improve performance.

Original RMI program 1 In optimisations for single location programs, we can aim to improve execution times by eliminating redundancy and ensuring that we exploit features of the underlying hardware architecture. In distributed programs these are still valid concerns, but other significant optimisations exist, in particular how latency and bandwidth overheads can be reduced. One typical example of this sort, centring on Java RMI [16] but which is generally applicable to various forms of remote communication, is *aggregation* [10,51,50]. We explain this idea using the simple program in Listing 1.

```
1 int m1(RemoteObject r, int a) {
2   int x = r.f(a);
3   int y = r.g(a, x);
4   int z = r.h(a, y);
5   return z;
6 }
```

Listing 1. Original RMI program 1

This program performs three remote method calls to the same remote object `r` with eight items transferred across the network (counting each parameter and return value as one). The result of the call to `f` at the remote server is stored in variable `x`, and is subsequently passed back to the server during the next call. The same occurs with the variable `y`. These variables are unused by the client, and are merely returned to the remote object `r` (where they were created) as parameters to the next call. We can immediately see that there is no need for `x` or `y` to ever be passed back to the client at all. Hence these three calls can be *aggregated* into a single call, reducing by a factor of three the network latency incurred by the method `m1` and approximately reducing by a factor of four the amount of data that must be shipped across the network.

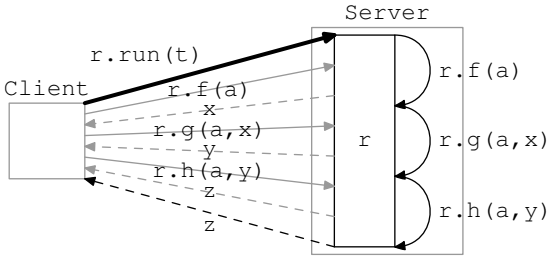
This optimisation methodology is implemented in the Veneer virtual Java Virtual Machine (vJVM) [51,50], where sequences of adjacent calls to the same remote object are grouped together into an execution *plan* in bytecode format. This is then uploaded to and executed by the server, with the result of the computation being returned to the client. This simple idea—remote evaluation of code [44]—can speed up distributed programs significantly, especially when operating across slower networks or when significant amounts of data may be transmitted otherwise. As a concrete example, in [51] the authors reported that over a moderate bandwidth and moderate latency ADSL connection, call aggregation yields a speedup over a factor of four for certain examples [16].

Optimised program 1 Call aggregation implicitly uses code passing: we first collect all the code that can be executed at a remote site and then send it, in one bundle, for execution there. This aspect is hidden as the transfer of bytecode in the implementation of [51,50], but requires explicit modelling if one wishes to discuss its properties or justify that it preserves the original program semantics. For this purpose we introduce two primitives, `freeze` and `defrost`. In Listing 2, we illustrate these primitives using the optimised version of the code of Listing 1.

```
1  // Client
2  int mOpt1(RemoteObject r, int a) {
3    thunk<int> t = freeze {
4      int x = r.f(a);
5      int y = r.g(a, x);
6      int z = r.h(a, y);
7      z;
8    };
9    return r.run(t);
10 }
11 // Server
12 int run(thunk<int> x) {
13   return defrost(x);
14 }
```

Listing 2. Optimised program 1

Here the client uses the `freeze` expression of the language to create a closure of the code for the three calls and the data held in variable `a`. The notation `thunk<int>` says that this code has type `int`, and now we see that the client has to make only one call across the network (to send the closure), calling `r.run(t)`. The receiving server evaluates it and returns the result typed by `int` to the client, again across the network. These mimic primitives found in well-known functional languages, for example the quotation and evaluation of code in Scheme, or the higher-order functions found in ML and Haskell.



Pale arrows	Original calls in the unoptimised program.
Dashed arrows	Returns from remote calls.
Thick arrows	Represent code mobility.

We annotate call arrows with the method invocation and return arrows with the name of the variable the client will use to store the return value of the method.

Fig. 1. Example optimisation (1)

In Figure 1 we show a diagram of the situation. As can be seen, the original sequence of calls (the paler arrows) requires 6 trips across the network. By aggregating the calls at the server, where they effectively become local, we see that only two trips are required (the thicker arrows).

Original RMI program 2 A more advanced communication optimisation, which reduces latency and uses bandwidth intelligently, is the idea of *server forwarding* [51,50]. It takes advantage of the fact that servers typically reside on fast connections, whilst the client-server connection can often be orders of magnitude slower. Consider the program in Listing 3.

```
1 int m2(RemoteObject r1, RemoteObject r2, int a) {
2   int x1 = r1.f1(a);
3   int y1 = r1.g1(a, x1);
4   int z1 = r1.h1(a, y1);
5   int x2 = r2.f2(z1);
6   int y2 = r2.g2(z1,x2);
7   int z2 = r2.h2(z1,y2);
8   return z2;
9 }
```

Listing 3. Original RMI program 2

The results of the first three calls are used as arguments to methods on another remote object `r2` in a second server. It would be better for the first server to communicate directly with the second.

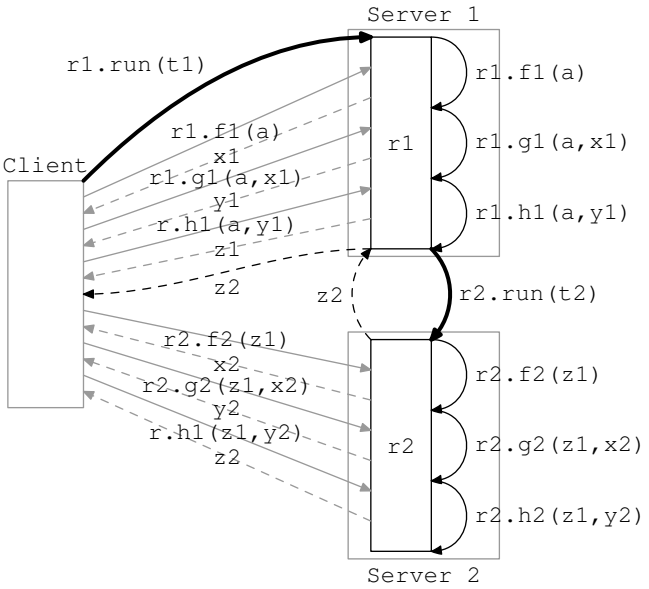


Fig. 2. Example optimisation (2)

Optimised program 2 Server forwarding again uses code passing as an execution mechanism. Listing 4 lists the optimised code of the original program in Listing 3. Unlike aggregation, server forwarding requires second order code passing, and so the optimised code contains nested occurrences of the `freeze` primitive. Figure 2 gives a diagram of the situation.

```
int mOpt2(RemoteObject r1, RemoteObject r2, int a) {
  thunk<int> t1 = freeze {
    int x1 = r1.f1(a);
    int y1 = r1.g1(a, x1);
    int z1 = r1.h1(a, y1);
    thunk<int> t2 = freeze {
      int x2 = r2.f2(z1);
      int y2 = r2.g2(z1, x2);
      int z2 = r2.h2(z1, y2);
      z2;
    };
    r2.run(t2);
  };
  return r1.run(t1);
}
```

Listing 4. Optimised program 2

Original RMI program 3 The semantics of RMI is different from normal, local method invocation. Passing a parameter to a remote method (or accepting a return value) can involve many operations hidden from the end-user; these runtime features make automatic semantic-preserving optimisation of

RMI much harder, in particular, when calls contain *objects as arguments*. To observe this, let us change the type of parameter `a` from `int` to class `MyObj` as in the code in Listing 5. This gives rise to two possibilities:

(1) `MyObj` is *remotely callable* i.e. when `MyObj` implements the `java.rmi.Remote` interface. In this situation, `a` is effectively passed by *reference*.
(2) `MyObj` is a regular class i.e. when instances of `MyObj` are *not remotely callable* (the class does not implement the `Remote` interface), `a` is automatically *serialised* and passed to the server where it is automatically *deserialised*. In this situation, `a` is effectively passed by *value*.

```
1 int m3(RemoteObject r, MyObj a) {
2   int x = r.f(a);
3   int y = r.g(a, x);
4   int z = r.h(a, y);
5   return z;
6 }
```

Listing 5. Original RMI program 3

Informally, the serialisation process explores the graph under an object in local memory, copying all objects directly or indirectly referred to. When passing such non-remotely callable objects as parameters to remote methods, the Java RMI system automatically performs this copying.

Consider the method `m3` above: if the call `r.f` performs an operation that side-effects the parameter `a`, then in the original program this side-effect is lost. The version of `a` supplied to the next method `r.g` is still just a copy of the initial `a` held in the client's memory, which has not changed. If we naïvely apply code passing optimisations to the problem, we might rewrite method `m3` to look a lot like `mOpt1`. Unfortunately now the next call `r.g` no longer has a copy of the original `a` to work on: it instead receives the version modified by `r.f`, potentially altering the meaning of the program and rendering the optimisation incorrect.

By applying explicit serialisation we can simulate the original program behaviour. By insisting each method call in the server operates on a fresh copy of the original `a`, we regain correctness as is shown below.

Optimised program 3 We show the case when `MyObj` is a class incapable of remote invocation. If there are no call-backs from the server to the client (discussed next), then the original RMI program has the same meaning as passing the code in Listing 6. First the client creates three copies of serialised object `a` by applying the explicit serialisation operator `serialize`. We write `serialize` as shorthand for the idiom in Java that involves writing objects to

an instance of `ObjectOutputStream`. The server immediately deserialises the arguments, creating three independent object graphs, thus avoiding problems with methods that alter their parameters (we write `deserialize` in place of reading from an `ObjectInputStream`). In the code in Listing 6, the declaration `ser<MyObj> b1` says that `b1` is a serialised representation of an object of class `MyObj`.

```
int mOpt3(RemoteObject r, MyObj a) {
  ser<MyObj> b1 = serialize(a);
  ser<MyObj> b2 = serialize(a);
  ser<MyObj> b3 = serialize(a);
  thunk<int> t = freeze {
    int x = r.f(deserialize(b1));
    int y = r.g(deserialize(b2), x);
    int z = r.h(deserialize(b3), y);
    z;
  };
  return r.run(t);
}
```

Listing 6. Optimised program 3

Two further problems We have seen that code passing primitives can help us to cleanly represent communication-based optimisation of RMI programs. Analysis of the code above immediately suggests two further problems that must be addressed.

(1) *Sharing between objects and call-backs:* the above copying method should not be applied naïvely, because it should preserve sharing between objects. It also may not be applicable if a call by the client to the server results in the server calling the client.
(2) *Overhead of class downloading:* if the server location does not contain the bytecode for `MyObj`, RMI automatically invokes a class downloading process to obtain the class from the network. In addition, verifying that the received class is safe to use (essentially by type-checking the bytecode at the time of dynamic linking) may require the downloading of many others (such as all superclasses of `MyObj` and classes mentioned in method bodies and so on), which may incur many trips across the network, increasing the risk of failures and adding latency.

To illustrate the first problem, consider the following simple code with the object identifier held in variable `r` remotely callable and the object identifiers held in `x` and `y` not:

```
x.f = y; r.h(x, y);
```

The content of `y` is shared with `x` in the original code, but if we apply the copying method then the server creates independent copies of `x` and `y`, breaking the original sharing structure.

For the second point of (1), imagine that the body of remote method `f` invoked at line 2 of the original program involves some communication back to the local site. Then it is possible for the value of `a` to be modified at the client side and so the optimised program is no longer correct: because in our optimised program, `a` is serialised in line 4 before `r.f` is performed, any effect that a call-back would have on `a` is lost, when it *should* be visible to the call `r.g`.

The second problem, overhead of class downloading, is more subtle from the communication-based optimisation viewpoint. Although the aim of this optimisation is to reduce the number of trips across the network, if there is a deep inheritance hierarchy above `MyObj`, sending code may not yield the performance benefit that the programmer expects. This is because many requests over the network may be required to obtain all the required classes.

As an example, if `MyObj` has a chain of n superclasses such that `MyObj` $<:$ `MyObj2` $<: \cdots <:$ `MyObjn`, and none of these are present at the server, there are at least n class downloads even with "verification off" in the framework of type safe dynamic linking [31,41]. With "verification on", this could be even more.

These hidden features of RMI make reasoning about the behaviour of a program, and establishing a clear optimisation strategy, hard.

Challenges Having provided the source-level presentation of several features necessary to discuss RMI optimisations, we may ask the following questions:

Q1. How can we precisely model this dynamic runtime behaviour, including code passing, serialisation and class downloading?
Q2. How can we verify the correctness of the optimised code, in the sense that the original code and the optimised code have the same contextual behaviour?
Q3. Having studied the optimisations above, can we improve our code mobility primitives to make them generally useful to application programmers?

A satisfactory solution to Q1. is a prerequisite for Q2. due to the interleaving of communication events which affect the observational behaviour of distributed programs. Various elements inherent in distributed computing make the semantic correctness of optimisations more subtle than in the local setting. The behaviour, hence the final answer, may differ depending on sharing of objects,

timing and class downloading strategies, as well as network failure. In our paper, Q1. will be answered by giving a clean formal semantics for distributed object-oriented features usually hidden from a programmer. We shall distill key runtime features, including class downloading and serialisation, so that important design choices (for example various class downloading and code mobility mechanisms) can be easily reflected in the semantics. Q2. will be answered by semantic justification of the above optimisations using the theory of mobile processes [35,43,21]. For Q3., we summarise our proposal below.

Optimised program 4 Class downloading is a fundamental mechanism in Java RMI programming. Yet so far we have treated it as a behind-the-scenes feature and left it as an implementation detail. However, by augmenting our primitives with a mechanism to control class downloading, a programmer is able to write down different strategies explicitly. This explicit control allows us to mitigate some of the problems class downloading induces that were explained in the previous section. For example, to represent one basic strategy of class downloading, we attach the tag `eager` to `freeze` in the original code 3 in Listing 5.

```
1 int mOpt4(RemoteObject r, MyObj a) {
2   ... // as in mOpt3
3   thunk<int> t = freeze[eager] {
4   ... // as in mOpt3
5 }
```

Listing 7. Optimised program 4

The tag `eager` in `freeze[eager]` controls the amount of class information sent along with the body of the closure by the user. With `eager`, the code is automatically frozen together with all classes that *may* be used. In the above case all classes appearing in `MyObj` and all their superclasses are shipped together with the code (see § 4.6 for the definition). Another option is for the user to select `lazy` which essentially leaves class downloading to the existing RMI system. Further the user might write a list of specific classes $\vec{C}$ to be shipped. For example, the program in Listing 8 is able to notice when it is in a high latency situation and act accordingly. If we imagine that the latency is very high, then it may be the case that the time to iteratively download all the superclasses exceeds the actual execution time of the closure being sent to the server. Because of this, the program is able to switch to the `eager` mode of class downloading, allowing improved performance. Moreover, from a point of view of failure there are fewer trips across the network with the eager policy, reducing the risk of a transient problem, such as a temporary network partition, disrupting the class downloading process.

```
// Client
thunk<int> t;
if (pingTime() > 1000) {// milliseconds
  t = freeze[eager] {...};
} else {
  t = freeze[lazy] {...};
}
```

Listing 8. Optimised program 5

The formal semantics for both implicit and explicit code mobility is given from the next section as part of the core language.

3 Language

3.1 User syntax

The syntax of the core language, which we call DJ, is an extension of FJ [25] and MJ [9], augmented with basic primitives for distribution and code-mobility, along with concurrent programming features that should be familiar to Java programmers. The syntax comes in two forms, and is given in Figure 3. The first form is called *user syntax*, and corresponds to terms that can be written by a programmer as source code. The second form is called *runtime syntax*. It extends the user syntax with constructs that only appear during program execution, and these are distinguished in the figure by placing them in shaded regions. We briefly discuss each syntactic category below.

Types T and U range over types for expressions and statements, which are explained in § 5. C, D, F range over class names. $\vec{f}$ denotes a vector of fields, and $\vec{T}\,\vec{f}$ is short-hand for a sequence of typed field declarations: $T_1 f_1; \ldots ; T_n f_n$. We assume sequences contain no duplicate names, and apply similar abbreviations to other sequences with ϵ representing the empty sequence. $T \to U$ denotes an *arrow type*, which is assigned to frozen expressions that expect a parameter of type T and return a value of type U. We abbreviate the type of thunked frozen expressions as $\texttt{thunk}\langle U\rangle \stackrel{\text{def}}{=} \texttt{unit} \to U$. We associate the type $\texttt{ser}\langle U\rangle$ with frozen *values*, an abbreviation defined as $\texttt{ser}\langle U\rangle \stackrel{\text{def}}{=} \texttt{unit} \to U$. If a value v has type U is frozen then the result has the type $\texttt{ser}\langle U\rangle$

Expressions The syntax is standard, including the standard synchronisation constructs of the Java language, except for two code passing primitives.

Syntax occurring only at runtime appears in shaded regions.

$$
\begin{array}{rrl}
\text{(Types)} & T & ::= \texttt{boolean} \mid \texttt{unit} \mid C \mid T \to U \\
\text{(Returnable)} & U & ::= \texttt{void} \mid T \\
\text{(Classes)} & L & ::= \texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} \\
\text{(Constructors)} & K & ::= C(\vec{T}\vec{f})\{\texttt{super}(\vec{f}); \texttt{this}.\vec{f} = \vec{f}\} \\
\text{(Methods)} & M & ::= U\ m(T\ x)\{e\} \\
\text{(Expressions)} & e & ::= v \mid x \mid \texttt{this} \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{while}\ (e)\ \{e\} \\
 & & \mid e.f \mid e; e \mid x = e \mid e.f = e \mid \texttt{new}\ C(\vec{e}) \mid e.m(e) \\
 & & \mid T\ x = e \mid \texttt{return}\ e \mid \texttt{return} \mid \texttt{freeze}[t](T\ x)\{e\} \\
 & & \mid \texttt{defrost}(e;\ e) \mid \texttt{fork}(e) \mid \texttt{sync}\ (e)\ \{e\} \mid e.\texttt{wait} \\
 & & \mid e.\texttt{notify} \mid e.\texttt{notifyAll} \mid \texttt{new}\ C^l(\vec{e}) \\
 & & \mid \texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e \mid \texttt{resolve}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e \\
 & & \mid \texttt{await}\ c \mid \texttt{sandbox}\ \{e\} \mid \texttt{insync}\ o\ \{e\} \\
 & & \mid \texttt{ready}\ o\ n \mid \texttt{waiting}(c)\ n \mid \texttt{return}(c)\ e \mid \texttt{Error} \\
\text{(Tags)} & t & ::= \texttt{eager} \mid \texttt{lazy} \mid \vec{C} \\
\text{(Values)} & v & ::= \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid () \\
 & & \mid o \mid \lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT}) \mid \epsilon \\
\text{(Class Sig.)} & \mathsf{CSig} & ::= \emptyset \mid \mathsf{CSig} \cdot [C \mapsto [\texttt{remoteable}]\ C\ \ \vec{T}\vec{f}\ \{\mathtt{m_i} : T_i' \to U_i\}] \\
\text{(Identifiers)} & u & ::= x \mid o \mid c \\
\text{(Threads)} & P & ::= \mathbf{0} \mid e \mid P_1 \,|\, P_2 \mid (\nu\,u)P \mid \texttt{forked}\ e \mid \texttt{go}\ e\ \texttt{with}\ c \\
 & & \mid e\ \texttt{with}\ c \mid \texttt{go}\ e\ \texttt{to}\ c \mid \texttt{Error} \\
\text{(Configurations)} & F & ::= (\nu\,\vec{u})(P, \sigma, \texttt{CT}) \\
\text{(Networks)} & N & ::= \mathbf{0} \mid l[F] \mid N_1 \,|\, N_2 \mid (\nu\,u)N \\
\text{(Stores)} & \sigma & ::= \emptyset \mid \sigma \cdot [x \mapsto v] \mid \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})] \\
\text{(Class tables)} & \texttt{CT} & ::= \emptyset \mid \texttt{CT} \cdot [C \mapsto L]
\end{array}
$$

Fig. 3. The syntax of the language DJ.

The first primitive, $\texttt{freeze}[t](T\ x)\{e\}$ takes the expression e and, without evaluating it, produces a flattened value representation parametrised by variable x with type T. Any parts of the local store required by the expression (such as the information held in variables free in e) are included in this new value, along with class information it may need for execution.

The tag t is a flag to control the amount of this information sent along with e by the user. If he specifies `eager`, then the code is automatically frozen together with all classes that *may* be used. If the user selects `lazy`, it is the responsibility of the receiving virtual machine to obtain missing classes. The third option is called *user-specified* information, and allows the programmer to supply a list of class names. Only these classes and their dependents (such as superclasses) are included with the frozen value.

Dual to freezing, the action $\texttt{defrost}(e_0;\ e)$ expects the evaluation of expression e to produce a piece of frozen code. This code is then executed, substituting its parameter with the value obtained by evaluating e_0, much like invoking a method. We abbreviate freeze and defrost expressions that take no parameters as $\texttt{freeze}[t]\{e\} \stackrel{\text{def}}{=} \texttt{freeze}[t](\texttt{unit}\ x)\{e\}$ ($x \notin \mathsf{fv}(e)$, where fv is a function returning the free variables of a term) and $\texttt{defrost}(e) \stackrel{\text{def}}{=} \texttt{defrost}(();\ e)$ respectively. Note that () denotes the constant of `unit` type.

To simplify the presentation, we only allow single parameters to methods and to frozen expressions. This does not restrict the expressiveness of programs written in DJ, as there is a semantics and type-preserving mapping from programs with multiple parameters to this subset. See Proposition 34 in § 7 for the formal proofs.

For clarity, we introduce two *derived* constructs that are syntactic sugar for serialisation and deserialisation.

$$\texttt{serialize}(e) \stackrel{\text{def}}{=} \texttt{freeze}[\texttt{lazy}]\{e\} \quad \text{and} \quad \texttt{deserialize}(e) \stackrel{\text{def}}{=} \texttt{defrost}(e)$$

Class Signatures A class signature CSig is a mapping from class names to their interface types (or signatures). We assume CSig is given globally, as a minimum interface agreed upon by remote parties, unlike class tables which are maintained on a per-location basis. Attached to each signature is the name of a direct superclass, as well as the declaration "`remoteable`" if instances of the class should be remotely callable (we frequently call such classes and instances "remoteable"). For a class C, the predicate $\mathsf{RMI}(C)$ holds iff "`remoteable`" appears in $\mathsf{CSig}(C)$. Class signatures contain only expected method signatures, not their implementation. This provides a lightweight mechanism for determining the type of remote methods.

Runtime syntax extends the user syntax to represent the distributed state of multiple sites communicating with each other, including remote operations in transit.

Expressions Location names are written $l, m, \ldots$ and can be thought of as IP addresses in a network. The expressions `new` $C^l(\vec{v})$, `download` $\vec{C}$ `from` l `in` e and `resolve` $\vec{C}$ `from` l `in` e define the machinery for class downloading, which will be explained along with the operational semantics in § 4.1 and § 4.2. The key expression is `new` $C^l(\vec{e})$, indicating that the definition of class C can be obtained from a location called l should it need to be instantiated. `await` c is fundamental to the model of method invocation and can be thought of as the return point for a call. `sandbox` $\{e\}$ represents the execution environment of some code e that originated from a frozen expression.

`insync` o $\{e\}$ denotes that expression e has previously acquired the lock on the object referenced by o. When an expression contains `ready` o n as a sub-term it indicates that it is ready to re-acquire the lock on object o. The expression `waiting`(c) n denotes an expression waiting for notification on channel c, at which point it may try to re-acquire a lock it was holding. n indicates the number of times that this waiting thread had entered its lock before yielding. Finally, the expression `Error` denotes the null-pointer error.

Values v is also extended with runtime terms. Object identifiers o denote references to instances of classes that may be available in the local store or held at a remote site. We shall often write "o-id" for brevity. Channels c are fundamental to the mechanism of method invocation in our formalism, and determine the return destination for both remote and local method calls, as illustrated in the operational semantics later. We call o and c *names*.

The most interesting extended value is a *frozen expression*, a piece of code or data that can be passed between methods as a value. Later, it can be "defrosted" at which point it is executed. $\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})$ denotes an expression e frozen with class table `CT` created at l. Expression e is parametrised by variable x with type T, and σ contains data local to the expression that was stored along with it at "freezing time". The identifiers $\vec{u}$ correspond to the domain of σ. x and $\vec{u}$ occur bound. `CT` ships class bodies that may be used during the execution of e. If it is empty and the party evaluating e lacks a required class, it attempts to download a copy from l. If σ or `CT` is empty, then we shall omit them entirely for clarity. Finally, the value ϵ serves as a constant that appears at runtime as the return value of `void` methods.

Threads $P \mid Q$ says P and Q are two threads running in parallel, while $(\nu\, u)P$ restricts identifier u local to P. $\mathbf{0}$ denotes an empty thread. This notation comes from the π-calculus [35]. It also includes `Error` which denotes the result of communication failure. The expression `forked` e says that expression e was previously forked from another thread. The remaining constructs of P are used for representing the RMI mechanism, and are illustrated when we discuss the operational semantics in § 4.

Configurations and Networks F represent an instance of a virtual machine. A configuration $(\nu\, \vec{u})(P, \sigma, \mathtt{CT})$ consists of threads P, a store σ containing local variables and objects, and a class table `CT`. Networks are written N, and comprise zero or more configurations held at named locations, executing in parallel. $\mathbf{0}$ denotes the empty network. $l[F]$ denotes a configuration F executing at location l. $N_1 \mid N_2$ and $(\nu\, \vec{u})N$ are understood as in threads.

A *store* σ is a mapping from variable names to values, written $[x \mapsto v]$, and from object identifiers to store objects, written $[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})]$. This indicates an identifier o maps to an object of class C with a vector of fields with values $\vec{f} : \vec{v}$. The set of channels $\{\vec{c}\}$ contains identifiers for threads currently waiting on o, i.e. those that have executed $o.\mathtt{wait}$ and have not received notification. The number, n, indicates how many times the lock on this object has been entered by a thread.

Finally, class tables `CT`, are a mapping from unlabelled class names to class definitions (metavariable L in Figure 3). Throughout the paper we write `FCT` for the *foundation class table* that contains the common classes that every location in the network should possess, roughly corresponding to the `java.*` classes. At the very minimum it contains the distinguished class *Object* that has no fields or methods and serves as the top of the inheritance hierarchy, i.e. $[\mathit{Object} \mapsto \mathtt{class}\ \mathit{Object}\ \{\ \}] \in \mathtt{FCT}$.

Auxiliary functions Several auxiliary functions are defined over the syntax of DJ. dom is defined over class tables and stores, and returns the domain of the mapping. We also write $\mathsf{dom}(F)$ etc. to denote the domain of stores which appear in F. The set of free variables $\mathsf{fv}(N)$ and names $\mathsf{fn}(N)$ are standard. We also use $\mathsf{fnv}(N) \stackrel{\mathsf{def}}{=} \mathsf{fv}(N) \cup \mathsf{fn}(N)$. The full definitions are given in the Appendix A.

The set of instantiated class names for a given term is given by the function icl which is defined over expressions, threads, configurations and class table entries. The instantiated class names of a value v is defined as $\mathsf{icl}(v) = \emptyset$. The instantiated class names of expressions and threads are defined recursively as the union of the instantiated class names of all sub-expressions, with the

exception that:

$$\mathsf{icl}(\texttt{new}\ C(\vec{e})) = \bigcup \mathsf{icl}(e_i) \cup \{C\} \quad \text{and importantly:} \quad \mathsf{icl}(\texttt{new}\ C^l(\vec{e})) = \bigcup \mathsf{icl}(e_i)$$

For class tables, we retrieve the instantiated class names appearing in the bodies of methods:

$$\mathsf{icl}(\texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}) = \bigcup \mathsf{icl}(e_i) \text{ where } M_i = U_i\ m_i(T_i'\ x_i)\{e_i\}$$

For stores, we set $\mathsf{icl}(\sigma) = \{C \mid [o \mapsto (C, \ldots)] \in \sigma\}$.

This function is necessary when we consider the code mobility inherent in both Java RMI and also in our explicit primitives. When shipping an object graph, icl provides a mechanism for determining the classes of the objects in that graph, so that the recipient of that graph knows which classes they need to download to safely incorporate those objects into the local store. In the case of the explicit primitives of DJ, since a closure may contain code that creates new objects, icl is required to label those instantiated classes to trigger automatic downloading.

4 Operational Semantics

This section presents the formal operational semantics of DJ, extending the standard small step call-by-value reduction of [40,9]. There are two reduction relations. The first is defined over configurations executing within an individual location, written $F \longrightarrow_l F'$, where l is the name of the location containing F. The second is defined over the networks, written $N \longrightarrow N'$. $F \longrightarrow_l F'$ promotes to $l[F] \longrightarrow l[F']$. Both relations are given modulo the standard structural equivalence rules of the π-calculus [35], written $\equiv$ and given in Appendix B. We define *multi-step* reductions as: $\twoheadrightarrow \stackrel{\text{def}}{=} (\longrightarrow \cup \equiv)^*$ and $\twoheadrightarrow_l \stackrel{\text{def}}{=} (\longrightarrow_l \cup \equiv)^*$.

First, we introduce the basic rules for evaluating networks and configurations in Figure 4 below.

4.1 Local expressions

The rules for the sequential part of the language are standard [25,9]. We list the reduction rules in Figure 5. When allocating a new object by New, we explicitly create a new restricted object identifier, which represents "freshness" or "uniqueness" of the address in the store. The auxiliary function $\mathsf{fields}(C)$

$$\text{RC-Par}\quad \frac{P_1, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1', \sigma', \mathtt{CT}') \qquad \vec{u} \notin \mathsf{fnv}(P_2)}{P_1 \mid P_2, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1' \mid P_2, \sigma', \mathtt{CT}')} \qquad \text{RC-Str}\quad \frac{F \equiv F_0 \longrightarrow_l F_0' \equiv F'}{F \longrightarrow_l F'}$$

$$\text{RC-Res}\quad \frac{(\nu\,\vec{u})(P, \sigma, \mathtt{CT}) \longrightarrow_l (\nu\,\vec{u}')(P', \sigma', \mathtt{CT}')}{(\nu\,u\vec{u})(P, \sigma, \mathtt{CT}) \longrightarrow_l (\nu\,u\vec{u}')(P', \sigma', \mathtt{CT}')} \qquad \text{RN-Conf}\quad \frac{F \longrightarrow_l F'}{l[F] \longrightarrow l[F']}$$

$$\text{RN-Par}\quad \frac{N \longrightarrow N'}{N \mid N_0 \longrightarrow N' \mid N_0} \qquad \text{RN-Res}\quad \frac{N \longrightarrow N'}{(\nu\,u)N \longrightarrow (\nu\,u)N'} \qquad \text{RN-Str}\quad \frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

Fig. 4. Evaluation rules for networks and configurations

is given in Figure 6. It examines the class signature and returns the field declarations for C.

Tagged class creation takes place in NewR and NewL. The former rule is applied whenever execution attempts to instantiate an object of a tagged class whose body is not present in the local class table. Instead of immediately allocating a new object, it first attempts to download the actual body of the class from the labelled location. This is discussed in detail in § 4.2. NewL is applied when an attempt is made to instantiate a tagged class whose body is already available locally. In this case the statement reduces to a normal untagged instantiation. The predicate $\mathsf{comp}(C, \mathtt{CT})$, introduced in § def:complete later, holds only when C and all superclasses of C are present in the local class table $\mathtt{CT}$.

To reduce the number of computation rules, we make use of the evaluation contexts in Figure 7 and the congruence rule Cong. Contexts contain a single hole, written [ ] inside them. $E[e]$ represents the expression obtained by replacing the hole in context E with the ordinary expression e. The evaluation order of terms in the language is determined by the construction of these contexts.

4.2 Class downloading

Class mobility is very important in Java RMI systems, since it reduces unnecessary coupling between communicating parties. If an interface can be agreed on, then any class that implements the interface can be passed to a remote consumer and type-safety will be preserved. However this only works if sites are able to dynamically acquire class files from one another. This hidden behaviour is omitted from known sequential formalisms, as it is not required in the single-location setting, and so the formalisation of class downloading is one of the key contributions of DJ.

VAR
$x, \sigma, \texttt{CT} \longrightarrow_l \sigma(x), \sigma, \texttt{CT}$

COND
$\texttt{if (true)}\ \{e_1\}\ \texttt{else}\ \{e_2\}, \sigma, \texttt{CT} \longrightarrow_l e_1, \sigma, \texttt{CT}$
$\texttt{if (false)}\ \{e_1\}\ \texttt{else}\ \{e_2\}, \sigma, \texttt{CT} \longrightarrow_l e_2, \sigma, \texttt{CT}$

WHILE
$\texttt{while}\ (e_1)\ \{e_2\}, \sigma, \texttt{CT} \longrightarrow_l \texttt{if}\ (e_1)\ \{e_2; \texttt{while}\ (e_1)\ \{e_2\}\}\ \texttt{else}\ \{\epsilon\}, \sigma, \texttt{CT}$

FLD
$$\frac{\sigma(o) = (C, \vec{f} : \vec{v})}{o.f_i, \sigma, \texttt{CT} \longrightarrow_l v_i, \sigma, \texttt{CT}}$$

ASS
$x = v, \sigma, \texttt{CT} \longrightarrow_l v, \sigma[x \mapsto v], \texttt{CT}$

FLDASS
$$\frac{\sigma' = \sigma[o \mapsto \sigma(o)[f \mapsto v]] \qquad o \in \mathsf{dom}(\sigma)}{o.f = v, \sigma, \texttt{CT} \longrightarrow_l v, \sigma', \texttt{CT}}$$

NEW
$$\frac{\mathsf{fields}(C) = \vec{T}\,\vec{f}}{\texttt{new}\ C(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l (\nu\, o)(o, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset), \texttt{CT})}$$

NEWR
$$\frac{\neg\mathsf{comp}(C, \texttt{CT})}{\texttt{new}\ C^m(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l \texttt{download}\ C\ \texttt{from}\ m\ \texttt{in new}\ C^m(\vec{v}), \sigma, \texttt{CT}}$$

NEWL
$\texttt{new}\ C^m(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l \texttt{new}\ C(\vec{v}), \sigma, \texttt{CT} \quad \mathsf{comp}(C, \texttt{CT})$

DEC
$T\ x = v; e, \sigma, \texttt{CT} \longrightarrow_l (\nu\, x)(e, \sigma \cdot [x \mapsto v], \texttt{CT}) \quad x \notin \mathsf{dom}(\sigma)$

CONG
$$\frac{e, \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(e', \sigma', \texttt{CT}')}{E[e], \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(E[e'], \sigma', \texttt{CT}')} \quad \vec{u} \notin \mathsf{fnv}(E)$$

Fig. 5. Local expressions

The rules for class downloading in DJ are given in Figure 8 and approximately model the lazy downloading mechanism found in JDK 1.3 without verification [14]. The *download* expression is responsible for the transfer of class table entries from a remote site. DOWNLOAD defines the semantics for this operation. For a download request $\texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e$ we first produce $\vec{D}$ by removing the names of any classes locally available (and thus eliminating

Field lookup

$$\mathsf{fields}(\mathit{Object}) = \epsilon \qquad \frac{\mathsf{CSig}(C) = [\texttt{remoteable}]\ C\ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\mathtt{m_i} : T_i' \to U_i\} \quad \mathsf{fields}(D) = \vec{T''}\vec{f'}}{\mathsf{fields}(C) = \vec{T''}\vec{f'}, \vec{T}\vec{f}}$$

Method body lookup

$$\frac{\mathtt{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} \quad U\ m(T\ x)\{e\} \in \vec{M}}{\mathsf{mbody}(m, C, \mathtt{CT}) = (x, e)}$$

$$\frac{\mathtt{CT}(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} \quad U\ m(T\ x)\{e\} \notin \vec{M}}{\mathsf{mbody}(m, C, \mathtt{CT}) = \mathsf{mbody}(m, D, \mathtt{CT})}$$

Method signature lookup

$$\frac{\mathsf{CSig}(C) = [\texttt{remoteable}]\ C\ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\mathtt{m_i} : T_i' \to U_i\}}{\mathsf{mtype}(m_i, C) = \vec{T_i}' \to U_i}$$

$$\frac{\mathsf{CSig}(C) = [\texttt{remoteable}]\ C\ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\mathtt{m_i} : T_i' \to U_i\} \qquad m \notin \{\vec{m}\}}{\mathsf{mtype}(m, C) = \mathsf{mtype}(m, D)}$$

Fig. 6. Lookup functions

$$\begin{aligned} E ::= {} & [\,] \mid \texttt{if}\ (E)\ \{e\}\ \texttt{else}\ \{e\} \mid E.f \mid E; e \mid x = E \mid E.f = e \mid o.f = E \\ & \mid \texttt{new}\ C(\vec{v}, E, \vec{e}) \mid E.m(e) \mid o.m(E) \mid T\ x = E \mid \texttt{defrost}(e;\ E) \mid \texttt{defrost}(E;\ v) \\ & \mid \texttt{sync}\ (E)\ \{e\} \mid E.\texttt{wait} \mid E.\texttt{notify} \mid E.\texttt{notifyAll} \mid \texttt{new}\ C^l(\vec{v}, E, \vec{e}) \\ & \mid \texttt{sandbox}\ \{E\} \mid \texttt{insync}\ o\ \{E\} \mid \texttt{forked}\ E \mid \texttt{go}\ E\ \texttt{with}\ c \mid E\ \texttt{with}\ c \\ & \mid \texttt{go}\ E\ \texttt{to}\ c \mid \texttt{return}(c)\ E \end{aligned}$$

Fig. 7. Evaluation contexts

duplication). We then compute vector $\vec{F}$ from all the instantiated class names in the bodies of the classes in $\vec{D}$. Finally, the classes named in $\vec{D}$ are downloaded and added to the local class table. Any occurrence of a member of $\vec{F}$ in a newly downloaded class body is tagged with the name of the remote site (l_2 in this case). *Resolution*, defined by RESOLVE, is the process of examining classes for unmet dependencies and scheduling the download of missing

DOWNLOAD

$$\frac{\{\vec{D}\} = \{\vec{C}\} \setminus \mathsf{dom}(\mathtt{CT}_1) \qquad \{\vec{F}\} = \mathsf{icl}(\mathtt{CT}_2(\vec{D})) \qquad \mathtt{CT}' = \mathtt{CT}_2(\vec{D})[\vec{F}^{l_2}/\vec{F}]}{\begin{array}{c} l_1[E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_2\ \mathtt{in}\ e] \mid P, \sigma_1, \mathtt{CT}_1] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \\ \longrightarrow l_1[E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l_2\ \mathtt{in}\ e] \mid P, \sigma_1, \mathtt{CT}_1 \cup \mathtt{CT}'] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \end{array}}$$

RESOLVE

$$\frac{\{\vec{D}\} = \{D \mid \mathtt{CT}(C_i) = \mathtt{class}\ C_i\ \mathtt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} \text{ and } D \notin \mathsf{dom}(\mathtt{FCT})\}}{\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l'\ \mathtt{in}\ e, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{download}\ \vec{D}\ \mathtt{from}\ l'\ \mathtt{in}\ e, \sigma, \mathtt{CT}}$$

DNOTHING

$$\mathtt{download}\ \emptyset\ \mathtt{from}\ l'\ \mathtt{in}\ e, \sigma, \mathtt{CT} \longrightarrow_l e, \sigma, \mathtt{CT}$$

ERR-CLASSNOTFOUND

$$\frac{\exists C_i \in \vec{C}.C_i \notin \mathsf{dom}(\mathtt{CT}_1) \cup \mathsf{dom}(\mathtt{CT}_2)}{\begin{array}{c} l_1[E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_2\ \mathtt{in}\ e] \mid P, \sigma_1, \mathtt{CT}_1] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \\ \longrightarrow l_1[E[\mathtt{Error}] \mid P, \sigma_1, \mathtt{CT}_1] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \end{array}}$$

Fig. 8. Evaluation rules for class resolution and downloading

classes. Informally this amounts to downloading immediate superclasses that are not members of the foundation class table.

The DOWNLOAD and RESOLVE rules work together to iteratively resolve all class dependencies for a given object. Once all dependencies have been met, normal execution continues after DNOTHING.

The rule ERR-CLASSNOTFOUND models a failure in this process, approximating the `ClassNotFoundException` that would occur in the case that the site l_2 does not possess some class requested by l_1. In this case, the code attempting the download will reduce to the `Error` expression.

In this paper we chose the option of class loading without verification as it allows significantly simpler presentation. However, our formalisation of class downloading is intended to be modular: it is possible to model different class loading mechanisms by adjusting the reduction rules for downloading and resolution and the class dependency algorithm introduced in Algorithm 2. For example, in rule RESOLVE the vector $\vec{D}$ is constructed from the direct superclasses of the classes being resolved. One aspect of Java verification is that it checks subtypes for method arguments. By inspecting the body of methods in the classes being resolved, we could extend $\vec{D}$ to reflect these checks as a first approximation.

Following on from this we observe that, with verification *on*, the overhead induced by Java's lazy class loading policy is increased—since verifying a class typically requires the loading of more classes than just the direct superclass—making an even stronger case for eager code passing.

4.3 Serialisation and deserialisation

One of the contributions of DJ is a precise formalisation of the semantics of serialisation using the frozen expressions which are detailed in § 4.6 (for the encoding, see § 3.1). Serialisation occurs in two instances. In the first, the expressions `serialize`(v) and `deserialize`(e) allow explicit flattening and re-inflation of objects by the programmer, whereas the second instance occurs when values must be transported across the network.

`serialize`(v) and `deserialize`(e) must appear automatically as runtime expressions to serialise parameters and return values of remote method invocations. This is because instances of non-remoteable classes—those classes without the `remoteable` keyword in their signature—are incapable of remote method invocation, and so cannot be passed by reference as parameters or as return values to remote methods. Should this occur, the remote party would receive the identifier of an unreachable object. Avoiding this problem involves making a deep clone of the object, and we see this in action in § 4.4.

4.4 Method invocation

Unlike sequential formalisms, DJ describes *remote method invocation*. To accommodate RMI, the rules for method call take a novel form employing concepts from the π-calculus, representing the context of a call by a local linear channel. While this technique is well-known in the π-calculus [35], DJ may be the first to use it to faithfully capture the semantics of RMI in a Java-like language. Among other benefits, it allows us to define the semantics of local and remote method calls concisely and uniformly: a method call is local when the receiver is co-located with the caller, whereas it becomes remote when the receiver is located elsewhere. Remote calls also differ from local ones because of the need for parameter and return value serialisation, which is reflected as several extra reduction steps.

We summarise the general picture of a remote method invocation in Figure 9, which starts from dispatch of a remote method and ends with delivery of its return value. The corresponding formal rules are given in Figure 10.

We start our illustration from local method calls. For a method call $o.m(v)$, if $o \in \mathsf{dom}(\sigma)$ then the rule MethLocal is applied. A new channel c is created to carry the return value of the method; the return point of the method call is replaced with the term `await` c corresponding to a receiver waiting for the return value supplied on channel c. The method call itself is spawned in a new thread as $o.m(v)$ `with` c carrying channel c.

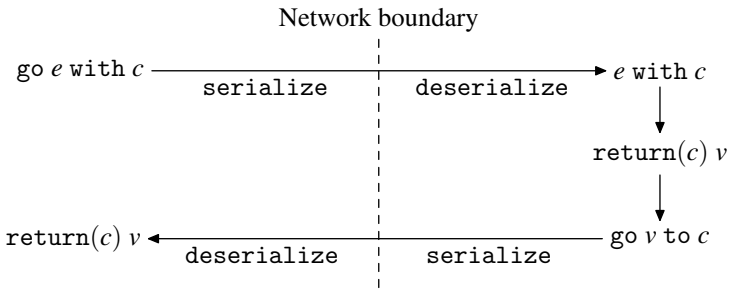


Fig. 9. Evaluation steps for a remote call

MethLocal

$$\frac{c \text{ fresh} \qquad o \in \mathsf{dom}(\sigma)}{E[o.m(v)] \mid P, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, c)(E[\mathtt{await}\ c] \mid o.m(v)\ \mathtt{with}\ c \mid P, \sigma, \mathtt{CT})}$$

MethRemote

$$\frac{c \text{ fresh}, o \notin \mathsf{dom}(\sigma)}{E[o.m(v)] \mid P, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, c)(E[\mathtt{await}\ c] \mid \mathtt{go}\ o.m(\mathtt{serialize}(v))\ \mathtt{with}\ c \mid P, \sigma, \mathtt{CT})}$$

MethInvoke

$$\frac{\sigma(o) = (C, \dots) \qquad \mathsf{mbody}(m, C, \mathtt{CT}) = (x, e)}{o.m(v)\ \mathtt{with}\ c, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, x)(e[o, \mathtt{return}(c)/\mathtt{this}, \mathtt{return}], \sigma \cdot [x \mapsto v], \mathtt{CT})}$$

Await

$$E[\mathtt{await}\ c] \mid \mathtt{return}(c)\ v, \sigma, \mathtt{CT} \longrightarrow_l E[v], \sigma, \mathtt{CT}$$

SerReturn

$$l[\mathtt{return}(c)\ v \mid P, \sigma, \mathtt{CT}] \longrightarrow l[\mathtt{go}\ \mathtt{serialize}(v)\ \mathtt{to}\ c \mid P, \sigma, \mathtt{CT}] \quad c \notin \mathsf{fn}(P)$$

Leave

$$\frac{o \in \mathsf{dom}(\sigma_2)}{l_1[\mathtt{go}\ o.m(v)\ \mathtt{with}\ c \mid P_1, \sigma_1, \mathtt{CT}_1] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \longrightarrow l_1[P_1, \sigma_1, \mathtt{CT}_1] \mid l_2[o.m(\mathtt{deserialize}(v))\ \mathtt{with}\ c \mid P_2, \sigma_2, \mathtt{CT}_2]}$$

Return

$$\frac{c \in \mathsf{fn}(P_2)}{l_1[\mathtt{go}\ v\ \mathtt{to}\ c \mid P_1, \sigma_1, \mathtt{CT}_1] \mid l_2[P_2, \sigma_2, \mathtt{CT}_2] \longrightarrow l_1[P_1, \sigma_1, \mathtt{CT}_1] \mid l_2[\mathtt{return}(c)\ \mathtt{deserialize}(v) \mid P_2, \sigma_2, \mathtt{CT}_2]}$$

Err-LostCall

$$\mathtt{go}\ o.m(v)\ \mathtt{with}\ c, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$$

Err-LostReturn

$$\mathtt{go}\ v\ \mathtt{to}\ c, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$$

Fig. 10. Remote method invocation

The next stage is the application of the method invocation rule MethInvoke. Both remote and local invocations apply this rule. The auxiliary function $\mathsf{mbody}(m, C, \mathtt{CT})$ is given in Figure 6, and is responsible for looking up the correct method body in the local class table. It returns a pair of the method code and the formal parameter name. The receiver is substituted $[o/\mathtt{this}]$ and a new store entry x is allocated for the formal parameter v. We apply the substitution $e[\mathtt{return}(c)/\mathtt{return}]$ to indicate that the return value of the method must be sent along channel c.

It is important to note that this last substitution preserves determinism, as an example:

$$\begin{aligned} & (\mathtt{if}\ (e)\ \{(e_1; \mathtt{return}\ v)\}\ \mathtt{else}\ \{(\mathtt{return}\ v')\})[\mathtt{return}(c)/\mathtt{return}] \\ \stackrel{\mathsf{def}}{=}\ & \mathtt{if}\ (e)\ \{(e_1; \mathtt{return}(c)\ v)\}\ \mathtt{else}\ \{(\mathtt{return}(c)\ v')\} \end{aligned}$$

The final stage of a method call is the application of rule Await, to communicate the return value to the caller.

When $o \notin \mathsf{dom}(\sigma)$ the method invocation is *remote*. The rule MethRemote is applied, with care being taken to automatically serialise the parameter v if it is an identifier for a non-remoteable object. We note that frozen values are also transferred to the remote location without modification (like base values).

After serialisation, we are left with a thread of the form $\mathtt{go}\ o.m(w)\ \mathtt{with}\ c$ where w is the serialised representation of the original parameter v. At this point, the network level rule Leave triggers the migration of the calling thread to the location that holds the receiving object in its local store. After transfer over the network, the parameter is automatically deserialised and MethInvoke applied. Again, the return value must be automatically serialised using SerReturn. Then it crosses the network by application of Return. After returning to the caller site, it is again deserialised.

The last two rules present instances of network failure. Network partition, that causes a remote method call to fail to reach its destination, is modelled by Err-LostCall. Likewise, in Err-LostReturn, the return value from a remote method call is lost. Both cases reduce to `Error`.

4.5 Multi-threaded programs

DJ contains several concurrency primitives that should be familiar to Java programmers, namely an implementation of *signal-and-continue* monitors [7]. The reduction rules are given in Figure 11, and we shall focus on the most important.

The rule FORK defines a simple command for creating a new thread. When evaluated, a new thread in the current location is started and begins executing an expression.

The rule SYNC defines a basic monitor construct. When we execute the term $E[\texttt{sync}\ (o)\ \{e\}]$, we are attempting to acquire the lock on the object identified by o. To determine whether a lock is taken, the function $\mathsf{lockcount}(\sigma, o)$ returns the number of times the monitor on object o in store σ has been re-entered by a thread. If this count is non-zero, then the predicate $\mathsf{insync}(o, E)$ is used to determine whether it is the current thread that owns the monitor (since Java allows re-entrant monitors). If this is the case then execution proceeds by incrementing the entry count using the function $\mathsf{setcount}(\sigma, o, n')$ with $n' = n + 1$, otherwise execution cannot continue. The predicates and functions are formally defined as follows.

Definition 1 (Lock functions and predicates)**.**

$$\mathsf{insync}(o, E) \iff \exists E_1, E_2 \text{ such that } E = E_1[\texttt{insync}\ o\ \{E_2[\ ]\}]$$

Suppose $\sigma(o) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$. Then we define:

$$\begin{aligned}
\mathsf{lockcount}(\sigma, o) &= n & \mathsf{blocked}(\sigma, o) &= \{\vec{c}\} \\
\mathsf{setcount}(\sigma, o, n') &= \sigma[o \mapsto (C, \vec{f} : \vec{v}, n', \{\vec{c}\})] \\
\mathsf{block}(\sigma, o, c) &= \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \cup \{c\})] \\
\mathsf{unblock}(\sigma, o, \vec{c}') &= \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \setminus \{\vec{c}'\})]
\end{aligned}$$

To temporarily release a lock held on an object o, the command $o.\texttt{wait}$ can be used with semantics as in WAIT. First, a new channel c is created and its name is added to the blocked set for the object o by application of the function $\mathsf{block}(\sigma, o, c)$. The currently executing thread then enters a sleeping state, written $\texttt{waiting}(c)\ n$ where n indicates the number of times the thread had entered the monitor on o before now. The lock count for o is then set to 0, allowing competing threads to acquire the lock.

To wake sleeping threads, the commands $o.\texttt{notify}$ and $o.\texttt{notifyAll}$ are provided. They differ in that the former non-deterministically wakes only one of the threads waiting on o, whereas the latter wakes them all. We shall focus on the rule NOTIFY. When notifying a thread, that thread must be waiting on some channel c which is held in the blocked set for o. This channel is then removed from the blocked set by the function $\mathsf{unblock}(\sigma, o, c)$. The woken thread then moves to the state of being $\texttt{ready}\ o\ n$, which means it is ready to re-acquire the lock on o, n times. It cannot *immediately* acquire this lock, since necessarily the thread that performs the notification is still within its

$$\textsc{Fork}\quad E[\texttt{fork}(e)], \sigma, \texttt{CT} \longrightarrow_l E[\epsilon] \mid \texttt{forked}\ e, \sigma, \texttt{CT}$$

$$\textsc{ThreadDeath}\quad \texttt{forked}\ v, \sigma, \texttt{CT} \longrightarrow_l \mathbf{0}, \sigma, \texttt{CT}$$

$$\textsc{Sync}\quad \frac{\mathsf{lockcount}(\sigma, o) = \begin{cases} 0 & \mathsf{setcount}(\sigma, o, 1) = \sigma' \\ n > 0 & \mathsf{insync}(o, E) \implies \mathsf{setcount}(\sigma, o, n+1) = \sigma' \end{cases}}{E[\texttt{sync}\ (o)\ \{e\}], \sigma, \texttt{CT} \longrightarrow_l E[\texttt{insync}\ o\ \{e\}], \sigma', \texttt{CT}}$$

$$\textsc{Wait}\quad \frac{\mathsf{lockcount}(\sigma, o) = n \qquad \mathsf{setcount}(\sigma, o, 0) = \sigma'' \qquad \mathsf{block}(\sigma'', o, c) = \sigma'}{E[o.\texttt{wait}] \mid P, \sigma, \texttt{CT} \longrightarrow_l (\nu\, c)(E[\texttt{waiting}(c)\ n] \mid P, \sigma', \texttt{CT})}$$

$$\textsc{Notify}\quad \frac{\mathsf{insync}(o, E) \qquad c \in \mathsf{blocked}(\sigma, o) \qquad \mathsf{unblock}(\sigma, o, c) = \sigma'}{E[o.\texttt{notify}] \mid E_1[\texttt{waiting}(c)\ n], \sigma, \texttt{CT} \longrightarrow_l E[\epsilon] \mid E_1[\texttt{ready}\ o\ n], \sigma', \texttt{CT}}$$

$$\textsc{NotifyAll}\quad \frac{\mathsf{insync}(o, E) \qquad \mathsf{blocked}(\sigma, o) = \{\vec{c}\} \qquad m \geq 0 \qquad \mathsf{unblock}(\sigma, o, \vec{c}) = \sigma'}{\begin{array}{c} E[o.\texttt{notifyAll}] \mid E_1[\texttt{waiting}(c_1)\ n_1] \mid \cdots \mid E_m[\texttt{waiting}(c_m)\ n_m], \sigma, \texttt{CT} \\ \longrightarrow_l E[\epsilon] \mid E_1[\texttt{ready}\ o\ n_1] \mid \cdots \mid E_m[\texttt{ready}\ o\ n_m], \sigma', \texttt{CT} \end{array}}$$

$$\textsc{NotifyNone}\quad \frac{\mathsf{insync}(o, E) \qquad \mathsf{blocked}(\sigma, o) = \emptyset}{E[o.\texttt{notify}], \sigma, \texttt{CT} \longrightarrow_l E[\epsilon], \sigma, \texttt{CT}}$$

$$\textsc{Ready}\quad \frac{\mathsf{lockcount}(\sigma, o) = 0 \qquad \mathsf{setcount}(\sigma, o, n) = \sigma'}{\texttt{ready}\ o\ n, \sigma, \texttt{CT} \longrightarrow_l \epsilon, \sigma', \texttt{CT}}$$

$$\textsc{LeaveCritical}\quad \frac{\mathsf{lockcount}(\sigma, o) = n \qquad \mathsf{setcount}(\sigma, o, n-1) = \sigma'}{\begin{array}{c} \texttt{insync}\ o\ \{v\}, \sigma, \texttt{CT} \longrightarrow_l v, \sigma', \texttt{CT} \\ \texttt{insync}\ o\ \{\texttt{return}(c)\ v\}, \sigma, \texttt{CT}, \longrightarrow_l \texttt{return}(c)\ v, \sigma', \texttt{CT} \end{array}}$$

Fig. 11. Concurrency primitives

critical section. However, as soon as that thread leaves its critical section the newly woken party can compete to acquire the lock.

4.6 Direct code mobility

Frozen expressions offer a direct way to manipulate code and data. They permit the storing of unevaluated terms that can, for example, be shipped to remote locations for evaluation or merely saved for future use. As we have seen in § 3.1, our formulation of the primitives subsumes the serialisation operations

FREEZE

$$\frac{\begin{array}{c}\sigma_y = \{[y \mapsto \sigma(y)] \mid y \in \mathsf{fv}(e) \setminus \{x\}\} \\ \sigma' = \mathsf{og}(\sigma, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y \qquad \{\vec{u}\} = \mathsf{dom}(\sigma') \\ \mathtt{CT}' = \begin{cases} \mathsf{cg}(\mathtt{CT}, \mathsf{icl}(e) \cup \mathsf{icl}(\sigma')) & t = \mathtt{eager} \\ \mathsf{cg}(\mathtt{CT}, \vec{C}) & t = \vec{C} \qquad \{\vec{F}\} = \mathsf{icl}(e) \setminus \mathsf{dom}(\mathtt{CT}') \\ \emptyset & t = \mathtt{lazy} \end{cases}\end{array}}{\mathtt{freeze}[t](T\ x)\{e\}, \sigma, \mathtt{CT} \longrightarrow_l \lambda(T\ x).(\nu\ \vec{u})(l, e[\vec{F}^l/\vec{F}], \sigma', \mathtt{CT}'), \sigma, \mathtt{CT}}$$

DEFROST

$$\frac{\{\vec{F}\} = \mathsf{icl}(\sigma') \setminus \mathsf{dom}(\mathtt{CT}')}{\begin{array}{l}\mathtt{defrost}(v;\ \lambda(T\ x).(\nu\ \vec{u})(m, e, \sigma', \mathtt{CT}')), \sigma, \mathtt{CT} \\ \longrightarrow_l (\nu\, x\vec{u})(\mathtt{download}\ \vec{F}\ \mathtt{from}\ m\ \mathtt{in}\ \mathtt{sandbox}\ \{e\}, \sigma \cup \sigma' \cdot [x \mapsto v], \mathtt{CT} \cup \mathtt{CT}')\end{array}}$$

LEAVESANDBOX

$$\mathtt{sandbox}\ \{v\}, \sigma, \mathtt{CT} \longrightarrow_l v, \sigma, \mathtt{CT}$$

Fig. 12. Creating and executing frozen expressions

found in Java that were explained in § 4.3.

As introduced in Figure 3, there are two operations associated with frozen values—for their creation and use—called *freezing* and *defrosting* respectively. Their rules are given in Figure 12.

Freezing is given by FREEZE, and has modes `lazy`, `eager`, and *user-specified*. Its operation is divided into two steps. The first step in any mode is to determine the store locations used by the expression e. We do this by examining the expression for any free variables, excluding the formal parameter x. The store entries for each variable are copied, σ_y. Next, we search for all the free object identifiers in e, written $\mathsf{fn}(e)$. Because variables may hold references to objects, we must then examine the store fragment σ_y for any object identifiers held in the co-domain of variable mappings. Finally, objects have internal structure, so we apply the object graph function given in Algorithm 1 to copy all non-remoteable objects transitively referenced by e or its variables, resulting in σ'. Base values stored in variables are copied "as-is".

In the second step the freezing mode matters because it directly affects the amount of class information included in $\mathtt{CT}'$. For the `lazy` case, no extra classes are provided with the expression, so the result of applying FREEZE is a value of the form $\lambda(T\ x).(\nu\ \vec{u})(l, e, \sigma, \emptyset)$.

When the case is `eager`, the creator of the frozen expression takes responsibility for including *all* classes that e depends upon. In the case that the user specifies a list of classes $\vec{C}$, only those classes and their dependencies are included. In either situation, we must use the class dependency algorithm in Algorithm 2 to determine the classes that the expression (or the user-specified

classes) rely upon.

4.7 Object graphs and class dependencies

4.7.1 Object graph

An object graph is essentially just a subset computed from a larger store, rooted at a specified object identifier o. Object graphs never contain instances of non-remoteable classes.

We define the predicate $\mathsf{reachable}(\sigma, o, o')$ to hold if there exists a path in store σ from the object with identifier o to the object with identifier o'. This can be an immediate link (when o' is stored in a field of o), or it can be via the fields of one or more intermediaries. It is extended in the natural way to threads by saying that an object identifier is reachable from a thread if it appears free in it, or is reachable via a free variable.

Definition 2 (Object graph reachability predicate)**.** Assume

$$\sigma(o') = (C, \vec{f} : \vec{v}) \text{ with } \neg\mathsf{RMI}(C)$$

Then:

$$\begin{aligned}
\mathsf{reachable}(\sigma, o', o) &\iff o \in \mathsf{fn}(\vec{v}) \vee \exists o'' \in \mathsf{fn}(\vec{v}).\mathsf{reachable}(\sigma, o'', o) \\
\mathsf{reachable}(\sigma, x, o) &\iff \sigma(x) = o \vee (\sigma(x) = o' \wedge \mathsf{reachable}(\sigma, o', o)) \\
\mathsf{reachable}(\sigma, P, o) &\iff \exists u \in (\mathsf{fn}(P) \cup \mathsf{fv}(P)).\mathsf{reachable}(\sigma, u, o) \vee o \in \mathsf{fn}(P)
\end{aligned}$$

The object graph algorithm is then responsible for copying all non-remoteable object instances reachable from some root o. For each copy it must set the lock count, n, to zero and empty the blocked set $\vec{c}$ to preserve linearity. Remoteable objects are not explored. It is defined as follows:

Algorithm 1 (Object graph calculation)**.** The function $\mathsf{og}(\sigma, v)$ computes the object graph of value v in store σ, and is defined as follows.

$$\begin{aligned}
\mathsf{og}(\sigma, v) &= \begin{cases} \emptyset & \text{if } v \notin \mathsf{dom}(\sigma) \vee \mathsf{RMI}(C) \\ [v \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)] \bigcup \mathsf{og}(\sigma_i, o_i) & \text{otherwise} \end{cases} \\
\mathsf{og}(\sigma, \vec{v}) &= \bigcup \mathsf{og}(\sigma, v_i)
\end{aligned}$$

where $\sigma(v) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$, $\{\vec{o}\} = \mathsf{fn}(\vec{v})$, $\sigma_1 = \sigma \setminus \{v\}$ and $\sigma_{i+1} = \sigma_i \setminus \mathsf{dom}(\mathsf{og}(\sigma_i, o_i))$.

See [4, Example 4.9] for an example of the algorithm. In the full Java language, fields may be marked `transient`. Such fields are never serialised (for example

they may contain a value that can be derived from other fields, or a reference to a non-serialisable object). Similarly, the Emerald language [24] supports a qualifier called "`attached`" that indicates which of an object's fields should be brought along it when it is copied. To support these extra features in DJ would involve the straightforward extension of syntax and a trivial modification to the object graph algorithm.

4.7.2 Class dependencies

An expression e directly depends upon a class C when $C \in \mathsf{icl}(e)$. e indirectly depends upon a class C when $\exists D \in \mathsf{icl}(e)$ and D is a subclass of C, or C is instantiated in the body of a method declared in D. Informally, dependency occurs when execution of an expression may at some point trigger instantiation of a class.

In order to calculate sets of dependencies we define an algorithm as follows:

Algorithm 2 (Class dependency set calculation)**.**

$$\mathsf{cg}(\mathtt{CT}, \vec{M}) = \bigcup \mathsf{cg}(\mathtt{CT}, \mathsf{icl}(e_i)) \text{ with } M_i = U_i\ m_i(T_i\ x_i)\{e_i\}$$

$$\mathsf{cg}(\mathtt{CT}, C) = \begin{cases} \emptyset & \text{if } C \notin \mathsf{dom}(\mathtt{CT}) \vee C \in \mathsf{dom}(\mathtt{FCT}) \\ \mathsf{cg}(\mathtt{CT}, \mathtt{CT}(C)) & \text{otherwise} \end{cases}$$

$$\mathsf{cg}(\mathtt{CT}, \vec{C}) = \bigcup \mathsf{cg}(\mathtt{CT}, C_i)$$

$$\mathsf{cg}(\mathtt{CT}, L) = \mathsf{cg}(\mathtt{CT} \setminus C, D) \cup\ \mathsf{cg}(\mathtt{CT} \setminus C, \vec{M}) \cup [C \mapsto L]$$

$$\text{where } L = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}$$

The following predicate describes an important property of class tables:

$$\mathsf{comp}(C, \mathtt{CT}) \stackrel{\mathsf{def}}{=} \forall D.C <: D.D \in \mathsf{dom}(\mathtt{CT})$$

which is read: class table `CT` is *complete with respect to* class C. When C is actually used, the class table `CT` at that location should be complete w.r.t. C. We extend the notion of completeness to entire class tables: we say a class table `CT` is *complete* if the following predicate holds:

$$\mathsf{ctcomp}(\mathtt{CT}) \stackrel{\mathsf{def}}{=} \forall D \in \mathsf{dom}(\mathtt{CT}).\mathsf{comp}(D, \mathtt{CT})$$

This means for every class $D \in \mathsf{dom}(\mathtt{CT})$, every superclass of D is also available in `CT`.

This subsection gives three small examples of the dynamic semantics, focusing on distributed primitives. We do not consider multi-threaded programming using the monitor constructs of DJ, so we omit the lock entry counter and waiting queue from store entries in this subsection.

Freeze and Defrost First we demonstrate freeze and defrost. After executing the program in Listing 9, at location l, we should obtain a frozen expression of the form:

$$\lambda(\texttt{int } x).(\nu\, o_1, o_2, y, a)(l, x + y + a.f, \sigma_1, \texttt{CT}_1)$$
$$\text{where } \sigma_1 = [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \ldots)] \cdot [y \mapsto 6] \cdot [a \mapsto o_1]$$
$$\text{and } \texttt{CT}_1 = [B \mapsto \ldots]$$

```
1 class A {
2   int f; B g;
3   A(int f, B g) { this.f = f; this.g = g; }
4 }
5 class B { }
6 // Program
7 int y = 6; A a = new A(5, new B());
8 freeze[B](int x){x + y + a.f};
```

Listing 9. Example of a program using freeze

To defrost a frozen value $\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma_1, \texttt{CT}_1)$ we apply DEFROST. Firstly, any classes supplied with the frozen value are appended to the current class table. The names of any classes instantiated in e are tagged with their originating location: `new` $C(\vec{e})$ becomes `new` $C^l(\vec{e})$. During execution of the newly defrosted code, when an expression such as the above `new` $C^l(\vec{v})$ is encountered then NEWR is applied if the body of C has not been downloaded to the execution location.

The second stage is to merge the data shipped with the value, σ_1, into the local store. It is not possible to merely append this to the local store, since this could cause a name clash (for example two entries for variable x in the same scope). Therefore we create new memory locations for the formal parameter of the frozen expression, as well as for every element in the domain of the accompanying store entries. This is written $(\nu\, x\vec{u})$. It is then safe to append the new store and allocate space for the formal parameter. We write the new store at the location as $\sigma \cup \sigma_1 \cdot [x \mapsto v]$.

The final aspect of the defrost rule is to download the classes for all the objects added to the store in the previous step, because we may have added instances of classes not present at this location. This means instead of immediately evaluating e we call $\texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in sandbox}\ \{e\}$. This accurately mimics the mechanism employed by the `RMIClassLoader` class used in Java RMI. When sending serialised objects, RMI implementations annotate the data stream for classes with a codebase URL. This is a pointer to a remote directory that the `RMIClassLoader` can refer to download classes that are not available at the current location.

After class downloading has completed, we are left with an expression of the form $\texttt{sandbox}\ \{e\}$. Execution inside the sandbox then proceeds until a value is computed, which is then propagated to the enclosing scope according to the rule LeaveSandbox. Take the frozen expression computed in the example previously and call it t. We now give another example of defrosting this time at a location m, where it is important to notice that a variable y is already in scope: the ν-operator will be used to avoid collision of bound variables and names. We abbreviate `download` to `dl` and `sandbox` to `sb` in the following:

$$
\begin{aligned}
& \texttt{defrost}(5;\ t), [y \mapsto \texttt{true}], \texttt{CT} \\
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y', a)(\texttt{dl}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y' + a.f\}, \sigma_2, \texttt{CT}_2) \\
\text{with } \sigma_2 = {} & [y \mapsto \texttt{true}] \cdot [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \ldots)] \\
& \cdot [y' \mapsto 6] \cdot [a \mapsto o_1] \cdot [x \mapsto 5] \\
\text{and } \texttt{CT}_2 = {} & \texttt{CT} \cdot [B \mapsto \ldots] \\
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y', a)(\texttt{resolve}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y' + a.f\}, \sigma_2, \texttt{CT}_3) \\
\text{with } \texttt{CT}_3 = {} & \texttt{CT}_2 \cdot [A \mapsto \ldots]
\end{aligned}
$$

Assuming that the superclass of A is *Object*, this should be already present in the local class table and hence downloading will not be required.

$$
\begin{aligned}
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y', a)(\texttt{dl}\ \emptyset\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y' + a.f\}, \sigma_2, \texttt{CT}_3) \\
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y', a)(\texttt{sb}\ \{x + y' + a.f\}, \sigma_2, \texttt{CT}_3) \\
\twoheadrightarrow_m\ & \texttt{sandbox}\ \{16\}, [y \mapsto \texttt{true}], \texttt{CT}_3 \\
\longrightarrow_m\ & 16, [y \mapsto \texttt{true}], \texttt{CT}_3
\end{aligned}
$$

In the final steps, we apply structural congruence rules (found in Appendix B) to "garbage-collect" the store entries added by the frozen expression since they are now no longer required.

Class downloading To illustrate the different class loading mechanisms, we change the above example as follows and investigate the cases when we change `B` in `freeze` to `eager` or `lazy`.

```
1 class A extends C{ ...}
```

```
class B { }
class C {D d(){return new D()}}
class D { }
```

Listing 10. Example of eager and lazy class downloading

- In the case of `eager`, the frozen expression ships all classes (`A,B,C,D`). Hence there is no downloading required after `defrost`.
- In the case of `lazy`, the frozen expression ships no classes. When defrosting, it downloads `A` and `B`. When resolving them at the next step, `A`'s superclass `C` is called to be downloaded. After `C` is downloaded, the final class table becomes $\texttt{CT}_5 = \texttt{CT}_3 \cdot [C \mapsto \texttt{class}\ C\ \{D\ \texttt{d()\{return new}\ D^l\texttt{()}\}\}]$. Note that `D` is *not* downloaded: hence it is renamed to $\texttt{D}^l$ so that if `D` requires instantiation, NewR will be applied and `D` downloaded from l.

Remote method invocation The last example is RMI. We replace class B in location l in Listing 9 with the one below. We assume location m has the remotely callable class R and that instances classes A and B are not capable of remote invocation.

```
// location l
class B { }
// Program
A a = new A(5, new B()); return r.f(a);
// location m
class R { Integer f(A x){ return x.f + 1} }
```

Listing 11. Example of remote method invocation

After the execution at the location l, we obtain the method invocation of the form:

$$
\begin{aligned}
&(\nu\, o_1, o_2)(\texttt{return}(d)\ r.f(o_1), \sigma_1', \texttt{CT}_1')\\
&\text{with } \sigma_2 = [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \ldots)]\\
&\text{and } \texttt{CT}_1' = [A \mapsto \ldots] \cdot [B \mapsto \ldots]
\end{aligned}
$$

This reduces to:

$$
\begin{aligned}
\longrightarrow_l\ & (\nu\, o_1, o_2, c)(\texttt{return}(d)\ \texttt{await}\ c \,|\, \texttt{go}\ r.f(\texttt{serialize}(o_1))\ \texttt{with}\ c, \sigma_1', \texttt{CT}_1')\\
\longrightarrow_l\ & (\nu\, o_1, o_2, c)(\texttt{return}(d)\ \texttt{await}\ c \,|\, \texttt{go}\ r.f(v)\ \texttt{with}\ c, \sigma_1', \texttt{CT}_1')\\
&\text{with } v = \lambda(\texttt{unit}\ x).(\nu\, o_1, o_2)(o_1, \sigma_1')
\end{aligned}
$$

When the remote method invocation happens, the argument o_1 is serialised and frozen value v is created. Now $\texttt{go}\ r.f(v)\ \texttt{with}\ c$ moves to the location m by LEAVE, opening the scope of c. After the message reaches the location m, v is deserialised, and starts to download classes A and B as follows. Below we assume $\texttt{CT}'_2 = [R \mapsto \ldots]$ and $\sigma'_2 = [r \mapsto (R, \ldots)]$:

$$
\begin{array}{rl}
 & r.f(\texttt{deserialize}(v))\ \texttt{with}\ c, \sigma'_2, \texttt{CT}'_2 \\
\longrightarrow_m & (\nu\, o_1, o_2)(r.f(o_1)\ \texttt{with}\ c, \sigma'_1 \cup \sigma'_2, \texttt{CT}'_2) \\
\longrightarrow_m & (\nu\, o_1, o_2)(r.f(\texttt{dl}\ A, B\ \texttt{from}\ l\ \texttt{in sb}\ \{o_1\})\ \texttt{with}\ c, \sigma'_1 \cup \sigma'_2, \texttt{CT}'_2) \\
\twoheadrightarrow_m & (\nu\, o_1, o_2)(r.f(o_1)\ \texttt{with}\ c, \sigma'_1 \cup \sigma'_2, \texttt{CT}'_2 \cup \texttt{CT}'_1) \\
\longrightarrow_m & (\nu\, o_1, o_2, x)(\texttt{return}(c)\ x.f + 1, \sigma'_1 \cup \sigma'_2 \cdot [x \mapsto o_1], \texttt{CT}'_2 \cup \texttt{CT}'_1) \\
\twoheadrightarrow_m & \texttt{return}(c)\ 6, \sigma'_2, \texttt{CT}'_2 \cup \texttt{CT}'_1 \\
\longrightarrow_m & \texttt{go}\ 6\ \texttt{to}\ c, \sigma'_2, \texttt{CT}'_2 \cup \texttt{CT}'_1
\end{array}
$$

Next "$\texttt{go}\ 6\ \texttt{to}\ c$" can safely return to the location l (since there exists only one $\texttt{await}\ c$) changing its form to "$\texttt{return}(c)\ 6$" by RETURN. Finally we get $\texttt{return}(d)\ 6$ by AWAIT, as required.

5 Typing System

This section presents the typing rules for DJ, focusing on the linear channel types and the use of invariants for typing runtime expressions and the new primitives. First we introduce the syntax of types and environments in Figure 13.

T represents expression types: booleans, class names, frozen expressions that take a parameter of type T and return elements of type U and the `unit` type. The metavariable U ranges over the same types as T but is augmented with the special type `void` with the usual meaning. We write $C <: D$ when class C is a subtype of class D. Our notion of subtyping is standard except for two points: firstly subtypes are judged on the class signature, and secondly we require that if $\mathsf{RMI}(C)$ holds for a class C, then this predicate must also hold for all superclasses of C. We also assume that the subtyping relation is acyclic as in [25,9] and give the definition in Figure 13. The arrow type is standard.

Two runtime types (which do not appear in programs) are newly introduced. *Return types* are ranged over by S are used to denote the type of value returned by a method invocation ($U\ m(T\ x)\{e\}$ is well-typed if e has the type $\texttt{ret}(U)$). *Channel types* are ranged over by metavariable τ and represents the types for channels used in method calls, which is explained in the next subsection.

There are two different kinds of environment. The environment for typing expressions, written Γ, is a finite map from variables, o-ids and `this` to types

$$
\begin{array}{lll}
T ::= \texttt{boolean} \mid \texttt{unit} \mid C \mid T \to U & & \text{(Types)} \\
U ::= \texttt{void} \mid T & & \text{(Returnable types)} \\
S ::= U \mid \texttt{ret}(U) & & \text{(Return types)} \\
\tau ::= \texttt{chan} \mid \texttt{chanI}(U) \mid \texttt{chanO}(U) & & \text{(Channel types)} \\
\Gamma ::= \emptyset \mid \Gamma, x : T \mid \Gamma, o : C \mid \Gamma, \texttt{this} : C & & \text{(Expression environment)} \\
\Delta ::= \emptyset \mid \Delta, c : \tau & & \text{(Channel environment)}
\end{array}
$$

The subtyping relation is defined as follows:

$$
T <: T \quad \frac{C <: D \qquad D <: F}{C <: F} \quad \frac{U_i' <: U_i \qquad 0 \le i < n}{\vec{U}' <: \vec{U}} \quad \frac{T' <: T \qquad U <: U'}{T \to U <: T' \to U'}
$$

$$
\frac{U' <: U}{\texttt{ret}(U') <: \texttt{ret}(U)} \qquad \frac{\mathsf{CSig}(C) = C \ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\texttt{m}_\texttt{i} : T_i' \to U_i\}}{C <: D}
$$

Fig. 13. Types, environments and subtyping

ranged over by T. Δ is a finite map from channel names to channel types, and appears in judgements for method calls and those involving multiple threads and locations. We often omit empty environments from judgements for clarity of presentation.

5.1 Linear channel types

One of the key tasks of the typing rules is to ensure *linear* use of channels. This means that for every channel c there is exactly one process waiting to input from c and one to output to c. In terms of DJ, this ensures that a method receiver always returns its value (if ever) to the correct caller, and that a returned value always finds the initial caller waiting for it. In Figure 13, $\texttt{chanI}(U)$ is *linear input* of a value of type U; $\texttt{chanO}(U)$ is the opponent called *linear output*. The type `chan` is given to channels that have matched input and output types. $\texttt{chanI}(U)$ is assigned to `await`, while $\texttt{chanO}(U)$ is to threads with/to c (either $\texttt{return}(c)\ e$, e `with/to` c, or `go` e `with/to` c).

To see the use of linear types, consider the following network; the return expression cannot determine the original location if we have two `await`s at the same channel c, violating the linearity of c.

$$
l_1[E_1[\texttt{await}\ c], \sigma_1, \texttt{CT}_1] \,|\, l_2[E_2[\texttt{await}\ c], \sigma_2, \texttt{CT}_2] \,|\, l_3[\texttt{go}\ v\ \texttt{to}\ c, \sigma_3, \texttt{CT}_3] \qquad (1)
$$

The uniqueness of the returned answer is also lost if return channel c appears twice.

$$l_1[\texttt{return}(c)\ e_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[\texttt{return}(c)\ e_2, \sigma_2, \texttt{CT}_2] \tag{2}$$

The aim of introducing linear channels is to avoid these situations during execution of runtime method invocations. The following binary operation $\asymp$ is used for controlling the composition of threads and networks.

Definition 3 (Channel environment composition)**.** The partial, commutative, binary composition operator on channel types, denoted $\odot$, is defined as $\texttt{chanI}(U) \odot \texttt{chanO}(U) \stackrel{\text{def}}{=} \texttt{chan}$. Then we define the composition of two channel environments $\Delta_1 \odot \Delta_2$ as:

$$\{c : \Delta_1(c) \odot \Delta_2(c) \mid c \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \cup \Delta_1 \setminus \mathsf{dom}(\Delta_2) \cup \Delta_2 \setminus \mathsf{dom}(\Delta_1)$$

Two channel types, τ and τ' are *composable* iff their composition is defined: $\tau \asymp \tau' \iff \tau \odot \tau'$ is defined. Similarly for $\Delta_1 \asymp \Delta_2$.

Note that $\odot$ and $\asymp$ are partial operators. Hence the composition of other combinations is not allowed. Once we compose linear input and output types, then it is typed by $\texttt{chan}$, hence it becomes uncomposable because $\texttt{chan} \not\asymp \tau$ for any τ. Intuitively if P is typed by environment Δ_1 and Q by Δ_2, and if $\Delta_1 \asymp \Delta_2$, then we can compose P and Q as $P \,|\, Q$ safely, preserving channel linearity. Hence (1) is untypable because of $\texttt{chanI}(U) \not\asymp \texttt{chanI}(U)$ at c. (2) is too by $\texttt{chanO}(U) \not\asymp \texttt{chanO}(U)$ at c.

5.2 Well-formedness

Well-formedness is defined for types, environments, stores and class tables. There are five kinds of judgement, and all are interrelated. In the following we assume α ranges over $S, U, [\texttt{remoteable}]\ C\ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\texttt{m}_\texttt{i} : T'_i \to U_i\}$ or τ, and we indicate in which Figure the rules used to derive the judgement can be found.

$\Gamma; \Delta \vdash \texttt{Env}$	$\Gamma; \Delta$ are well-formed environments (Figure 14).
$\vdash \alpha : \texttt{tp}$	α is a well-formed type (Figure 15).
$\Gamma; \Delta \vdash \sigma : \texttt{ok}$	σ is a well-formed store in $\Gamma; \Delta$ (Figure 16).
$\vdash \mathsf{CSig} : \texttt{ok}$	CSig is a well-formed signature (Figure 15).
$\vdash \texttt{CT} : \texttt{ok}$	$\texttt{CT}$ is a well-formed class table (Figure 17).

The judgements are standard. Note that CSig only contains well-formed types; and C is well-formed if its CSig entry is so.

$$\emptyset \vdash \texttt{Env} \qquad \frac{\Gamma \vdash \texttt{Env} \quad \vdash T : \texttt{tp} \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : T \vdash \texttt{Env}} \qquad \frac{\Gamma \vdash \texttt{Env} \quad \vdash C : \texttt{tp} \quad \texttt{this} \notin \mathsf{dom}(\Gamma)}{\Gamma, \texttt{this} : C \vdash \texttt{Env}}$$

$$\frac{\Gamma \vdash \texttt{Env} \quad \vdash C : \texttt{tp} \quad o \notin \mathsf{dom}(\Gamma)}{\Gamma, o : C \vdash \texttt{Env}} \qquad \frac{\Gamma \vdash \texttt{Env}}{\Gamma; \emptyset \vdash \texttt{Env}} \qquad \frac{\Gamma; \Delta \vdash \texttt{Env} \quad \vdash \tau : \texttt{tp} \quad c \notin \mathsf{dom}(\Delta)}{\Gamma; \Delta, c : \tau \vdash \texttt{Env}}$$

Fig. 14. Well-formedness for environments

$$\vdash \texttt{boolean} : \texttt{tp} \qquad \vdash \texttt{void} : \texttt{tp} \qquad \vdash \texttt{chan} : \texttt{tp} \qquad \vdash \texttt{remoteable}\ \mathit{Object}\ : \texttt{tp}$$

$$\frac{\vdash U : \texttt{tp} \vee U \in \mathsf{CSig}}{\vdash \texttt{chanI}(U) : \texttt{tp} \quad \vdash \texttt{chanO}(U) : \texttt{tp} \quad \vdash \texttt{ret}(U) : \texttt{tp}} \qquad \frac{\vdash T : \texttt{tp} \vee T \in \mathsf{dom}(\mathsf{CSig}) \quad \vdash U : \texttt{tp} \vee U \in \mathsf{dom}(\mathsf{CSig})}{\vdash T \to U : \texttt{tp}}$$

$$\frac{\forall C \in \mathsf{dom}(\mathsf{CSig}) \ \vdash C : \texttt{tp}}{\vdash \mathsf{CSig} : \texttt{ok}} \qquad \frac{\vdash \mathsf{CSig}(C) : \texttt{tp}}{\vdash C : \texttt{tp}}$$

$$\frac{\begin{array}{c}\vdash D : \texttt{tp} \qquad \forall S \in \{\vec{T'}, \vec{U}\} \ \vdash S : \texttt{tp} \vee S \in \mathsf{dom}(\mathsf{CSig}) \\ \mathsf{fields}(C) = \vec{T}\vec{f} \qquad m_i \in \mathsf{CSig}(D) \implies \mathsf{mtype}(m_i, C) <: \mathsf{mtype}(m_i, D) \\ \mathsf{RMI}(C) \implies \mathsf{RMI}(D)\end{array}}{\vdash [\texttt{remoteable}]\ C\ \texttt{extends}\ D\ \vec{T}\vec{f}\ \{\texttt{m}_\texttt{i} : T_i' \to U_i\} : \texttt{tp}}$$

Fig. 15. Well-formedness for types and class signatures

$$\frac{\Gamma; \Delta \vdash \texttt{Env}}{\Gamma; \Delta \vdash \emptyset : \texttt{ok}} \qquad \frac{\begin{array}{c}\Gamma; \Delta \vdash \sigma : \texttt{ok} \\ \Gamma \vdash x : T \qquad x \notin \mathsf{dom}(\sigma) \\ \Gamma \vdash v : T' \qquad T' <: T\end{array}}{\Gamma; \Delta \vdash \sigma \cdot [x \mapsto v] : \texttt{ok}}$$

$$\frac{\Gamma \vdash o : C \qquad \Gamma; \Delta \vdash \sigma : \texttt{ok} \qquad o \notin \mathsf{dom}(\sigma) \qquad \Gamma; \Delta \vdash (C, \vec{f} : \vec{v}, n, \{\vec{c}\}) : \texttt{ok}}{\Gamma; \Delta \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})] : \texttt{ok}} \qquad \frac{\begin{array}{c}\Gamma \vdash \vec{v} : \vec{T'} \qquad \vec{T'} <: \vec{T} \\ \mathsf{fields}(C) = \vec{T}\vec{f} \qquad n \geq 0 \\ \Gamma; \Delta \vdash c_i : \texttt{chanO}(\texttt{void})\end{array}}{\Gamma; \Delta \vdash (C, \vec{f} : \vec{v}, n, \{\vec{c}\}) : \texttt{ok}}$$

Fig. 16. Well-formedness for stores

5.3 Value and expression typing

Types are assigned to values and expressions using only the expression environment Γ. They have judgements of the form:

$$\Gamma \vdash e : \alpha \qquad\qquad e \text{ has type } \alpha \text{ in expression environment } \Gamma.$$

$$\frac{\texttt{this} : C, x : T \vdash e : \texttt{ret}(U') \qquad \mathsf{mtype}(m, C) = T \to U \qquad U' <: U}{\texttt{this} : T \vdash U\ m(T\ x)\{e\} : \texttt{ok in}\ C}$$

$$\frac{\texttt{this} : C \vdash \vec{M} : \texttt{ok in}\ C \qquad \mathsf{fields}(D) = \vec{T'}\vec{f'} \qquad \mathsf{fields}(C) = \vec{T}\vec{f} \qquad K = C\ (\vec{T'}\vec{f'}, \vec{T}\vec{f})\{\texttt{super}(\vec{f'}); \texttt{this}.\vec{f} = \vec{f}\}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} : \texttt{ok}}$$

$$\vdash \emptyset : \texttt{ok}$$

$$\frac{L = \texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\} \qquad \vdash L : \texttt{ok} \qquad \vdash \texttt{CT}' : \texttt{ok}}{\vdash \texttt{CT}' \cdot [C \mapsto L] : \texttt{ok}}$$

Fig. 17. Well-formedness for class tables

$$\text{TV-Basic}\ \frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \texttt{true} : \texttt{boolean}\quad \texttt{false} : \texttt{boolean}\quad () : \texttt{unit}\quad \epsilon : \texttt{void}}$$

$$\text{TV-Null}\ \frac{\Gamma \vdash \texttt{Env} \qquad \vdash C : \texttt{tp}}{\Gamma \vdash \texttt{null} : C}$$

$$\text{TV-Oid}\ \frac{\Gamma, o : C, \Gamma' \vdash \texttt{Env}}{\Gamma, o : C, \Gamma' \vdash o : C}$$

$$\text{TV-Frozen}\ \frac{\Gamma, x : T, \vec{u} : \vec{T} \vdash e : U \qquad \Gamma, \vec{u} : \vec{T}; \emptyset \vdash \sigma : \texttt{ok} \qquad \vdash \texttt{CT} : \texttt{ok}}{\Gamma \vdash \lambda(T\ x).(\nu\ \vec{u})(l, e, \sigma, \texttt{CT}) : T \to U}$$

Fig. 18. Typing rules for values

where α ranges over T, U and S. The typing rules for values are given in Figure 18 and for expressions in Figure 19. The typing judgement is local in the sense that it does not require knowledge about method bodies held at other locations, requiring only the declared signature of the method. This is possible by the use of the class signatures and invariants as explained later.

First we focus on the key typing rule for frozen expressions, TV-Frozen. In order for such a value to be well-typed we must ensure that the store σ and $\texttt{CT}$ are well-formed, and that the expression e computes a result of the expected type when supplied its formal parameter. The simplicity of this rule comes from the assumption that runtime values are created under the invariants defined in § 6. By combining with the invariants, we shall see:

- Instances of remotely callable classes are not contained in σ, i.e. if $o \in \mathsf{dom}(\sigma)$, then we have $\sigma(o) = (C, \ldots)$ with $\neg\mathsf{RMI}(C)$. This is guaranteed by the combination of invariants from $\mathsf{Inv}(4)$ to $\mathsf{Inv}(8)$ in § 6.1.2.
- The closure contains no free variables and no free identifiers for non-remotely callable objects: for example, by the combination of the invariants from $\mathsf{Inv}(4)$ to $\mathsf{Inv}(14)$ in § 6.1.4, we know $\sigma(o_i) = v_i$ is closed so that we can ensure that the resulting frozen value is closed again.

TE-VAR
$$\frac{\Gamma, x : T, \Gamma' \vdash \texttt{Env}}{\Gamma, x : T, \Gamma' \vdash x : T}$$

TE-THIS
$$\frac{\Gamma, \texttt{this} : C, \Gamma' \vdash \texttt{Env}}{\Gamma, \texttt{this} : C, \Gamma' \vdash \texttt{this} : C}$$

TE-COND
$$\frac{\exists S : S_1 <: S \wedge S_2 <: S \quad \Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash \texttt{if}\ (e)\ \{e_1\}\ \texttt{else}\ \{e_2\} : S}$$

TE-WHILE
$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \quad \Gamma \vdash e_2 : \texttt{void}}{\Gamma \vdash \texttt{while}\ (e_1)\ \{e_2\} : \texttt{void}}$$

TE-FLD
$$\frac{\Gamma \vdash e : C \quad \vdash C : \texttt{tp} \quad \mathsf{RMI}(C) \implies e = \texttt{this} \vee e = o \quad \mathsf{fields}(C) = \vec{T}\vec{f}}{\Gamma \vdash e.f_i : T_i}$$

TE-SEQ
$$\frac{\Gamma \vdash e_1 : U \quad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1; e_2 : S}$$

TE-ASS
$$\frac{\Gamma \vdash e : T' \quad T' <: T \quad \Gamma \vdash x : T}{\Gamma \vdash x = e : T'}$$

TE-FLDASS
$$\frac{\Gamma \vdash e.f : T \quad T' <: T \quad \Gamma \vdash e' : T'}{\Gamma \vdash e.f = e' : T'}$$

TE-NEW
$$\frac{\mathsf{fields}(C) = \vec{T}\vec{f} \quad T'_i <: T_i \quad \Gamma \vdash e_i : T'_i \quad \vdash C : \texttt{tp}}{\Gamma \vdash \texttt{new}\ C(\vec{e}) : C \qquad \texttt{new}\ C^l(\vec{e}) : C}$$

TE-METH
$$\frac{\mathsf{mtype}(m, C) = T \to U \quad \Gamma \vdash e_0 : C \quad \Gamma \vdash e : T' \quad T' <: T}{\Gamma \vdash e_0.m(e) : U}$$

Fig. 19. Typing rules for expressions

The conditions for the initial class table are more complicated as shall be explained in the next section.

5.3.1 Locality for field access and thread synchronisation

There are two important restrictions which we should impose in correspondence with the current Java implementation. The first constraint is to disallow field access and assignment to a remotely callable object in a different location. Hence the following example should be prohibited.

$$l[E[o.f]|P, \sigma_1, \texttt{CT}_1] \,|\, m[Q, \sigma_2 \cdot [o \mapsto (C, \ldots)], \texttt{CT}_2] \tag{3}$$

However we wish to allow to type the following with instances of class C remotely callable:

$$l[E[o.f]|P, \sigma_1 \cdot [o \mapsto (C, \ldots)], \texttt{CT}_1] \,|\, m[Q, \sigma_2, \texttt{CT}_2] \tag{4}$$

TE-Dec

$$\frac{\Gamma \vdash e : T \qquad T <: T' \qquad \Gamma, x : T \vdash e_0 : S}{\Gamma \vdash T'\ x = e; e_0 : S}$$

TE-Return

$$\frac{\Gamma \vdash e : U}{\Gamma \vdash \texttt{return}\ e : \texttt{ret}(U)}$$

TE-ReturnVoid

$$\frac{\Gamma \vdash \texttt{Env}}{\Gamma \vdash \texttt{return} : \texttt{ret}(\texttt{void})}$$

TE-Freeze

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \texttt{freeze}[t](T\ x)\{e\} : T \to U}$$

TE-Defrost

$$\frac{\Gamma \vdash e_0 : T' \qquad T' <: T \qquad \Gamma \vdash e : T \to U}{\Gamma \vdash \texttt{defrost}(e_0;\ e) : U}$$

TE-Fork

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash \texttt{fork}(e) : \texttt{void}}$$

TE-Sync

$$\frac{\mathsf{RMI}(C) \implies e = \texttt{this} \vee e = o \qquad \Gamma \vdash e_1 : C \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash \texttt{sync}\ (e_1)\ \{e_2\} : S}$$

TE-Monitor

$$\frac{\mathsf{RMI}(C) \implies e = \texttt{this} \vee e = o \qquad \Gamma \vdash e : C}{\begin{array}{c}\Gamma \vdash e.\texttt{wait} : \texttt{void} \\ e.\texttt{notify} : \texttt{void} \\ e.\texttt{notifyAll} : \texttt{void}\end{array}}$$

TE-ClassLoad

$$\frac{\Gamma \vdash e : U \qquad \vdash \vec{C} : \texttt{tp}}{\begin{array}{l}\Gamma \vdash \texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e : U \\ \quad \texttt{resolve}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e : U\end{array}}$$

TE-InSync

$$\frac{\Gamma \vdash o : C \qquad \Gamma \vdash e : S}{\Gamma \vdash \texttt{insync}\ o\ \{e\} : S}$$

TE-Sandbox

$$\frac{\Gamma \vdash e : U}{\Gamma \vdash \texttt{sandbox}\ \{e\} : U}$$

TE-Ready

$$\frac{\Gamma \vdash o : C \qquad n > 0}{\Gamma \vdash \texttt{ready}\ o\ n : \texttt{void}}$$

TE-Hole

$$\frac{\vdash U : \texttt{tp}}{\Gamma \vdash [\,]^U : U}$$

Fig. 19 (continued). Typing rules for expressions.

An early version of the work simply replaced the typing rule for field access with one that prevented it on any instance of a remoteable class. While safe, this was overly restrictive because no update to the fields of this object could take place, even at the location where it was held in store. Hence (4) above was untypable.

In order to propose a typing rule to prevent remote field access statically but allow field access on remotely callable objects locally, we require a combination of the locality invariants in § 6.1.2, the rule TE-Fld and also the initial conditions explained in Definition 8. The rule TE-Fld restricts field accesses

only for non-remoteable classes if e is neither `this` or o. The special expression `this` is allowed to have a remoteable class because `this` is always instantiated by an object identifier o that is present in the local store (see METHINVOKE). This constraint, together with our initial conditions guarantees that a thread making a field access is always co-located with the object it is updating.

The second restriction required to ensure that DJ corresponds with the existing Java RMI implementation is related to thread synchronisation. In Java there is no way to synchronise on a remote object identifier, instead it is merely possible to synchronise on the *stub* to a remote object. The behaviour of this operation is substantially different from what might be expected, since it does not acquire the lock on the underlying object held at the remote site and so does not prevent other clients in the network from accessing that resource. This is illustrated in the following example.

```
1  // Client 1 in Location 2
2  // ... import reference to r via RMI registry
3  synchronized (r) {
4    r.set(1);
5    return r.get();
6  }
7  // Client 2 in Location 3
8  // ... import reference to r via RMI registry
9  synchronized (r) {
10   r.set(2);
11   return r.get();
12 }
```

Listing 12. Incorrect synchronisation program

In the above, suppose we have a remoteable class which contains synchronised methods `set` and `get` in location 1 and two clients in locations 2 and 3. In this example the clients happen to be aware that their server is providing a shared resource, so they try to guarantee a "transaction" by "locking" the remote object. However this only locks the local stub objects, and does not prevent interleaving of operations: hence it is possible for client 1 to return 2 and client 2 to return 1. To avoid this situation by type-checking, we can just put the same condition as the field access as defined in TE-SYNC. By combining the invariants of locality, we can detect the above situation.

To implement a server-side locking solution would require engineering effort and an agreed protocol between clients. For instance, we consider a semaphore-style arrangement to guarantee the atomicity of a "transaction" in the following example:

```
1  // Client 1 in Location 2
2  // ... import reference to r via RMI registry
```

```
3  r.down();
4  r.set(1);
5  int v = r.get();
6  r.up();
7  return v;
8
9  // Client 2 in Location 3
10 // ... import reference to r via RMI registry
11 r.down();
12 r.set(2);
13 int v = r.get();
14 r.up();
15 return v;
```

Listing 13. Correct synchronisation program

This would require synchronised `down()` and `up()` methods to be installed in the remote object `r`, and would be very fragile since it relies on the good behaviour of clients to correctly signal the semaphore upon leaving the critical section. This option would be typable by our system, since it does not require synchronisation on the remote object `r`.

5.4 Thread and network typing

Threads, configurations and networks are assigned types under both the expression environment Γ and the channel environment Δ. The judgements take the following forms:

$\Gamma; \Delta \vdash P : \texttt{thread}$	P is a well-typed `thread` in environment $\Gamma; \Delta$.
$\Gamma; \Delta \vdash F : \texttt{conf}$	F is a wt. configuration in environment $\Gamma; \Delta$.
$\Gamma; \Delta \vdash N : \texttt{net}$	N is a wt. network in environment $\Gamma; \Delta$.

The typing rules for threads are given in Figure 20, and for configurations and networks in Figure 21. The most important rule for threads is TT-Par; we type a parallel compositions of threads if a composition of their respective channel environments preserves the linearity of channels. This is checked by $\Delta_1 \asymp \Delta_2$.

We must make a similar check in TC-Conf, since the blocked queue of threads waiting for locks requires the use of a channel environment to type the store σ. A configuration is then well-typed in an environment $\Gamma; \Delta_1 \odot \Delta_2$ if its threads, P, are well typed in the environment $\Gamma; \Delta_1$ and its store σ is well-typed under $\Gamma; \Delta_2$ with $\Delta_1 \asymp \Delta_2$. The class table must also be well-formed, and must contain a copy of the foundation classes `FCT`. The rule TN-Conf

TT-Nil
$$\frac{\Gamma;\emptyset \vdash \texttt{Env}}{\Gamma;\emptyset \vdash \mathbf{0} : \texttt{thread}}$$

TT-Par
$$\frac{\Gamma;\Delta_i \vdash P_i : \texttt{thread} \qquad \Delta_1 \asymp \Delta_2}{\Gamma;\Delta_1 \odot \Delta_2 \vdash P_1 \mid P_2 : \texttt{thread}}$$

TT-Await
$$\frac{\Gamma;\Delta \vdash E[\,]^U : \texttt{thread} \qquad c \notin \mathsf{dom}(\Delta)}{\Gamma;\Delta, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c]^U : \texttt{thread}}$$

TT-Res
$$\frac{\Gamma;\Delta, c : \texttt{chan} \vdash P : \texttt{thread}}{\Gamma;\Delta \vdash (\nu\, c)P : \texttt{thread}}$$

TT-Return
$$\frac{\Gamma \vdash e : \texttt{ret}(U') \qquad U' <: U}{\Gamma; c : \texttt{chanO}(U) \vdash e[\texttt{return}(c)/\texttt{return}] : \texttt{thread}}$$

TT-Waiting
$$\frac{\Gamma;\Delta \vdash E[\,]^{\texttt{void}} : \texttt{thread} \qquad c \notin \mathsf{dom}(\Delta) \qquad n > 0}{\Gamma;\Delta, c : \texttt{chanI}(\texttt{void}) \vdash E[\texttt{waiting}(c)\ n]^{\texttt{void}} : \texttt{thread}}$$

TT-Forked
$$\frac{\Gamma \vdash e : S}{\Gamma;\emptyset \vdash \texttt{forked}\ e : \texttt{thread}}$$

TT-GoSer
$$\frac{\Gamma \vdash o : C \qquad \mathsf{RMI}(C) \qquad \Gamma \vdash o.m(v) : U}{\Gamma; c : \texttt{chanO}(U) \vdash \texttt{go}\ o.m(\texttt{serialize}(v))\ \texttt{with}\ c : \texttt{thread}}$$

TT-MethWith
$$\frac{\Gamma \vdash o.m(v) : U}{\Gamma; c : \texttt{chanO}(U) \vdash o.m(v)\ \texttt{with}\ c : \texttt{thread}}$$

TT-DeserWith
$$\frac{\Gamma \vdash o : C \qquad T' <: T \qquad \mathsf{RMI}(C) \qquad \mathsf{mtype}(m, C) = T \to U \qquad \Gamma \vdash v : \texttt{unit} \to T'}{\begin{array}{c}\Gamma; c : \texttt{chanO}(U) \vdash o.m(\texttt{deserialize}(v))\ \texttt{with}\ c : \texttt{thread} \\ \texttt{go}\ o.m(v)\ \texttt{with}\ c : \texttt{thread}\end{array}}$$

TT-ValTo
$$\frac{\Gamma \vdash v : U' \qquad U' <: U}{\begin{array}{c}\Gamma; c : \texttt{chanO}(U) \vdash \texttt{go}\ \texttt{serialize}(v)\ \texttt{to}\ c : \texttt{thread} \\ \texttt{go}\ v\ \texttt{to}\ c : \texttt{thread}\end{array}}$$

Fig. 20. Thread and network typing rules

$$\textsc{TC-Weak}\ \frac{\Gamma;\Delta \vdash F : \mathtt{conf} \qquad c \notin \mathsf{dom}(\Delta)}{\Gamma;\Delta, c : \mathtt{chan} \vdash F : \mathtt{conf}} \qquad \textsc{TC-ResC}\ \frac{\Gamma;\Delta, c : \mathtt{chan} \vdash F : \mathtt{conf}}{\Gamma;\Delta \vdash (\nu\, c)F : \mathtt{conf}}$$

$$\textsc{TC-ResId}\ \frac{\Gamma, u : T;\Delta \vdash F : \mathtt{conf} \quad u \in \mathsf{dom}(F)}{\Gamma;\Delta \vdash (\nu\, u)F : \mathtt{conf}} \qquad \textsc{TC-Conf}\ \frac{\Gamma;\Delta_1 \vdash P : \mathtt{thread} \qquad \Gamma;\Delta_2 \vdash \sigma : \mathtt{ok} \quad \vdash \mathtt{CT} : \mathtt{ok} \qquad \mathtt{FCT} \subseteq \mathtt{CT} \qquad \Delta_1 \asymp \Delta_2}{\Gamma;\Delta_1 \odot \Delta_2 \vdash P, \sigma, \mathtt{CT} : \mathtt{conf}}$$

$$\textsc{TN-Nil}\ \frac{\Gamma;\emptyset \vdash \mathtt{Env}}{\Gamma;\emptyset \vdash \mathbf{0} : \mathtt{net}} \qquad \textsc{TN-Conf}\ \frac{\Gamma;\Delta \vdash F : \mathtt{conf}}{\Gamma;\Delta \vdash l[F] : \mathtt{net}}$$

$$\textsc{TN-Par}\ \frac{\Gamma;\Delta_i \vdash N_i : \mathtt{net} \qquad \mathsf{dom}(N_1) \cap \mathsf{dom}(N_2) = \emptyset \quad \Delta_1 \asymp \Delta_2 \qquad \mathsf{loc}(N_1) \cap \mathsf{loc}(N_2) = \emptyset}{\Gamma;\Delta_1 \odot \Delta_2 \vdash N_1 \mid N_2 : \mathtt{net}}$$

$$\textsc{TN-Weak}\ \frac{\Gamma;\Delta \vdash N : \mathtt{net} \quad c \notin \mathsf{dom}(\Delta)}{\Gamma;\Delta, c : \mathtt{chan} \vdash N : \mathtt{net}} \qquad \textsc{TN-ResId}\ \frac{\Gamma, u : T;\Delta \vdash N : \mathtt{net} \quad u \in \mathsf{dom}(N)}{\Gamma;\Delta \vdash (\nu\, u)N : \mathtt{net}} \qquad \textsc{TN-ResC}\ \frac{\Gamma;\Delta, c : \mathtt{chan} \vdash N : \mathtt{net}}{\Gamma;\Delta \vdash (\nu\, c)N : \mathtt{net}}$$

Fig. 21. Configuration and network typing rules

promotes configurations to the network level. For the rule TN-Par, we use the set of location names in a network N are given by the function $\mathsf{loc}(N)$, defined as: $\mathsf{loc}(\mathbf{0}) = \emptyset, \mathsf{loc}(l[F]) = \{l\}, \mathsf{loc}(N_1 \mid N_2) = \mathsf{loc}(N_1) \cup \mathsf{loc}(N_2)$ and $\mathsf{loc}((\nu\, u)N) = \mathsf{loc}(N)$.

6 Network Invariants and Type Preservation

We first introduce several runtime invariants and show that if an initial network satisfies certain conditions then reductions always preserve these runtime invariants. Next we establish subject reduction by the use of invariants. Finally combining subject reduction and invariants, we derive progress and other guarantees.

6.1 Network invariants

We start from the definition of a property over networks, given in Definition 4.

Definition 4 (Properties)**.** Let ψ denote a property over networks (i.e. ψ is a subset of the set of all networks). We write $N \models \psi$ if N satisfies ψ (i.e. if $N \in \psi$); we also write $N \not\models \psi$ if N does not satisfy ψ. We define the error property Err as the set of the networks which contain `Error` as subexpression, i.e. $\mathsf{Err} = \{N \mid N \equiv (\nu\, \vec{u})(l[E[\texttt{Error}] \mid P, \sigma, \texttt{CT}] \mid N')\}$. We say ψ is a *network invariant with an initial property* ψ_0 if $\psi = \{N \mid \exists N_0.(N_0 \models \psi_0,\ N_0 \twoheadrightarrow N,\ N \not\models \mathsf{Err})\}$

The following lemma is needed to formulate the network invariants of DJ. This concerns the *canonical forms*: every typeable network can be written in such a form. Intuitively, a canonical form is one in which all restricted identifiers are moved out to the network level.

Lemma 5 (Canonical forms)**.** *Suppose that* $\Gamma; \Delta \vdash N : \texttt{net}$ *then we have* $N \equiv (\nu\, \vec{u})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \texttt{CT}_i])$ *where* n *denotes the number of locations in* N*.*

Proof. By induction on the number of networks in parallel, n. □

The following lemma states that the typability is preserved under the structure rules. By this and the above lemma, we only have to consider the canonical forms for defining the network invariants.

Lemma 6 (Structural equivalence preserves typability)**.**

(1) If $\Gamma; \Delta \vdash F : \texttt{conf}$ *and* $F \equiv F'$ *then* $\Gamma; \Delta \vdash F' : \texttt{conf}$*.*
(2) Assume $\Gamma; \Delta \vdash P : \texttt{thread}$ *and* $P \equiv P'$*, then* $\Gamma; \Delta \vdash P' : \texttt{thread}$*.*
(3) If $\Gamma; \Delta \vdash N : \texttt{net}$ *and* $N \equiv N'$ *then* $\Gamma; \Delta \vdash N' : \texttt{net}$*.*

Proof. By induction on typing derivations paying attention to the last rule applied. See Appendix D.3. □

In order to ensure the correct execution of networks and type preservation, we require certain properties to remain invariant.

Definition 7 (Network invariants)**.** For network $N \equiv (\nu\, \vec{u})(\prod_{0 \le i < n} l_i[F_i])$ with $F_i = (P_i, \sigma_i, \texttt{CT}_i)$, and assuming $0 \le j < n$, $i \neq j$ where required, we define property $\mathsf{Inv}(r)$ as a set of networks which satisfy the condition r (with $1 \le r \le 17$) as defined below.

The majority of these properties fall into one of three important categories: *class availability*, *locality* and *linearity*. Each invariant has a clear operational (and arguably engineering) meaning.

6.1.1 Class availability

$\mathsf{Inv}(1)$ $\texttt{FCT} \subseteq \texttt{CT}_i$
$\mathsf{Inv}(2)$ $P_i \equiv E[\texttt{new}\ C(\vec{v})] \,|\, Q_i \implies \mathsf{comp}(C, \texttt{CT}_i)$
$\mathsf{Inv}(3)$ $C \in \mathsf{dom}(\texttt{CT}_i) \cap \mathsf{dom}(\texttt{CT}_j) \implies$
$\texttt{CT}_i(C) = \texttt{CT}_j(C) \vee \texttt{CT}_i(C) = \texttt{CT}_j(C)[\vec{D}^{l_i}/\vec{D}]$ with $\mathsf{icl}(\texttt{CT}_i(C)) = \{\vec{D}\}$

Key invariant properties in the presence of distribution are those of *class availability*. For example when a class is needed, it and all its superclasses must be present in the local class table. This requirement eliminates erroneous networks containing locations such as: $l[E[\texttt{new}\ C(\vec{v})], \sigma, \emptyset]$ where class C is not present in l's empty class table, so the initial step of execution will cause a crash. Note that even if C is present, if its superclass D is not then this is also an unexpected state. For example, in our system $\mathsf{Inv}(2)$ says that if we attempt to instantiate C, we need to have all its superclasses.

$\mathsf{Inv}(3)$ models the strict default class version control of the Java serialisation API. For example suppose we serialise an instance of the following class:

```
1 class A implements java.io.Serializable {
2   private int i;
3   private int j = 0;
4   A(int i) {
5     this.i = i;
6   }
7 }
```

If we then pass this to a remote consumer who has also has a class `A`, then deserialisation is not guaranteed to succeed, even if they have a binary compatible copy of the class:

```
1 class A implements java.io.Serializable {
2   private int i;
3   A(int i) {
4     this.i = i;
5   }
6 }
```

This is because it is impossible to recreate the original `A` at the new site without special low level programming. Moreover the `serialVersionUID`—a long integer hash value computed from the structure of a class file—will differ between the serialised object and the version of `A` held by the consumer [19]. [1]

[1] It is possible to override this value at the programmer level, however we do not consider such advanced techniques for versioning serialised objects.

6.1.2 Locality

$\mathsf{Inv}(4)$ $\mathsf{fv}(P_i) \subseteq \mathsf{dom}(\sigma_i) \subseteq \{\vec{u}\}$
$\mathsf{Inv}(5)$ $\mathsf{dom}(\sigma_i) \cap \mathsf{dom}(\sigma_j) = \emptyset$
$\mathsf{Inv}(6)$ $o \in \mathsf{fn}(F_i) \cap \mathsf{fn}(F_j) \implies \exists!k.\ \sigma_k(o) = (C, \ldots) \wedge \mathsf{RMI}(C)$
$\mathsf{Inv}(7)$ $o \in \mathsf{fn}(F_i) \wedge \exists k.\ \sigma_k(o) = (C, \ldots) \wedge \neg\mathsf{RMI}(C) \implies k = i$
$\mathsf{Inv}(8)$ $o \in \mathsf{fn}(F_i) \implies \exists k\ 1 \leq k \leq n.\ o \in \mathsf{dom}(\sigma_k)$
$\mathsf{Inv}(9)$ Suppose

$$R_i \in \{\ o.m(e)\ \texttt{with}\ c, E[o.f], E[o.f = e], E[\texttt{sync}\ (o)\ \{e\}],$$
$$E[\texttt{insync}\ o\ \{e\}], E[o.\texttt{notify}], E[o.\texttt{notifyAll}], E[o.\texttt{wait}], E[\texttt{ready}\ o\ n]\ \}$$

Then $P_i \equiv Q_i \mid R_i \implies \sigma_i(o) = (C, \ldots) \wedge \mathsf{comp}(C, \mathtt{CT}_i)$

An important property in the system is the locality of store entries such as local variables and object identifiers, captured by these invariants. For instance, combining $\mathsf{Inv}(4)$ and $\mathsf{Inv}(5)$, we can derive $\mathsf{fv}(P_i) \cap \mathsf{fv}(P_j) = \emptyset$, which ensures that local variables are not shared between threads at different locations. In $\mathsf{Inv}(9)$ we ensure that non-remote operations like field access and thread synchronisation are not attempted on remote object references. This particular situation highlights the necessity of the invariants, since we cannot guarantee this property alone in the typing system as we discussed in § 5.3.

6.1.3 Linearity invariants

Below we say, for some E and R, that *thread* P *inputs at* c if $P \equiv E[\texttt{await}\ c] \mid R$ or $P \equiv E[\texttt{waiting}(c)\ n] \mid R$; dually *thread* P *outputs at* c if $P \equiv R \mid Q$ with $R \equiv \texttt{return}(c)\ e$ or $R \equiv \texttt{go}\ e/e\ \texttt{with/to}\ c$ for some Q and e.

$\mathsf{Inv}(10)$ $P_i \equiv Q_i \mid R_i$ and Q_i inputs at $c \implies$ neither R_i nor P_j inputs at c.
$\mathsf{Inv}(11)$ $P_i \equiv Q_i \mid R_i$ and and Q_i outputs at $c \implies$ neither R_i nor P_j outputs at c.

Linearity of channel usage ensures the determinacy of method calls and returns and also the notification of blocked threads. This is ensured by the linear type checking.

6.1.4 Closure and lock invariants

Closures
$\mathsf{Inv}(12)$ $P_i \equiv E[v] \mid Q_i$ then $\mathsf{fv}(v) = \emptyset$
$\mathsf{Inv}(13)$ $\sigma_i(x) = v \implies \mathsf{fv}(v) = \emptyset$
$\mathsf{Inv}(14)$ $\sigma_i(o) = (C, \vec{f} : \vec{v}) \implies \mathsf{fv}(v_j) = \emptyset$

$\mathsf{Inv}(15)$ $P_i \equiv E[\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \mathtt{CT})] \,|\, Q_i$ and $\mathsf{fn}(\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \mathtt{CT})) = \{\vec{u'}\} \implies \exists k.\sigma_k(u'_j) = (C_j, \ldots)$ with $\mathsf{RMI}(C_j)$.

Locks

$\mathsf{Inv}(16)$ $P_i \equiv E[\mathtt{ready}\ o\ n] \,|\, Q_i \implies \mathsf{insync}(o, E) \wedge n > 0$

$\mathsf{Inv}(17)$ $P_i \equiv E[\mathtt{waiting}(c)\ n] \,|\, Q_i \implies \exists!o.c \in \mathsf{blocked}(\sigma_i, o) \wedge \mathsf{insync}(o, E) \wedge n > 0$

The *closure* invariants ensure that values and store entries do not contain any unbound variables. This is important to guarantee that newly created frozen expressions are similarly closed.

The *lock* invariants ensure the correct behaviour of the locking primitives at runtime. $\mathsf{Inv}(16)$ ensures that a thread that is ready to reacquire a lock will set that lock's count to a non-zero number. $\mathsf{Inv}(17)$ ensures that a thread does not wait for a non-existent lock.

6.2 Initial network

Before proving the network invariant, we define the initial network configurations. Roughly speaking an initial configuration contains no runtime values and expressions except o-ids, store objects and $\mathtt{return}(c)$; these latter two are required because in DJ we do not model resource lookup via a naming service, therefore the only way to obtain remote references is to supply them at the start of execution to participating locations. Similarly, we do not model the concept of a "main" method, therefore we allow locations to contain threads initially; these have notionally been generated by compiling multiple user-defined main programs. Definition 8 states these conditions formally.

Definition 8 (Initial network)**.** Network $N \equiv (\nu\,\vec{u})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ is called an *initial network* if it satisfies the following conditions (called *initial properties*):

- it contains no runtime expressions or values except o-ids, objects in the store, and parallel compositions of $\mathtt{return}(c)\ e$; and $\mathtt{freeze}[t](T\ x)\{e\}$ does not contain free o-ids, i.e. $\mathsf{fn}(e) = \emptyset$.
- it satisfies all properties $\mathsf{Inv}(i)$ except $\mathsf{Inv}(2)$, which is replaced by:
  - (a) $\mathsf{icl}(P_i) \subseteq \mathsf{dom}(\mathtt{CT}_i)$,
  - (b) $C \in \mathsf{icl}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_i) \implies \mathsf{comp}(C, \mathtt{CT}_i)$ and
  - (c) $\sigma_i(o) = (C, \ldots) \implies \mathsf{comp}(C, \mathtt{CT}_i)$.
- we also strengthen $\mathsf{Inv}(9)$ by replacing the reduction context E by an *arbitrary* context. This ensures that field accesses, assignments and so forth do not appear in *any* part of the initial program (not just in redex position).

We denote the set of networks satisfying these conditions by Init.

The extra requirement states that all initial class tables are complete w.r.t. classes in the program and stores. For example, suppose

$$\texttt{new}\ A().m(), \emptyset, \texttt{CT}$$

$$\text{with} \quad \texttt{CT}(A) = \texttt{class}\ A\ \texttt{extends}\ B\ \{;\ \texttt{void}\ m()\{\texttt{new}\ C(); \texttt{return}\}\}$$

First A should be defined in $\texttt{CT}$ (this is ensured by (a) in $\mathsf{Inv}(2')$); secondly B should be also defined in $\texttt{CT}$ (this is ensured by (a) and (b): since $A \in \mathsf{dom}(\texttt{CT})$, we have $\mathsf{comp}(A, \texttt{CT})$, which implies $B \in \mathsf{dom}(\texttt{CT})$); and thirdly, C should be defined in $\texttt{CT}$ too since $\texttt{new}\ C()$ appears after the method invocation at $\texttt{m}$. This condition is ensured by (b) since $C \in \mathsf{icl}(\texttt{CT})$. The condition (c) is similarly understood. We also note that during runs of programs, the initial properties may *not* be satisfied since classes can be downloaded lazily. Later we formalise this situation in Lemma 9 and prove the invariant $\mathsf{Inv}(2)$. The initial condition of $\mathsf{Inv}(9)$ is similarly understood as (c).

6.3 Type preservation and progress properties

To prove some cases of the subject reduction theorem, we require some invariants to hold in the assumptions. Therefore the proof routine for type preservation is divided into the following three steps:

Step 1 We prove one step invariant property for a typed network starting from the initial properties. This step has two sub-cases:
(i) *Assume* $\Gamma; \Delta \vdash N_0 : \texttt{net}$ *and* N_0 *is an initial network. Then* $N_0 \longrightarrow N_1$ *implies* $N_1 \models \mathsf{Inv}(r)$ *for each* $1 \leq r \leq 17$ *if* $N_1 \not\models \mathsf{Err}$.
(ii) *Assume* $\Gamma; \Delta \vdash N_m : \texttt{net}$ $(m \geq 1)$ *and* $N_m \models \mathsf{Inv}(r)$ *for all* $1 \leq r \leq 17$. *Then* $N_m \longrightarrow N_{m+1}$ *implies* $N_{m+1} \models \mathsf{Inv}(r)$ *for each* $1 \leq r \leq 17$ *if* $N_{m+1} \not\models \mathsf{Err}$.

Step 2 We prove the subject reduction theorem using Step 1, i.e. $\Gamma; \Delta \vdash N : \texttt{net}$ and $N \longrightarrow N'$ implies $\Gamma; \Delta \vdash N' : \texttt{net}$.

Step 3 Then invariant of $\mathsf{Inv}(r)$ is a corollary of Steps 1 and 2.

The proof of **Step 1** is given in the next subsection. Then assuming this holds, the proof of **Step 2** proceeds by induction on the derivation of reduction with a case analysis on the final typing rule applied. It is given in § 6.5.

This subsection lists the key additional invariants related to dynamic downloading of classes, synchronisation, and graph calculation which are used for the main proofs of the network invariants. We shall use the notation P_{im} to denote the threads at location i after m reduction steps. For proofs, see Appendix E.

Lemma 9 (Class table properties)**.** *Assume:*

$\Gamma;\Delta \vdash N_k : \mathtt{net}$ *for* $0 \leq k \leq m$, $\quad N_0 \in \mathsf{Init}$,
$N_k \in \mathsf{Inv}(r)$ *for* $k > 0$, $\quad 1 \leq r \leq 17$
$N_0 \twoheadrightarrow N_m \longrightarrow N_{m+1} \equiv (\nu\, \vec{u}_{m+1})(\prod_{0\leq i<n} l_i[P_{im+1}, \sigma_{im+1}, \mathtt{CT}_{im+1}])$ *with* $m > 0$

Then we have:

(1) $\mathtt{CT}_{im} \subseteq \mathtt{CT}_{im+1}$.
(2) $C \in \mathsf{icl}(P_{im+1})$ *implies* $C \in \mathsf{dom}(\mathtt{CT}_{im+1})$.
(3) Assume $\mathsf{reachable}(\sigma_{im+1}, P_{im+1}, o)$ *and* $\sigma_{im+1}(o) = (C, \ldots)$. *Then we have either*
  (a) $\mathsf{comp}(C, \mathtt{CT}_{im+1})$ *or*
  (b) either $P_{im+1} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im+1}$ *or*
  $P_{im+1} \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im+1}$
  where $\exists D \in \vec{C}.C <: D$ *and* $\neg\mathsf{reachable}(\sigma_{im+1}, Q_{im+1}, o)$

Lemma 9 says (1) the class table at each location always increases; (2) if a class is instantiated in a thread, then it is always in the domain of the class table; (3) if a free name or variable in P_{im+1} is reachable to o through store σ_{im+1}, then the class of o is complete otherwise it (or one of its superclasses) is currently being downloaded.

The next lemma states that the number of entries by a thread to an object's monitor is correctly accounted by said object.

Lemma 10 (Lock coherence)**.** *Assume* $\Gamma;\Delta \vdash N_k : \mathtt{net}(0 \leq k \leq m)$, N_0 *satisfies the initial network conditions and* $\mathsf{Inv}(r) \models N_k$ *for the invariants indexed over by* r. *Assume*

$$N_0 \longrightarrow N_1 \longrightarrow \cdots \longrightarrow N_{m+1} \equiv (\nu\, \vec{u}_{m+1})(\prod_{0\leq i<n} l_i[P_{im+1}, \sigma_{im+1}, \mathtt{CT}_{im+1}])$$

with $\mathsf{Err} \not\models N_k$. *Now:*

(1) Suppose $P_{im+1} \equiv E_1[\mathtt{insync}\ o\ \{\ldots E_p[\mathtt{insync}\ o\ \{e\}]\ldots\}] \,|\, Q_{im+1}$ *and* $e \neq E'[\mathtt{insync}\ o\ \{e'\}]$ *then:*

(a) $e \neq E[\texttt{waiting}(c)\ n']$ *with* $c \in \mathsf{blocked}(o, \sigma_{im+1})$ *and* $e \neq E[\texttt{ready}\ o\ n']$. *implies* $\mathsf{lockcount}(\sigma_{im+1}, o) = p$,
(b) $e = E[\texttt{ready}\ o\ n'] \implies p = n'$,
(c) $e = E[\texttt{waiting}(c)\ n']$ *with* $c \in \mathsf{blocked}(o, \sigma_{im+1})$ *implies* $p = n'$.

(2) Suppose $\mathsf{lockcount}(\sigma_{im+1}, o) = p$ *and* $p > 0$. *Then:*

$$P_{im+1} \equiv E_1[\texttt{insync}\ o\ \{\ldots E_p[\texttt{insync}\ o\ \{e\}]\ldots\}] \mid Q_{im+1}$$
and $e \neq E'[\texttt{insync}\ o\ \{e'\}]$

The final lemma is used to establish the correctness of the two algorithms introduced in § 4.7.

Lemma 11 (Correctness of algorithms)**.**

(1) Suppose $\Gamma; \Delta \vdash \sigma : \texttt{ok}$ *and* $\sigma' = \mathsf{og}(\sigma, o)$. *Then*
(a) If $\sigma(o) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$ *and* $\neg\mathsf{RMI}(C)$ *then* $[o \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)] \in \sigma'$.
(b) $\mathsf{reachable}(\sigma, o, o')$ *iff* $\mathsf{reachable}(\sigma', o, o')$.
(c) If $[o \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)] \in \sigma'$ *then* $[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})] \in \sigma$ *and* $\neg\mathsf{RMI}(C)$.
(d) $\Gamma; \emptyset \vdash \sigma' : \texttt{ok}$.
(2) Suppose $\vdash \texttt{CT} : \texttt{ok}$ *with* $C_i \in \mathsf{dom}(\mathsf{CSig})$. *Then we have* $\vdash \mathsf{cg}(\texttt{CT}, \vec{C}) : \texttt{ok}$.
(3) $\mathsf{ctcomp}(\texttt{CT})$ *and* $\texttt{CT}' = \mathsf{cg}(\texttt{CT}, C)$ *imply* $\mathsf{ctcomp}(\texttt{CT}' \cup \texttt{FCT})$.

Proof. See Appendix C. □

Summary of the one-step invariant proof We summarise the proofs of **Step 1** for each invariant. We use induction on the number of reduction steps, examining the last applied reduction rule. The proof requires a careful case analysis since several invariants and typing rules are mutually related. Below "we use $\mathsf{Inv}(r)$" means that "we assume $\mathsf{Inv}(r)$ holds at the inductive step m"; and "the case of the rule (r)" means that "the case when the last applied rule is (r)".

$\mathsf{Inv}(1)$ and $\mathsf{Inv}(2)$ use Lemma 9 (1). For $\mathsf{Inv}(3)$, we analyse the rules that change the class table: Download and Defrost. $\mathsf{Inv}(4)$ requires a case analysis on the three rules, Dec, MethInvoke and Defrost, with which the set of free variables of a term changes. For all cases, we use $\mathsf{Inv}(12)$. $\mathsf{Inv}(5)$ only requires examination of the case of Defrost. For $\mathsf{Inv}(6)$, we analyse MethRemote and Return, assuming $\mathsf{Inv}(8)$, $\mathsf{Inv}(5)$ and $\mathsf{Inv}(15)$. The interesting case for $\mathsf{Inv}(7)$ is when o newly appears at the $m+1$-step. We have four such cases, New, Defrost, Leave and MethRemote. For all cases, we use $\mathsf{Inv}(15)$. $\mathsf{Inv}(8)$ is mechanical by examination of the rules for structural equivalence. $\mathsf{Inv}(9)$ is one of the most non-trivial invariants. We derive it from Lemma 9 (1,

3), assuming $\mathsf{Inv}(2)$, $\mathsf{Inv}(8)$ and $\mathsf{Inv}(7)$ hold at the mth-step. $\mathsf{Inv}(10)$ and $\mathsf{Inv}(11)$ are straightforward by the definition of $\Delta_1 \asymp \Delta_2$. $\mathsf{Inv}(12)$ requires investigation of the cases where a value comes into a redex position. We have five cases, and use $\mathsf{Inv}(13)$ and $\mathsf{Inv}(14)$. For $\mathsf{Inv}(13)$, we check the cases where new variable mappings are added to the store, or when an existing mapping is changed. We have three cases, DEC, DEFROST and ASS, and all use $\mathsf{Inv}(12)$. $\mathsf{Inv}(14)$ needs to check NEW, DEFROST and FLDASS. All are straightforward by application of $\mathsf{Inv}(12)$. For $\mathsf{Inv}(15)$, the only interesting case is FREEZE, and we use Lemma 11. For $\mathsf{Inv}(16)$ and $\mathsf{Inv}(17)$, we use Lemma 10.

6.5 Proofs of type preservation

We first prove the following standard substitution lemma. Below, α denotes either U or T.

Lemma 12 (Substitution and context lemma)**.**

(1) Assume $\Gamma, x : T \vdash e : \alpha$ *and* $\Gamma \vdash v : T'$*. Suppose that* e *does not contain* $x = e'$ *or* $T\ x = e'$ *as its subterm. Then we have* $\Gamma \vdash e[v/x] : \alpha'$ *for some* $\alpha' <: \alpha$*.*
(2) $\Gamma, \mathtt{this} : C \vdash e : \alpha$ *and* $\Gamma \vdash o : C'$ *with* $C' <: C$ *imply* $\Gamma \vdash e[o/\mathtt{this}] : \alpha'$ *for some* $\alpha' <: \alpha$*.*
(3) $\Gamma \vdash E[\ ]^U : \alpha$ *and* $\Gamma \vdash e : U'$ *with* $U' <: U$ *iff* $\Gamma \vdash E[e]^U : \alpha$*.*

Proof. (1,2) By induction on the structure of expression e using Lemma 6. See Appendix F.1. (3) is by induction on the structure on E. All proofs are mechanical. □

Now we achieve the main theorem.

Theorem 13 (Subject reduction)**.**

(1) Assume $\Gamma, \vec{u} : \vec{T} \vdash e : \alpha$*,* $\Gamma, \vec{u} : \vec{T} \vdash \sigma : \mathtt{ok}$ *and* $\vdash \mathtt{CT} : \mathtt{ok}$*. Suppose* $(\nu\, \vec{u})(e, \sigma, \mathtt{CT}) \longrightarrow_l (\nu\, \vec{u}')(e', \sigma', \mathtt{CT}')$ *and* $e' \not\models \mathsf{Err}$*. Then we have* $\Gamma, \vec{u}' : \vec{T}' \vdash e' : \alpha'$ *for some* $\alpha' <: \alpha$*,* $\Gamma, \vec{u}' : \vec{T}' \vdash \sigma' : \mathtt{ok}$ *and* $\vdash \mathtt{CT}' : \mathtt{ok}$*.*
(2) Assume $\Gamma; \Delta \vdash F : \mathtt{conf}$*,* $F \longrightarrow_l F'$ *and* $F' \not\models \mathsf{Err}$*. Then we have* $\Gamma; \Delta \vdash F' : \mathtt{conf}$*.*
(3) Assume $\Gamma; \Delta \vdash N : \mathtt{net}$*,* $N \longrightarrow N'$ *and* $N' \not\models \mathsf{Err}$*. Then we have* $\Gamma; \Delta \vdash N' : \mathtt{net}$*.*

Proof. See Appendix F.2. □

Note that the above theorem guarantees type safety: if there is neither a null pointer error nor an unavoidable network error (i.e. $N' \not\models \mathsf{Err}$), then the typability ensures that an execution does not go wrong.

Corollary 14 (Network invariant)**.** $\wedge_{1\leq r\leq 17}\, \mathsf{Inv}(r)$ *is a network invariant with the initial network properties* Init *defined in Definition 8.*

6.6 Progress and Linearity Properties

Finally we can derive the following advanced progress and linearity properties.

Definition 15 (Progress invariants)**.** Assuming $0 \leq k < n$, then given network $N \equiv (\nu\, \vec{u})(\prod_{0\leq i<n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$, we define property $\mathsf{Prog}(r)$ as a set which satisfy the following conditions.

$\mathsf{Prog}(1)$ $P_i \equiv E[\mathtt{new}\ C(\vec{v})]\,|\,Q_i \implies \mathsf{comp}(C, \mathtt{CT}_i)$
Classes are always available for instantiation.

$\mathsf{Prog}(2)$ $P_i \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_k\ \mathtt{in}\ e]\,|\,Q_i \implies \vec{C} \in \mathsf{dom}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_k)$
Download operations always succeed in retrieving the required classes from the specified location.

$\mathsf{Prog}(3)$ $P_i \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ m\ \mathtt{in}\ e]\,|\,Q_i \implies \vec{C} \in \mathsf{dom}(\mathtt{CT}_i)$
No attempt is made to resolve classes that are not available in the local class table.

$\mathsf{Prog}(4)$ $P_i \equiv E[o.f_j]\,|\,Q_i \implies [o \mapsto (C, \ldots)] \in \sigma_i \wedge \mathsf{fields}(C) = \vec{T}\vec{f}$
No attempt is made to invoke a field access on the store if the class of the store does not provide that field.

$\mathsf{Prog}(5)$ $P_i \equiv E[o.f_j = v]\,|\,Q_i \implies [o \mapsto (C, \ldots)] \in \sigma_i \wedge \mathsf{fields}(C) = \vec{T}\vec{f}$
No attempt is made to invoke a field access on the store if the class of the store does not provide that field.

$\mathsf{Prog}(6)$ $P_i \equiv E[x]\,|\,Q_i \implies x \in \mathsf{dom}(\sigma_i)$
Expressions only access variables in the local store.

$\mathsf{Prog}(7)$ $P_i \equiv E[x = v]\,|\,Q_i \implies x \in \mathsf{dom}(\sigma_i)$
Expressions only assign to variables in the local store.

$\mathsf{Prog}(8)$ $P_i \equiv o.m(v)\ \mathtt{with}\ c\,|\,Q_i \wedge \sigma_i(o) = (C, \ldots) \implies \mathsf{mbody}(m, C, \mathtt{CT}_i)$ defined
No attempt is made to invoke a method on an object of a given class if that class does not provide that method.

$\mathsf{Prog}(9)$ $P_i \equiv \mathtt{go}\ o.m(v)\ \mathtt{with}\ c\,|\,Q_i \implies \exists!k.\ o \in \mathsf{dom}(\mathtt{CT}_k)$
Remote method calls always refer to a unique live location in the network.

$\mathsf{Prog}(10)$ $P_i \equiv \mathtt{go}\ v\ \mathtt{to}\ c\,|\,Q_i \wedge c \in \{\vec{u}\} \implies \exists!k.\ P_k \equiv E[\mathtt{await}\ c]\,|\,Q_k$
If a method return exists, there must be exactly one location waiting for it on

that channel.

Theorem 16 (Progress, locality and linearity)**.** $\wedge_{1\leq r\leq 10}\ \mathsf{Prog}(r)$ *is a network invariant with the initial network properties* Init *defined in Definition 8.*

Proof. Immediately $\mathsf{Prog}(1)$ is derived from $\mathsf{Inv}(2)$. $\mathsf{Prog}(2)$ is by the monotonicity of the class tables proved in Lemma 9 (1). $\mathsf{Prog}(3)$ is obvious by DOWNLOAD. $\mathsf{Prog}(4)$ and $\mathsf{Prog}(5)$ are proved by $\mathsf{Inv}(9)$. $\mathsf{Prog}(6)$ and $\mathsf{Prog}(7)$ are obvious by $\mathsf{Inv}(4)$. $\mathsf{Prog}(8)$ is derived from $\mathsf{Inv}(9)$. $\mathsf{Prog}(9)$ is by combining $\mathsf{Inv}(8)$ and $\mathsf{Inv}(5)$. $\mathsf{Prog}(10)$ is straightforward by $\mathsf{Inv}(10)$ and $\mathsf{Inv}(11)$.

□

6.7 Progress with synchronisation and normal forms

In this final subsection of § 6, we first investigate a simple progress property in the presence of the synchronisation primitives. Then we will show the normal forms of DJ—the shape of whole networks when all computation has terminated. We start from the first proposition which says that one monitor is held by only one thread.

Proposition 17 (Mutual exclusion)**.** *For a location* $l[P, \sigma, \mathtt{CT}]$*, suppose*

$$\begin{array}{rl} & P \equiv E_1[\mathtt{insync}\ o\ \{e_1\}] \mid \cdots \mid E_n[\mathtt{insync}\ o\ \{e_n\}] \mid Q \\ \textit{then} & \forall j.1 \leq j \leq n.\ (e_j = E'_j[\mathtt{waiting}(c)\ ...] \vee e_j = E'_j[\mathtt{ready}\ o\ \ldots]) \\ \textit{or} & \exists! j.1 \leq j \leq n.\ (e_j \neq E'_j[\mathtt{waiting}(c)\ \ldots] \wedge e_j \neq E'_j[\mathtt{ready}\ o\ \ldots]) \end{array}$$

with $c \in \mathsf{blocked}(o, \sigma)$.

Proof. See Appendix E.4. □

Below we list a simple progress property. (1) states if expression e which holds a monitor is neither error nor a synchronisation expression, then e can always progress; and (2) says that a thread can progress from `ready` only if other threads holding that monitor are waiting or ready themselves.

Proposition 18 (Progress with synchronisation)**.** *For a location* $l[P, \sigma, \mathtt{CT}]$*,*

(1) Suppose $P \equiv E[\mathtt{insync}\ o\ \{E'[e]\}] \mid Q_i$*,* $e \mid Q, \sigma, \mathtt{CT} \longrightarrow e' \mid Q', \sigma', \mathtt{CT}'$*, and* $e \notin \{\mathtt{insync}\ o'\ \{e'\}, \mathtt{waiting}(c)\ n, \mathtt{ready}\ o\ n, \mathtt{sync}\ (o')\ \{e'\}, \mathtt{Error}\}$*. Then* $E[\mathtt{insync}\ o\ \{E'[e]\}] \mid Q, \sigma, \mathtt{CT} \longrightarrow E[\mathtt{insync}\ o\ \{E'[e']\}] \mid Q', \sigma', \mathtt{CT}'$*.*

(2) Suppose $P \equiv E[\texttt{ready}\ o\ n] \mid Q$. *Assume if* $Q \equiv E'[\texttt{insync}\ o\ \{e'\}] \mid R_i$ *then* $e' \in \{E''[\texttt{ready}\ o\ n'], E''[\texttt{waiting}(c)\ n']\}$. *Then* $E[\texttt{ready}\ o\ n] \mid Q, \sigma, \texttt{CT} \longrightarrow E[\epsilon] \mid Q, \sigma, \texttt{CT}$

Proof. (1) If P_i satisfies the assumption, then we have that by Proposition 17, $E[\texttt{insync}\ o\ \{E'[e]\}]$ is only the thread which holds the monitor o. Hence progress is obvious by the definition of $\longrightarrow$. (2) is by $\mathsf{lockcount}(\sigma, o) = 0$. □

In the presence of synchronisation, there could be no progress in the program even if it is well-typed and does not reach an error state. For example, threads may deadlock by requesting monitors in a certain order, stopping them from proceeding forever: a simple example is

$$E[\texttt{insync}\ o\ \{\texttt{sync}\ (o')\ \{e\}\}] \mid E'[\texttt{insync}\ o'\ \{\texttt{sync}\ (o)\ \{e'\}\}]$$

Also waiting processes $\texttt{waiting}(c)\ n$ may not proceed forever because of a lack of "notify" (i.e. lost-wakeup). Then, as its consequence, $\texttt{ready}\ o\ n$ may never exit. We can define the states of deadlock and liveness, and prove a general progress property under a certain kind of scheduling. We leave this topic to a forthcoming exposition. We conclude this section with the normal forms theorem.

Theorem 19 (Normal forms)**.** *Assume* $N_0 \models \mathsf{Init}$ *and* $N_0 \twoheadrightarrow N \not\longrightarrow$ *and* $N \not\models \mathsf{Err}$. *Then we have* $N \equiv (\nu\, \vec{u})(\prod_{0 \le i < n} l_i[\prod_{0 \le j_i < n_i} P_{j_i}, \sigma_i, \texttt{CT}_i])$ *with* P_{j_i} *is either* $\texttt{go}\ v\ \texttt{to}\ c$ *or* $E[\texttt{insync}\ o\ \{e\}]$ *with* $e \in \{\texttt{waiting}(c)\ n, \texttt{ready}\ o\ n, \texttt{sync}\ (o')\ \{e'\}\}$.

Proof. By induction on the length of the reduction sequence leading to N. By the initial condition Init, we can set $\Delta = \vec{c}' : \vec{\texttt{chan}} \cup \vec{c_i} : \texttt{chan0}(\vec{U_i})$. The proof is direct from the progress properties. We only investigate the cases that the reduction happens across different networks. Suppose, for example, by contradiction, that $N \not\longrightarrow$ but there exists P_i such that $P_i \equiv o.m(\vec{v})\ \texttt{with}\ c \mid Q_i$. If o is an identifier for a non-remotely callable object, then $N \longrightarrow N'$ by $\mathsf{Prog}(8)$. Assume that o is an identifier to a remotely callable object and $o \notin \mathsf{dom}(\sigma_i)$. This time by MethRemote, $N \longrightarrow N'$, contradiction. Next suppose there exists P_i such that $P_i \equiv \texttt{go}\ v\ \texttt{to}\ c \mid Q_i$ with $c \in \{\vec{u}\}$ or $c : \texttt{chan} \in \Delta$. Then by $\mathsf{Prog}(10)$, there exists k such that $P_k \equiv E[\texttt{await}\ c] \mid Q_k$. Then we can apply Return, hence a contradiction. The unicity of $\texttt{go}\ v_{j_i}\ \texttt{to}\ c_{j_i}$ is derived by $\mathsf{Inv}(11)$.

□

7 Correctness of RMI Optimisations

We prove the correctness of the optimised code in § 2 by showing that the original and optimised programs have the same semantics. We do this by attempting to translate the original program into its optimised form by application of a series of sound syntactic transformation rules: if such a translation is possible, then two programs semantically equivalent.

This approach relies on the provision of a set of transformation rules that themselves preserve the program semantics. We determine that this is the case for a given transformation rule if it satisfies two conditions; it must satisfy the *noninterference property* [27,42], given in Definition 26 and; it must preserve the type of the network under transformation. Informally writing $N \mapsto N'$ for a transformation from network N to optimised network N', if the rule we applied satisfies these two conditions then both N and N' are observationally equal, hence semantically equivalent.

The technique we propose relies upon the equational laws of the linear types of mobile processes [29,54], and allows proofs of correctness to be carried out mechanically using $\mapsto$.

7.1 Observational congruence

We define an observational congruence over the syntax of DJ by applying the equational theory of process algebra [23].

Definition 20 (Congruence over networks)**.** A relation $\mathcal{R}$ over networks is a *congruence* if it is generated by the following rules.

$$\frac{N_1 \equiv N_2}{N_1 \;\mathcal{R}\; N_2} \quad \frac{N_2 \;\mathcal{R}\; N_1}{N_1 \;\mathcal{R}\; N_2} \quad \frac{N_1 \;\mathcal{R}\; N \;\; N \;\mathcal{R}\; N_2}{N_1 \;\mathcal{R}\; N_2} \quad \frac{N_1 \;\mathcal{R}\; N_2}{(\nu\, u)N_1 \;\mathcal{R}\; (\nu\, u)N_2} \quad \frac{N_1 \;\mathcal{R}\; N_2}{N_1 \,|\, N \;\mathcal{R}\; N_2 \,|\, N}$$

To formulate the behavioural equivalence over DJ, we must introduce two conditions from [23,36]. In concurrent programs, the meaning of a term that relies upon shared state can change over time, as that state is mutated by other threads. Therefore we require that if two networks are equated and one performs some computation, then the other should also be able to make computation to arrive at an equated state again. We call this a *reduction closure* property [23], and it ensures that programs that are initially equated remain equated during program execution. If this was not the case then programs that were initially thought to be equal may display different results after passage of time. The second condition that we introduce, often called "barbs" [36], is

used to describe the actions on channels that an observer of a network may witness.

Definition 21 (Reduction-closedness and the observational predicate)**.**

- A congruence $\mathcal{R}$ is *reduction closed* iff whenever $N_1 \mathcal{R} N_2$, $N_1 \rightarrow\!\!\!\!\rightarrow N_1'$ implies there exists an N_2' such that $N_2 \rightarrow\!\!\!\!\rightarrow N_2'$ with $N_1' \mathcal{R} N_2'$.
- We define the observational predicate $\downarrow_c$ and $\Downarrow_c$ as follows.

$$N \downarrow_c \text{ if } N \equiv (\nu\, \vec{u})(l[\texttt{go}\ v\ \texttt{to}\ c \,|\, P, \sigma, \texttt{CT}] \,|\, N') \qquad \text{with } c \notin \{\vec{u}\}$$
$$N \Downarrow_c \text{ if } \exists N'.(N \rightarrow\!\!\!\!\rightarrow N' \wedge N' \downarrow_c)$$

We say $\mathcal{R}$ *respects the observational predicate* if $N_1 \mathcal{R} N_2$ implies $N_1 \Downarrow_c$ iff $N_2 \Downarrow_c$, i.e. both N_1 and N_2 exhibit the same barbs.

This notion of observation is *asynchronous* because return statements have no continuation, and is chosen to match the normal forms of DJ (see Proposition 18).

Definition 22 (Sound reduction congruence)**.** A congruence is a *sound reduction congruence* if it is reduction closed and respects the observational predicate. $\cong^{\bullet}$ denotes the maximum sound reduction congruence over networks.

We note that $\cong^{\bullet}$ also includes error states (i.e. networks that satisfy the Err property given in Definition 4).

7.2 Typed observational congruence

Typed congruence relations are generated from rules similar to those for untyped congruences, however they defined relative to some typing environment. In this work, we are interested in typed congruences because types are used to restrict the behaviour of networks, contexts, and observers, and hence affect the observational semantics of programs. Moreover, our main aim is to study the behaviour of *well-typed* programs, and so insisting that any context in which we test our program is itself well-typed eliminates many "bad" observers. For example an ill-typed observer, making non-linear use of a channel, could adversely affect the program under test. This should be avoided.

Hereafter, we assume all networks started executing from an initial network satisfying Init, as given in Definition 8.

Definition 23 (Typed congruence relation between networks)**.** A relation $\mathcal{R}$ over networks is *typed* when $\Gamma_1; \Delta_1 \vdash N_1 \mathcal{R} \Gamma_2; \Delta_2 \vdash N_2$ implies $\Gamma_1 = \Gamma_2$ and $\Delta_1 = \Delta_2$. We write $\Gamma; \Delta \vdash N_1 \mathcal{R} N_2$ when $\Gamma; \Delta \vdash N_1$ and $\Gamma; \Delta \vdash N_2$ are

related by a typed relation $\mathcal{R}$. A typed congruence relation between networks is generated from the following rules:

$$\frac{N_1 \equiv N_2 \qquad \Gamma;\Delta \vdash N_i : \mathtt{net}}{\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N_2}\ (\textsc{str}) \qquad \frac{\Gamma;\Delta \vdash N_2 \;\mathcal{R}\; N_1}{\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N_2}\ (\textsc{sym})$$

$$\frac{\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N \qquad \Gamma;\Delta \vdash N \;\mathcal{R}\; N_2}{\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N_2}\ (\textsc{tra}) \qquad \frac{\Gamma, u : T;\Delta \vdash N_1 \;\mathcal{R}\; N_2}{\Gamma;\Delta \vdash (\nu\, u)N_1 \;\mathcal{R}\; (\nu\, u)N_2}\ (\textsc{resid})$$

$$\frac{\Gamma;\Delta, c : \mathtt{chan} \vdash N_1 \;\mathcal{R}\; N_2}{\Gamma;\Delta \vdash (\nu\, c)N_1 \;\mathcal{R}\; (\nu\, c)N_2}\ (\textsc{resc}) \qquad \frac{\Gamma;\Delta_1 \vdash N_1 \;\mathcal{R}\; N_2 \;\; \Gamma;\Delta_2 \vdash N : \mathtt{net} \quad \Delta_1 \asymp \Delta_2 \qquad \mathsf{loc}(N_i) \cap \mathsf{loc}(N) = \emptyset}{\Gamma;\Delta_1 \odot \Delta_2 \vdash N_1 \,|\, N \;\mathcal{R}\; N_2 \,|\, N}\ (\textsc{par})$$

By the subject reduction theorem, we immediately know $\twoheadrightarrow$ is a typed pre-congruence. Reduction closure property extends their untyped counterparts discriminating the error state. "Respect for the observational predicate" is also adjusted as follows.

Definition 24 (Typed reduction-closedness and observational predicate)**.**

- A typed congruence $\mathcal{R}$ on networks is *reduction-closed* whenever $\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N_2$, $N_1 \twoheadrightarrow N_1' \not\models \mathsf{Err}$ implies, for some N_2', $N_2 \twoheadrightarrow N_2'$ with $\Gamma;\Delta \vdash N_1' \;\mathcal{R}\; N_2'$.
- We say $\mathcal{R}$ *respects the observational predicate* if $\Gamma;\Delta \vdash N_1 \;\mathcal{R}\; N_2$ and $c : \mathtt{chan0}(T) \in \Delta$ imply $N_1 \Downarrow_c$ iff $N_2 \Downarrow_c$.

In the definition of the observational predicate above, linear channels ensure a one-one correspondence between sender and receiver (in DJ, the method caller and the thread executing the method body to return a value), and so for this reason we can assume that channels typed `chan` are participating only in the internal behaviour of the network in question. What interests us from a contextual reasoning viewpoint is those channels that are going to be used to emit information from the network under examination. Hence we require that c is typed as an output channel.

Then we say that a typed congruence is a *sound typed reduction congruence* if it is reduction closed and respects the observational predicate. Again there exists a maximum sound typed reduction congruence.

Definition 25 (Maximum sound typed reduction congruence)**.** $\cong$ denotes the maximum sound typed reduction congruence over networks $\{N \mid \exists N_0.N_0 \models \mathsf{Init}$ and $N_0 \twoheadrightarrow N$ with $N \not\models \mathsf{Err}\}$.

In this subsection we shall shortly introduce a set of transformation rules that can be used to restructure programs rigorously, and hence check the equivalence of programs. First we formally define the noninterference property.

Definition 26 (Noninterference)**.** Let us assume $\rhd$ is a typed relation closed under the rules (STR, RESI, RESC, PAR) in Definition 23, i.e. structure rules, name restrictions and parallel composition. Then we say $\rhd$ satisfies a *noninterference property* if (1) $N \longrightarrow N_1$ and $N \rhd N_2$, then either $N_1 \equiv N_2$ or there exists N' such that $N_1 \rhd N'$ and $N_2 \twoheadrightarrow N'$; (2) $N \rhd\!\!\longrightarrow N'$ implies either there exists N_0 such that $N \twoheadrightarrow N_0 \rhd N'$ or $N \twoheadrightarrow N'$.

Below we present a different and useful characterisation of "respect for the observational predicate". It is equivalent to the formulation of Definition 21 when the relation in question is reduction closed.

Definition 27 (Strong-weak observation)**.** We say a typed relation $\mathcal{R}$ respects the *strong-weak observational predicate* when, given $\Gamma;\Delta \vdash N_1 \ \mathcal{R}\ N_2$ and $c : \mathtt{chan0}(T) \in \Delta$ we have that (1) $N_1 \downarrow_c$ implies $N_2 \Downarrow_c$; and (2) $N_2 \downarrow_c$ implies $N_1 \Downarrow_c$.

Lemma 28. *Assume $\rhd$ is a typed relation closed under the rules in (*STR, RESI, RESC, PAR*) in Definition 23; and $\rhd^0 \stackrel{\text{def}}{=} \equiv$.*

(1) Suppose $\rhd \subset \twoheadrightarrow$ and $\rhd$ satisfies (1) *in Definition 26. Then $\rhd^*$ is reduction closed.*
(2) Suppose $\rhd$ satisfies a noninterference property. Then $\rhd^*$ is reduction closed.
(3) Assume that $\rhd^*$ is reduction-closed and respects the strong-weak observational predicate. Then $\rhd^*$ is sound.

Proof. (1) By taking $\mathcal{R} = \{(N, M) \mid N \rhd^* M \not\models \mathsf{Err}\}$ and showing $\mathcal{R}$ is a typed congruence and reduction-closed. First by construction of $\rhd$, $\mathcal{R}$ is a typed congruence. Next suppose $N\mathcal{R}M$ and $N \twoheadrightarrow N'$. Assume $\mathcal{R} \stackrel{\text{def}}{=} \equiv$. Then by $\equiv \subseteq \twoheadrightarrow$, we have $M \twoheadrightarrow N'$. Similarly, suppose $N\mathcal{R}M$ and $N \equiv N'$, then by $\equiv \subseteq \mathcal{R}$, we have $N'\mathcal{R}M$. Now suppose $N \rhd^+ M$ and $N \longrightarrow^+ N'$. Then by definition, there exists M' such that $M \twoheadrightarrow M'$ and $N' \rhd^+ M'$, which implies $N'\mathcal{R}M'$. Finally suppose $N\mathcal{R}M$ and $M \twoheadrightarrow M'$. By $\rhd \subset \longrightarrow$, we have that $N\mathcal{R}M \twoheadrightarrow M'$ and so $N \twoheadrightarrow N' \equiv M'$, as required. For (2), by (1), we only have to prove the case $N\mathcal{R}M$ and $M \twoheadrightarrow M'$. Then by the second condition, we have that $N \twoheadrightarrow M_0\mathcal{R}M'$ for some M_0, as desired. (3) is standard. □

Definition 29. Suppose $\rhd$ is a typed relation over networks that satisfies the

non-interference property of Definition 26. Now suppose $\blacktriangleright$ is a typed relation closed under the rules (STR, RESI, RESC, PAR) in Definition 23. Then we say that $\blacktriangleright$ satisfies a "non-interference up to transformation by $\rhd$" if, supposing $N_1 \blacktriangleright N_2$, (1) If $N_1 \longrightarrow N_1'$ then $N_1' \equiv N_2$ or there exists N_2' such that $N_2 \twoheadrightarrow N_2'$ and $N_1' \rhd^* \blacktriangleright \rhd^* N_2'$; and (2) If $N_2 \longrightarrow N_2'$ then $N_1 \equiv N_2'$ or there exists N_1' such that $N_1 \twoheadrightarrow N_1' \rhd^* \blacktriangleright \rhd^* N_2'$.

Lemma 30. *Suppose $\blacktriangleright$ satisfies a non-interference property up to transformation by $\rhd$. Define $\blacktriangleright^0 \stackrel{\text{def}}{=} \equiv$. Then $\rhd^* \cup \blacktriangleright^*$ is reduction closed. Moreover if $\blacktriangleright$ respects to the strong-weak observation predicate, then $\rhd^* \cup \blacktriangleright^*$ is sound.*

Proof. Straightforward, see [52, §6 in the full version]. □

Code that can move safely The optimisations we have introduced improve program performance by restructuring programs to eliminate communication redundancies. Not all programs in DJ can be optimised in the most obvious way (by simply bundling remote calls together and executing them at the server), because not all code can be safely relocated in the network without its meaning changing (as we discussed in § 2). To ensure that a piece of code can be safely relocated, it must satisfy the following *mobility predicate.*

Intuitively, code satisfies this predicate when it does not re-use identifiers that may have been leaked to remote sites. For instance the code $o.m(o'); o'.m'(\texttt{null})$, with o a remoteable object identifier and o' non-remoteable, would not be mobile. This is because a side-effect by method m to object o' could become visible to the subsequent call $o'.m'(\texttt{null})$ if that code was moved to the site where o is located.

In addition it does not contain the terms which takes a form of $o.f$ and $o.f = v$ with o is remoteable (since they break the locality invariants, see § 5.3.1 and § 6.1.2).

The predicate $\mathsf{Mobile}_{\Gamma,\sigma}(Q, r, s)$ states that code Q, typed in environment Γ is safe to move, provided the set of the memory locations r from store σ are moved with Q (this meaning will be clearer when the transformation rule (MOB) is introduced in the next paragraph). The set s reports the object identifiers this code potentially leaks to remote parties, which should always be disjoint with other sub-terms. For example, assume with o remoteable and o' not; then neither $o.m(o'); o'.m'(\texttt{null})$ nor $o.m(o'); o.m(o')$ are mobile because o' is potentially leaked to the remote site but is shared with other sub-terms. On the other hand, $o.m(o_1); o.m(o_2)$ *is* mobile if the objects reachable from o_1 are unreachable from o_2; then serialisation of o_1 cannot affect the contents of o_2 or any objects it points to.

The full predicate is defined inductively over the syntax of DJ, and is given in Appendix G.2. We have the following lemma that states that code that is initially safe to move remains so after reduction.

Lemma 31 (Invariance of mobility predicate)**.** *Suppose* $\mathsf{Mobile}_{\Gamma,\sigma}(P, r, s)$ *and* $P, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(P', \sigma', \mathtt{CT}')$*. Then we have one of the following:*

- $\mathsf{Mobile}_{\Gamma,\vec{u}:\vec{T},\sigma'}(P', r', s')$*; or*
- $P' \equiv E[\mathtt{await}\ c] \,|\, \mathtt{go}\ e\ \mathtt{with}\ c \,|\, P''$*; or*
- $P' \equiv E[\mathtt{await}\ c] \,|\, o.m(v)\ \mathtt{with}\ c \,|\, P''$ *with* $\Gamma \vdash o : C$ *and* $\mathsf{RMI}(C)$*; or*
- $P' \equiv \mathtt{go}\ e\ \mathtt{to}\ c \,|\, P''$*.*

Proof. See Appendix G.2.1. □

Transformation Rules We define the key transformation rules below, assuming that the right hand side is typed under $\Gamma; \Delta$. We omit surrounding contexts where they are unnecessary, and discuss each category of rule in turn.

Linearity

(L1)
$\mathtt{return}(c)\ E[\mathtt{sandbox}\ \{e_1; \ldots ; e_n\}] \mapsto e_1; \ldots ; \mathtt{return}(c)\ E[e_n]$

(L2)
$E[\mathtt{await}\ c] \,|\, e[\mathtt{return}(c)/\mathtt{return}] \mapsto E[\mathtt{sandbox}\ \{e[e'/\mathtt{return}\ e']\}]$

(L1) is standard. (L2) means that a method body e can be evaluated in-line. This is ensured by linearity of channel c.

Class

(CM)
$$\frac{\mathsf{ctcomp}(\mathtt{CT}') \qquad \vdash \mathtt{CT}' : \mathtt{ok}}{l[P, \sigma, \mathtt{CT}] \mapsto l[P, \sigma, \mathtt{CT} \cup \mathtt{CT}']}$$

(CN)
$$\frac{C \in \mathsf{dom}(\mathtt{CT})}{\begin{array}{l} l[P, \sigma, \mathtt{CT}] \mapsto l[P[C^l/C], \sigma, \mathtt{CT}] \\ l[P, \sigma, \mathtt{CT}] \mapsto l[P, \sigma[C^l/C], \mathtt{CT}] \\ l[P, \sigma, \mathtt{CT}] \mapsto l[P, \sigma, \mathtt{CT}[C^l/C]] \end{array}}$$

(CM) says that a complete class table can always be added to a location. (CN) means that we can always rename class names.

Closed

$$\frac{\{x = e\} \notin P \cup E[\,] \text{ or } x \notin \mathsf{fv}(P)}{(\nu\, x)(E[x] \mid P, \sigma \cdot [x \mapsto v]) \mapsto (\nu\, x)(E[v] \mid P, \sigma \cdot [x \mapsto v])} \text{(CR)}$$

$$\frac{P[v/x] \text{ defined}}{(\nu\, x)(P, \sigma \cdot [x \mapsto v]) \mapsto P[v/x], \sigma} \text{(SUBS)}$$

$$\frac{\neg\mathsf{reachable}(\sigma, P, o)}{(\nu\, o)(E[o.f_i] \mid P, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})]) \mapsto (\nu\, o)(E[v_i] \mid P, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})])} \text{(CF)}$$

$$\frac{\begin{array}{c}\mathsf{dom}(\sigma') \cap \{u \mid \mathsf{reachable}(\sigma, P, u)\} = \emptyset \\ \mathsf{fnv}(\sigma') \subseteq \mathsf{dom}(\sigma') \qquad \sigma' \text{ and } \mathtt{CT}' \text{ as given in premise of FREEZE.}\end{array}}{\begin{array}{c}(\nu\, \vec{u})(E[\mathtt{freeze}[t](T\ x)\{e\}] \mid P, \sigma, \mathtt{CT}) \\ \mapsto (\nu\, \vec{u})(E[\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \mathtt{CT}')] \mid P, \sigma, \mathtt{CT})\end{array}} \text{(FR)}$$

(CR) says that the timing of reading x is unimportant if x is not assigned in P or $E[\,]$; or x does not appear in P. (SUBS) means that we can assign value v to local variable x in P, provided the substitution is defined. Informally, disallowed substitutions are of the form $(x = e)[v/x]$ as this would lead to an error in execution. (CF) says that the timing of reading a field is unimportant if o is unreachable from P. (FR) means that the timing of closure creation is unimportant, provided it shares no information with other parties. Note "$\nu\, x$" in (CR), (SUBS) and "$\nu\, \vec{u}$" in (FR) ensure this lack of sharing.

Synchronisation

Suppose P and E do not include `notify`, `notifyAll`, `wait` or `ready`. Then:

$$\frac{\begin{array}{c}l[E[e] \mid P, \sigma_1, \mathtt{CT}] \longrightarrow (\nu\, \vec{u})l[E[e'] \mid P, \sigma_1', \mathtt{CT}'] \\ \sigma_2 = \mathsf{setcount}(\sigma_1, o, \mathsf{lockcount}(\sigma, o) + 1) \\ \sigma_2' = \mathsf{setcount}(\sigma_1', o, \mathsf{lockcount}(\sigma, o) + 1)\end{array}}{l[P \mid E[\mathtt{insync}\ o\ \{e\}] \mid P, \sigma_2, \mathtt{CT}] \mapsto (\nu\, \vec{u})l[E[\mathtt{insync}\ o\ \{e'\}] \mid P, \sigma_2', \mathtt{CT}']} \text{(SYNC)}$$

In the absence of monitor constructs, synchronised reduction is deterministic.

Mobile Code

(MOB)

$$\frac{\begin{array}{c}\mathsf{Mobile}_{\Gamma,\sigma_q}(Q,\vec{u},s) \qquad \{\vec{u}\} = \mathsf{dom}(\sigma_q) \\ u_i \notin \mathsf{fnv}(P,R,\sigma,\sigma_r) \qquad \{\vec{C}\} = \mathsf{icl}(Q) \cup \mathsf{icl}(\sigma_q) \qquad \mathtt{CT}'' = \mathsf{cg}(\mathtt{CT}',\vec{C})\end{array}}{\begin{array}{c}l[P,\sigma,\mathtt{CT}] \mid m[(\nu\,\vec{u})(Q \mid R,\sigma_q \cdot \sigma_r,\mathtt{CT}')] \\ \mapsto l[(\nu\,\vec{u})(P \mid Q,\sigma \cdot \sigma_q,\mathtt{CT} \cup \mathtt{CT}'')] \mid m[R,\sigma_r,\mathtt{CT}']\end{array}}$$

(AWAIT)

$$\frac{\begin{array}{c}\mathsf{Mobile}_{\Gamma,\sigma_q}(E[\,],\vec{u},s) \qquad \{\vec{u}\} = \mathsf{dom}(\sigma_q) \\ u_i \notin \mathsf{fnv}(P,R,\sigma,\sigma_r) \qquad \{\vec{C}\} = \mathsf{icl}(E[\,]) \cup \mathsf{icl}(\sigma_q) \qquad \mathtt{CT}'' = \mathsf{cg}(\mathtt{CT}',\vec{C}) \\ \mathsf{Mobile}_{\Gamma,\sigma_e}(e,\vec{u}_e,s') \qquad \{\vec{u}_e\} = \mathsf{dom}(\sigma_e) \qquad u_{ei} \notin \mathsf{fnv}(P,\sigma)\end{array}}{\begin{array}{c}l[e[\mathtt{return}(c)/\mathtt{return}] \mid P,\sigma \cdot \sigma_e,\mathtt{CT}] \mid m[(\nu\,\vec{u})(E[\mathtt{await}\ c] \mid R,\sigma_q \cdot \sigma_r,\mathtt{CT}')] \\ \mapsto l[(\nu\,\vec{u})(e[\mathtt{return}(c)/\mathtt{return}] \mid E[\mathtt{await}\ c] \mid P,\sigma \cdot \sigma_e \cdot \sigma_q,\mathtt{CT} \cup \mathtt{CT}'')] \mid m[R,\sigma_r,\mathtt{CT}']\end{array}}$$

(MOB) says that a mobile thread and accompanying store can move to a remote site safely, provided they do not share information with the originating site. The rule (AWAIT) allows the caller of a method to be co-located with the body of that method.

Deterministic Rule

(NI)
$$\frac{N \longrightarrow N'}{N \mapsto N'}$$ none of VAR, ASS, FLD, FLDASS, DOWNLOAD, FREEZE, NOTIFY, SYNC or READY applied in deriving $N \longrightarrow N'$.

This rule states that all rules that do not access or mutate the store or class table of a location are semantics preserving transformations.

Definition 32 (Transformation rules)**.** We define the transformation rule $N \mapsto N'$ as a binary relation generated by the above axioms, closed under the following parallel composition, restriction and structure rules:

$$\text{(PAR)}\ \frac{N \mapsto N'}{N \mid N_0 \mapsto N' \mid N_0} \qquad \text{(RES)}\ \frac{N \mapsto N'}{(\nu\,u)N \mapsto (\nu\,u)N'} \qquad \text{(STR)}\ \frac{N \equiv N_0 \mapsto N_0' \equiv N'}{N \mapsto N'}$$

Theorem 33.

(1) **(type preservation)** *Assume* $\Gamma;\Delta \vdash N : \mathtt{net}$ *and* $N \not\models \mathsf{Err}$*. Then* $N \mapsto N'$ *implies* $\Gamma;\Delta \vdash N' : \mathtt{net}$*.*

(2) **(noninterference)** *The binary relation* $\mapsto$ *defined in Definition 32 satisfies a noninterference property and respects the observational predicate under a network invariant.*

(3) **(semantic preservation)** $N \mapsto N'$ *implies* $N \cong N'$*.*

Proof. By rule induction of $\mapsto$. (1) We first note that $\longrightarrow$ is a typed relation and the rules except (L1), (L2),(CM),(CN),(SUBS), (MOB) and (AWAIT) are included in $\twoheadrightarrow$. Hence we only have to check these six rules. The cases (L1), (L2), (CN) and (CR) are vacuous. The case (SUBS) is proved by Substitution Lemma 12 (1). For (MOB), we need to check the invariants hold. Then the rest is mechanical. (2) It is mechanical by investigating each of the above rules in turn. First, we check the property stated in Lemma 28 (1) holds for the rules except (L1), (L2), (CM), (CN), (SUBS), (MOB) and (AWAIT), and Lemma 28 (2) hols for the rules (L1), (L2), (CM), (CN) and (SUBS). Then we check (MOB) and (AWAIT) satisfy Lemma 30. Preservation of the observational predicate is easy. (3) uses (1) and (2) together with Lemma 28 and Lemma 30. See [3] for the detailed proofs. □

Note that by Definition 4, Theorem 33 excludes the error expression and thread. This is because the transformation is *not* sound *if an error occurs during execution*, as we shall discuss in the next subsection. More formally, $N \mapsto N'$ does not always imply $N \cong^{\bullet} N'$.

Proposition 34.

(1) $\texttt{freeze}[t](T\ x)\{e\} \cong \texttt{freeze}[t'](T\ x)\{e\}$.

(2) There is a fully abstract embedding $[\![N]\!]$ *of networks* N *that contain methods* $\texttt{m}(\vec{e})$ *and frozen expressions* $\texttt{freeze}[t](\vec{T}\ \vec{x})\{e\}$ *with multiple parameters into networks with methods and frozen expressions with only single parameters.*

Proof. (1) Use Lemma 28 and (CM). (2) A translation of `freeze` is standard by currying. We encode methods with multiple parameters into those with just a single parameter in the most intuitive manner. Each method, instead of taking a vector $\vec{T}\ \vec{x}$ of parameters, takes a single parameter of a newly created class C. C contains fields $T_1\ f_1; \ldots ; T_n\ f_n$; where field f_i corresponds to the ith parameter of the original method definition. Then, all call sites for a particular method are replaced with a constructor call to an instance of the correct "parameter class", so $o.m(\vec{v})$ becomes $o.m(\texttt{new}\ C(\vec{v}))$ for some C. We then prove that $N \cong [\![N]\!]$. See Appendix H for the encoding. □

Call-backs Before proving the main theorem, we formalise the notion of call-backs between two locations.

Definition 35. For a network $N \equiv (\nu\, \vec{u})(l_1[E[\texttt{await}\ c] \mid P, \sigma, \texttt{CT}] \mid N')$, $c \rightsquigarrow c'$

denotes a chain of channels from c to c' as defined below.

$$c \rightsquigarrow c' \text{ iff } \begin{cases} & P \equiv E'[\texttt{await } c'] \mid P' \text{ and } E'[\texttt{await } c'] \text{ outputs at } c \\ \text{or} & N' \equiv l_2[E'[\texttt{await } c'] \mid P', \sigma', \texttt{CT}'] \mid N'' \\ & \qquad \text{and } E'[\texttt{await } c'] \text{ outputs at } c \\ \text{or} & \exists c''.c \rightsquigarrow c'' \rightsquigarrow c' \end{cases}$$

We say there is a *call-back from* l_2 *to* l_1 in network N if

$$N \equiv (\nu\, \vec{u})(l_1[E[\texttt{await } c] \mid P, \sigma, \texttt{CT}] \mid l_2[E'[\texttt{await } c'] \mid P', \sigma', \texttt{CT}'] \mid N')$$

and $c \rightsquigarrow c'$ and P outputs at channel c'.

This definition is understood as follows. Suppose we have made some chain of method calls, originating at site l_1 and ultimately ending up with a method being called at site l_2. If the method in site l_2 then subsequently makes another method call, this time to an object held back at l_1 then this is known as a "call-back". The equation between the third RMI program (RMI3) in Listing 5 in § 2 and the third optimal program (Opt3) in Listing 6 holds if there is no call-back as explained in § 2. Our framework can also justify the incorrectness of the optimisation between (RMI3) and (Op3) in the presence of call-back. Since most RMI programs do not use call-backs, we make no further investigation.

7.4 Proofs of correctness of RMI optimisations

We now prove the correctness of the optimised programs in § 2. We transform one program to another using the transformation rules defined above.

We first demonstrate how to transform the optimised program 1 (Opt1) in Listing 1 to the original program 1 (RMI1) in Listing 2. Let us assume e is a program from line 2 to 4 in (RMI1). We omit the surrounding context as there is no class loading in this example. After the method invocation by $o.\texttt{mOpt1}(o', n)$ `with` c,

$$(\nu\, ar)(\texttt{thunk}\langle\texttt{int}\rangle\ t = \texttt{freeze}[\texttt{lazy}]\{e; z\}; \texttt{return}(c)\ r.\texttt{run}(t), [a \mapsto n] \cdot [r \mapsto o'])$$

Let $e' = e[n/a][o'/r]$ and $v = \lambda(\texttt{unit}\ x).(l, e'; z, \emptyset, \emptyset)$. Then the above configuration is transformed to:

$$
\begin{array}{lll}
\mapsto & \texttt{thunk}\langle\texttt{int}\rangle\ t = \texttt{freeze}[\texttt{lazy}]\{e'; z\}; \texttt{return}(c)\ o'.\texttt{run}(t) & (\textsc{subs}) \\
\mapsto & (\nu\, t)(\texttt{return}(c)\ r.\texttt{run}(t), [t \mapsto v]) & (\textsc{fr}) \\
\mapsto & \texttt{return}(c)\ r.\texttt{run}(v), \emptyset & (\textsc{cr}) \\
\mapsto^{+} & (\nu\, x)(\texttt{return}(c)\ \texttt{defrost}(x), [x \mapsto v]) & (\textsc{mob}),(\textsc{ni}),(\textsc{cr}) \\
\mapsto & \texttt{return}(c)\ \texttt{defrost}(v) & (\textsc{cr}) \\
\mapsto^{+} & \texttt{return}(c)\ \texttt{sandbox}\ \{e'; z\} & (\textsc{ni}) \\
\mapsto & e'; \texttt{return}(c)\ z & (\textsc{l1})
\end{array}
$$

The last line is identical to (RMI1) after the method invocation by $o.\texttt{m1}(o', n)$ $\texttt{with}\, c$ and by (SUBS). Note that `defrost` and `sandbox` do not affect other parties, so that the reduction (NI) satisfies a noninterference property, hence this reduction preserves the semantics. Because we have $\mathsf{Mobile}_{\Gamma,\sigma}(v, \emptyset, \emptyset)$, we can apply (MOB) in the fourth line. Hence (Opt1) is transformed to (RMI1).

The correctness of (Opt2) in Listing 4 is also straightforward by repeating the same routine twice.

We show (RMI3) in Listing 5 is equivalent with (Opt3) in Listing 6 under the assumption there is no call-back. Then the body of (Opt3) is equivalent to $\texttt{return}\ r.\texttt{run}(\texttt{freeze}[\texttt{eager}](T\ x)\{e[\vec{e}'/\vec{b}]; z\})$ and $e'_i = \texttt{deserialize}(v_i)$ where $v_i = \lambda(\texttt{unit}\ x).(\nu\, \vec{u})(l, a, \sigma_i)$ is a serialised value at line i $(3 \leq i \leq 5)$ in (Opt3). We assume that σ_i does not contain remote object identifiers, hence v_i is mobile. Then we apply a similar transformation with the above to derive (RMI3). See Appendix I for the detailed proofs.

Note that our freezing preserves sharing between objects (Point 1 in (Opt3) in § 2), hence we can prove the following equation: [2]

$$x.\texttt{f} = y; r.\texttt{h}(x, y) \ \cong\ x.\texttt{f} = y; r.\texttt{run}(\texttt{freeze}\{r.\texttt{h}(x, y)\}).$$

Finally by Proposition 34 (1), we can derive (Opt4) from (Opt3), hence (Opt4) is equivalent to (RMI3). Not all equations are valid if a network error occurs during executions. For example, `eager` and `lazy` are not equal in the presence of Err-ClassNotFound, hence Proposition 34 (1) is not applicable. To summarise, we have:

Theorem 36 (Correctness of the Optimisations)**.**

[2] More precisely, in our formal syntax with the methods with single parameters, it is translated into: $x.\texttt{f} = y; C\ z = \texttt{new}\ C(x, y); r.\texttt{run}(\texttt{freeze}\{r.\texttt{h}(z)\})$ for some fresh C. One can easily check $r.\texttt{h}(z)$ is mobile, hence the correctness is proved by the same routine as above.

(1) (RMI1) *and* (Opt1) *are equivalent up to* $\cong$.
(2) (RMI2) *and* (Opt2) *are equivalent up to* $\cong$.
(3) (RMI3) *and* (Opt3) *are equivalent up to* $\cong$ *without call-back.*
(4) (Opt3) *and* (Opt4) *are equivalent up to* $\cong$, *hence* (RMI3) *and* (Opt4) *are equivalent up to* $\cong$ *without call-back.*
(5) None of them are equivalent up to $\cong^{\bullet}$.

8 Related Work

Class loading and downloading Class loading and downloading are crucial to many useful Java RMI applications, offering a convenient mechanism for distributing code to remote consumers. The class verification and maintenance of type safety during linking are studied in [31,41]. Our formulation of class downloading is modular, so it is adaptable to model other linking strategies [14,15], see § 4.2. We set the class table invariant $\mathsf{Inv}(3)$ in Definition 7. This is because the Java serialisation API imposes the strict default class version control discussed in § 6.1.1. Another solution is to explicitly model the Java exception `InvalidClassException` to check for mismatch between downloaded and existing classes. This dynamic approach leads to the same invariant to prove the subject reduction theorem.

Most of the literature surrounding class loading in practice takes the lazy approach. As we discussed earlier, in the setting of remote method invocation laziness can be expensive due to delay involved in retrieving a large class hierarchy over the network. Krintz et al [30] propose a class splitting and prefetching algorithm to reduce this. Their specific example is applet loading: if the time spent in an interactive portion of an applet is used to download classes that may be needed in future, we can fetch them ahead of time so that the user does not encounter a large delay, sharing the motivation for our (eager) code mobility primitive. The partly eager class loading in their approach is implicit, but requires control flow information about the program in question to determine where to insert instructions to trigger ahead-of-time fetching. This framework may be difficult to apply in a general distributed setting, since clients may not have access to the code of a remote server. Also their approach merely mitigates the effect of network delay rather than removing it; it still requires the sequential request of a hierarchy of superclasses. We believe an explicit thunk primitive as we proposed in the present work may offer an effective alternative in such situations.

Distributed objects Obliq [12] is a distributed, object-based, and lexically scoped language proposed by Cardelli. One key feature of the language is that methods are stored within objects—there is no hierarchy of tables to inspect

as in most class-based languages. Merro et al [33] encode a core part of Obliq into the untyped π-calculus. They use their encoding to show a flaw in part of the original migration semantics and propose a repair. Later Nestmann et al [37] formalised a typing system for a core Obliq calculus and studied different kinds of object aliasing. Briais and Nestmann [11] then strengthened the safety result in [33] by directly developing the must equivalence at the language level (without using the translation into the π-calculus). They also apply a noninterference property to show the two terms (with and without surrogation) are must-equivalent. DJ models two important concerns in distributed class-based object-oriented languages missing from Obliq, that is object serialisation and dynamic class downloading associated with inheritance in Java (note that the same term "serialisation" used in [12] is used in the context of transaction theory). These features require a consistent formulation of dynamic deep copying of object/class graphs. As we have seen in § 7, detailed analysis of these features is required to justify the correctness of the optimisation examples in § 2. The proof method using syntactic transformations in § 7 is also new.

Emerald [24] is another example of a distributed object-based language. It supports classes represented as objects, however there is no concept of class loading as in DJ—information about inheritance hierarchies is discarded at compile-time. Objects in Emerald may be *active* in that they are permitted their own internal thread of control that runs concurrently with method invocations on that object. Such objects may explicitly move themselves to other locations by making a library call. In DJ the fundamental unit of mobility is arbitrary higher-order expressions: this general code freezing primitive can represent object mobility similar to Emerald when it is combined with standard Java RMI. Finally, there has been no study of the formal semantics of Emerald.

Gordon and Hankin [17] extend the object calculus [1] with explicit concurrency primitives from the π-calculus. Their focus is synchronisation primitives (such as fork and join) rather than distribution, so they only use a single location. Jeffrey [26] treats an extension of [17] for the study of locality with static and dynamic type checking. The concurrent object calculus is not class-based, hence neither work treats dynamic class loading or serialisation (though [26] treats transactional serialisation as in [12]), which are among the key elements for analysis of RMI and code mobility in Java.

Scope and runtime formalisms for Java Zhao et al [56] propose a calculus with primitives for explicit memory management, called SJ, for a study of containment in real-time Java. The SJ calculus proposes a typing discipline based on the idea of *scoped types*—memory in real-time applications is allocated in a strict hierarchy of scopes. Using the existing Java package structure to divide such scopes, their typing system statically prevents some scope

invariants being broken. Their focus is on real-time concurrency in a single location, while ours is on dynamic distribution of code in multiple locations. DJ also guarantees similar scoping properties by invariants, for example $\mathsf{Inv}(6)$ in Definition 7 ensures that identifiers for non-remotely callable objects do not leak to other locations in the presence of synchronisation primitives.

The representation of object-oriented runtime in formal semantics is not limited to distributed programs, as found in study of execution models of the .NET CLR by Gordon and Syme [18] and Yu et al [55].

The JavaSeal [48] project is an implementation of the Seal calculus for Java. It is realised as an API and run-time system inside the JVM, targeted as a programming framework for building multi-agent systems. The semantics of these APIs depend on distributed primitives in the implementation language, which are precisely the target of the formal analysis in the present paper. JavaSeal may offer a suggestion for the implementation and security treatment of higher-order code passing proposed in the present paper.

Functions with marshaling primitives Ohori and Kato [38] extend a purely functional part of ML with two primitives for remote higher-order code evaluation via channels, and show that the type system of this language is sound with respect to a low-level calculus. The low-level calculus is equipped with runtime primitives such as closures of functions and creation of names. Their focus is pure polymorphic functions, hence they treat neither side-effects nor (distributed) object-oriented features. Acute [2] is an extension of OCaml equipped with type-safe marshaling and distributed primitives. By using flags called marks, the user can control dynamic loading of a sequence of modules when marshaling his code. This facility is similar to our lazy and eager class loading. The language also provides more flexible way to rebind local resources and modules. An extension of our freeze operator for fine-grained rebinding is an interesting topic, though as we discussed in § 6.1.1, it is not suitable in practice due to the Java serialisation API.

Staged computation and meta-programming Taha and Sheard [45] give a dialect of ML containing staging annotations to generate code at runtime, and to control evaluation order of programs. The authors give a formal semantics of their language, called MetaML, and prove that the code a well-typed program generates will itself be type-safe.

The `freeze` and `defrost` primitives in DJ can be thought of as staging annotations, and also guarantee that frozen expressions should be well-typed in any context. However we study distribution and concurrency in an imperative

setting, with strong emphasis on runtime features. These features are not discussed in MetaML as it is a functional language, nor the problems associated with classloading we address.

Kamin et al [28] extend the syntax of Java with staging annotations and provide a compiler for a language called Jumbo. They allow creation of classes at runtime, focusing on single-location performance optimisation: there is no discussion of use in distributed applications, a main focal point of our work. They give no static guarantees about type safety of generated code, nor do they allow code to be generated in fragments smaller than an entire class. They do not consider higher-order quotation, permitting only one level of quotation and anti-quotation.

Zook et al [57] propose Meta-AspectJ as a meta-programming tool for an aspect-oriented language. They implement a compiler that takes code templates, containing quoted Aspect-J code, and turns them into aspect declarations that can be applied as normal to Java programs. Their system is more focused on compile-time code generation, and offers weaker static guarantees: well-typed generators do not guarantee type safety of the generated aspects.

9 Conclusions and Further Work

This paper introduced a Java-like core language for RMI with higher-order code mobility. It models the runtime behaviour of distributed computation including dynamic class downloading and object serialisation. Using the new primitives for code mobility, we subsumed the existing serialisation mechanism of Java and were able to precisely describe examples of communication-based optimisations for RMI programs on a formal foundation. We established type preservation and safety properties of the language using distributed invariants. Finally, by the behavioural theory developed in § 7, we were able to systematically prove the correctness of the examples in § 2.

Explicit code mobility as a language primitive gives powerful control over code distribution strategies in object-oriented distributed applications. This is demonstrated in the examples in § 2. In [10,51,50], these optimisations are informally described as implementation details. Not only is source-level presentation necessary for their semantic justification, but also explicit treatment of code mobility gives programmers fine-grained control over the evaluation order and location of executing code. It also opens the potential for source-level verification methodologies for access control, secrecy and other security concerns, as briefly discussed below. Note current customised class downloading mechanisms do not offer active code mobility and algorithmic control of code distribution (as in the last example of § 2).

Further, the fine-grained control of code mobility has a direct practical significance: the optimisation strategy in [51,50] cannot aggregate code in which new object generation is inserted, such as:

```
int m3(RemoteObject r, MyObj a) {
  int x = r.f(a);
  int y = r.g(new MyObj(x));
  int z = r.h(a, y);
  return z;
}
```

where `MyObj` is a non-remoteable class in the client. This is because we need active class code delivery if this code is to be executed in a remote server. In contrast, the freeze primitive in our language can straightforwardly handle aggregation of this code. We also believe that, in comparison with direct, byte-code level implementation in [51,50], the use of our high-level primitives may not jeopardise efficiency but rather can even enhance it by e.g. allowing more flexible inter-procedure optimisation.

The complexity of the third program optimisation poses the question of whether the original copying semantics of Java RMI are themselves correct in the first place: making a remote call can entail subtly different invocation semantics to calling a local method. Our code freezing primitive allows us to make the call semantics explicit, and also allows us to support more traditional ideas about object mobility [24,12], such as side-effects in calls at the server side.

The class-based language considered in the present work does not include such language features as casting [25,9], exceptions [6] and parametric polymorphism [25]; although these features can be represented by extension of the present syntax and types, their precise interplay with distributed language constructs requires examination. Further the first author's forthcoming PhD thesis [3] gives a more detailed analysis of serialisation mechanism, allowing interference during calculating object graphs.

An important future topic is enrichment of the invariants and type structures to strengthen safety properties (e.g. for security). Here we identify two orthogonal directions. The first concerns mobility. As can be seen in the second example in § 2, the current type structure of higher-order code (e.g. `thunk<int>`) tells the consumer little about the behaviour of the code he is about to execute, which can be dangerous [32,10]. In Java, the `RMISecurityManager` can be used with an appropriate policy file to ensure that code downloaded from remote sites has restricted capability. By extending DJ with principals, we can examine the originator of a piece of code to determine suitable privileges prior to execution [49]. To ensure the integrity of resources we can dynamically check invariants when code arrives (e.g. by adding constraints

in DEFROST), or we could allow static checking by adding more fine-grained information about the accessibility of methods in class signatures, along the lines of [53].

The second direction is to extend the syntax and operational semantics to allow complex, structured, communications. For this purpose we have been studying *session types* [22,47] for ensuring correct pattern matching of sequences of socket communications, incorporating a new class of channels at the user syntax level. Our operational semantics for RMI is smoothly extensible to model advanced communication protocols. Session types are designed using class signatures, and safety is proved together with the same invariance properties developed in this paper.

Study of the semantics of failure and recovery in our framework is an important topic. So far we have incorporated the possibility of failures in class downloading and remote invocation due to network partition (defined by **Err**-rules in § 4). When a message is lost, some notion of time-out is generally used to determine whether to re-transmit or fail. Such error recovery can be investigated by defining different invocation semantics (for example at-most-once [34]) and adding runtime extensions to DJ. This point is also relevant when we consider socket-based communication instead of RMI.

We have implemented an initial version of our new primitives for code mobility [46,39]. This takes the form of a source-to-source translator, compiling the `freeze` and `defrost` operations into standard Java source. Eager class loading via RMI requires modification to the class loading mechanism, which is achieved by installing a custom class loader working in conjunction with our translated source. This approach has the advantage that we can use an ordinary Java compiler and existing tools, and that the JVM would not need modification. However a more direct approach (for example extending the virtual machine) may yield better performance.

The examples in § 2 and the transformation rules in § 7 lead to the question of how to automatically translate from RMI source programs to programs exploiting code mobility for added efficiency. Developing a general theory and an integrated tool is non-trivial due to an interplay between inter node and procedure optimisations. Furthermore we need to formalise a cost theory for distributed communication with respect to the distance of the locations and the size of code and class tables transferred. DJ can be used as a reference model to define efficiency since it exposes distributed runtime explicitly by means of syntax and reduction rules. For example, we can add marshaling costs to the FREEZE rule with respect to the size of the frozen expression; we can investigate the cost of class downloading with respect to the size of a downloaded class table $\mathtt{CT}'$ and a distance between location l_1 and location l_2, using rule DOWNLOAD. An interesting further topic is an application to

DJ of the cost-preorder theory developed for process algebra [8] to compare program performance.

Acknowledgements We deeply thank Paul Kelly for his discussions about the Veneer vJVM and the RMI optimisations in [51,50]. We are very grateful to the anonymous reviewers who provided us with detailed comments and suggestions to improve the presentation of the paper. We thank Luca Cardelli, Susan Eisenbach, Kohei Honda, Andrew Kennedy and members of the SLURP and ToCS groups at Imperial College London for their discussions, Farzana Tejani and Karen Osmond for their work on implementation. We thank Mariangiola Dezani, Sophia Drossopoulou and Uwe Nestmann for their comments on an earlier version of this paper. The first author is partially supported by an EPSRC PhD studentship and the second author is partially supported by EPSRC Advanced Fellowship (GR/T03208/01), EPSRC GR/S55538/01, EPSRC GR/T04724/01 and EU IST-2005-015905 MOBIUS project.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Acute. Acute home page. `http://www.cl.cam.ac.uk/users/pes20/acute`, 2005.

[3] Alexander Ahern. *Code Mobility and Java RMI*. PhD thesis, Deparment of Computing, Imperial College London, To appear.

[4] Alexander Ahern and Nobuko Yoshida. Formal Analysis of a Distributed Object-Oriented Language and Runtime. Technical Report 2005/01, Department of Computing, Imperial College London: Available at: `http://www.doc.ic.ac.uk/~aja/dcbl.html`, 2005.

[5] Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with Explicit Code Mobility. In *OOPSLA '05, the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230. ACM Press, 2005.

[6] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Simplifying types in a calculus for Java exceptions. Technical report, DISI - Università di Genova, 2002.

[7] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[8] S. Arun-Kumar and Matthew Hennessy. An efficiency preorder for processes. *Acta Inf.*, 29(8):737–760, 1992.

[9] Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.

[10] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *OOPSLA '94*, pages 341–354. ACM Press, 1994.

[11] Sébastien Briais and Uwe Nestmann. Mobile objects "must" move safely. In *FMOODS 2002*, pages 129–146. Kluwer Academic Publishers, 2002.

[12] Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Systems Research Center, Digital Equipment Corporation, 1994.

[13] R. Christ. SanFrancisco Performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1), 2000.

[14] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Dynamic Linking. In *The First Workshop on Unanticipated Software Evolution (USE 2002)*, Málaga, Spain, June 2002. `http://joint.org/use2002/proceedings.html`.

[15] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *12th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, April 2003.

[16] David Flanagan. *Java Examples in a Nutshell*. O'Reilly UK, 2000.

[17] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. Technical Report 457, University of Cambridge Computer Laboratory, February 1999.

[18] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 248–260. ACM Press, 2001.

[19] Todd Greanier. Discover the secrets of the Java Serialization API. `http://java.sun.com/developer/technicalArticles/Programming/serialization/`, 2005.

[20] Kohei Honda. Composing processes. In *Proceedings of POPL'96*, pages 344–357, 1996.

[21] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147, 1991.

[22] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138, 1998.

[23] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *TCS*, 151(2):385–435, 1995.

[24] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programming Language. Technical report, Department of Computer Science, University of British Columbia, Vancouver BC, Canada V6T 1Z2, October 1991.

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

[26] Alan Jeffrey. A Distributed Object Calculus. In *FOOL*. ACM Press, 2000.

[27] Cliff Jones. Constraining interference in an object-based design method. In *Proceedings of TAPSOFT'93*, volume 668 of *Lecture Notes in Computer Science*, pages 136–150. Springer Verlag, 1993.

[28] Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for Java and its applications. In *CGO03*. IEEE, 2003.

[29] Naoki Kobayashi, Benjamin Pierce, and David Turner. Linear types and π-calculus. In *Proceedings of POPL'96*, pages 358–371, 1996.

[30] Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 276–291. ACM Press, 1999.

[31] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44. ACM Press, 1998.

[32] Gary McGraw and Greg Morrisett. Attacking malicious code: a report to the infosec research council. *IEEE Software*, 17(5):33–44, 2000.

[33] Massimo Merro, Josva Kleist, and Uwe Nestmann. Mobile objects as mobile processes. *Information and Computation*, 177(2):195–241, 2002.

[34] Sun Microsystems Inc. Java Remote Method Invocation (RMI). `http://java.sun.com/products/jdk/rmi/`, 2005.

[35] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.

[36] Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. ICALP'92*, volume 623 of *LNCS*, pages 685–695. Springer Verlag, 1992.

[37] Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. Aliasing models for mobile objects. *Inf. Comput.*, 177(2):195–241, 2002.

[38] Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–112. ACM Press, 1993.

[39] Karen Osmond, Alexander Ahern, and Nobuko Yoshida. DJ SourceForge Homepage. `http://dj-project.sourceforge.net/`.

[40] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[41] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–336. ACM Press, 2000.

[42] John C. Reynolds. Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 39–46. ACM Press, 1978.

[43] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[44] James W. Stamos and David K. Gifford. Implementing remote evaluation. *IEEE Trans. Softw. Eng.*, 16(7):710–722, 1990.

[45] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217. ACM Press, 1997.

[46] Farzana Tejani. Implementation of a distributed mobile Java. Master's thesis, Imperial College London, MEng. Computing Final Year Project, Imperial College London, 2005.

[47] Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511, 2004.

[48] Jan Vitek, Ciarán Bryce, and Walter Binder. Designing JavaSeal or how to make Java safe for agents. *Electronic Commerce Objects*, 1998.

[49] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 116–128. ACM Press, 1997.

[50] Kwok Yeung. *Dynamic performance optimisation of distributed Java applications*. PhD thesis, Imperial College London, 2004.

[51] Kwok Yeung and Paul Kelly. Optimizing Java RMI programs by communication restructuring. In *Middleware'03*, volume 2672 of *Lecture Notes in Computer Science*, pages 324–343, 2003.

[52] Nobuko Yoshida. Graph types for monadic mobile processes. In *FSTTCS'16*, volume 1180 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 1996. Full version as Edinburgh LFCS Technical Report,ECS-LFCS-96-350, 1996.

[53] Nobuko Yoshida. Channel dependency types for higher-order mobile processes. In *POPL '04, Conference Record of the 31st Annual Symposium on Principles of Programming Languages*, pages 147–160. ACM Press, 2004. Full version available at `www.doc.ic.ac.uk/~yoshida`.

[54] Nobuko Yoshida, Martin Berger, and Kohei Honda. Strong Normalisation in the π-Calculus. In *Proc. LICS'01*, pages 311–322. IEEE, 2001. The full version in *Journal of Inf. & Comp.*, 191 (2004) 145–202, Elsevier.

[55] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .net common language runtime. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 39–51. ACM Press, 2004.

[56] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time Java. In *Proceedings of the 25th Annual IEEE Symposium on Real-time Systems*, 2004.

[57] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag GmbH, 2004.

A Auxiliary definitions

This appendix contains the full definitions of some of the functions used in the main sections.

A.1 Domains

The function dom returns the domain of a mapping. It is defined over stores, class tables, class signatures and configurations, and is given as follows.

$$\mathsf{dom}(\emptyset) = \emptyset,$$

$$\mathsf{dom}(\sigma \cdot [x \mapsto \ldots]) = \mathsf{dom}(\sigma) \cup \{x\}, \quad \mathsf{dom}(\sigma \cdot [o \mapsto \ldots]) = \mathsf{dom}(\sigma) \cup \{o\}$$

$$\mathsf{dom}(\mathtt{CT} \cdot [C \mapsto \ldots]) = \mathsf{dom}(\mathtt{CT}) \cup \{C\}$$

$$\mathsf{dom}(\mathsf{CSig} \cdot [C \mapsto \ldots]) = \mathsf{dom}(\mathsf{CSig}) \cup \{C\}$$

$$\mathsf{dom}((\nu\, \vec{u})(P, \sigma, \mathtt{CT})) = \mathsf{dom}(\sigma) \setminus \{\vec{u}\}$$

$$\mathsf{dom}(\mathbf{0}) = \emptyset \qquad \mathsf{dom}(l[F]) = \mathsf{dom}(F)$$

$$\mathsf{dom}(N_1 \mid N_2) = \mathsf{dom}(N_1) \cup \mathsf{dom}(N_2) \qquad \mathsf{dom}((\nu\, u)N) = \mathsf{dom}(N) \setminus \{u\}$$

The functions for determining free variables fv and free names fn are defined as follows. For classes and methods:

$$\mathsf{fv}(\texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}) = \bigcup \mathsf{fv}(M_i)$$
$$\mathsf{fv}(U\ m(T\ x)\{e\}) = \mathsf{fv}(e) \setminus \{x\}$$

$$\mathsf{fn}(\texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}) = \bigcup \mathsf{fn}(M_i)$$
$$\mathsf{fn}(U\ m(T\ x)\{e\}) = \mathsf{fn}(e)$$

For values:

$$\mathsf{fv}(\texttt{true}) = \mathsf{fv}(\texttt{false}) = \mathsf{fv}(()) = \mathsf{fv}(\texttt{null}) = \mathsf{fv}(\epsilon) = \mathsf{fv}(o) = \emptyset$$
$$\mathsf{fv}(\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})) = ((\mathsf{fv}(e) \setminus \{x\}) \cup\ \mathsf{fv}(\sigma) \cup \mathsf{fv}(\texttt{CT})) \setminus \{\vec{u}\}$$

$$\mathsf{fn}(\texttt{true}) = \mathsf{fn}(\texttt{false}) = \mathsf{fn}(()) = \mathsf{fn}(\texttt{null}) = \mathsf{fn}(\epsilon) = \emptyset$$
$$\mathsf{fn}(o) = \{o\}$$
$$\mathsf{fn}(\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})) = (\mathsf{fn}(e) \cup\ \mathsf{fn}(\sigma) \cup \mathsf{fn}(\texttt{CT})) \setminus \{\vec{u}\}$$

For expressions we omit the cases where the free variables (resp. names) of a term are merely the union of the free variables of its subterms.

$$\mathsf{fv}(x) = \{x\} \qquad \mathsf{fn}(x) = \emptyset$$
$$\mathsf{fv}(\texttt{this}) = \emptyset \qquad \mathsf{fn}(\texttt{this}) = \emptyset$$
$$\mathsf{fv}(x = e) = \{x\} \cup \mathsf{fv}(e) \qquad \mathsf{fn}(x = e) = \mathsf{fn}(e)$$
$$\mathsf{fv}(T\ x = e_0; e_1) = \mathsf{fv}(e_0) \cup (\mathsf{fv}(e_1) \setminus \{x\}) \qquad \mathsf{fn}(T\ x = e_0; e_1) = \bigcup \mathsf{fn}(e_i)$$
$$\mathsf{fv}(\texttt{return}) = \emptyset \qquad \mathsf{fn}(\texttt{return}) = \emptyset$$
$$\mathsf{fv}(\texttt{freeze}[t](T\ x)\{e\}) = \mathsf{fv}(e) \setminus \{x\} \qquad \mathsf{fn}(\texttt{freeze}[t](T\ x)\{e\}) = \mathsf{fn}(e)$$
$$\mathsf{fv}(\texttt{await}\ c) = \emptyset \qquad \mathsf{fn}(\texttt{await}\ c) = \{c\}$$
$$\mathsf{fv}(\texttt{insync}\ o\ \{e\}) = \mathsf{fv}(e) \qquad \mathsf{fn}(\texttt{insync}\ o\ \{e\}) = \{o\} \cup \mathsf{fn}(e)$$
$$\mathsf{fv}(\texttt{ready}\ o\ n) = \emptyset \qquad \mathsf{fn}(\texttt{ready}\ o\ n) = \{o\}$$
$$\mathsf{fv}(\texttt{waiting}(c)\ n) = \emptyset \qquad \mathsf{fn}(\texttt{waiting}(c)\ n) = \{c\}$$
$$\mathsf{fv}(\texttt{Error}) = \emptyset \qquad \mathsf{fn}(\texttt{Error}) = \emptyset$$

For threads:

$$\mathsf{fv}(\mathbf{0}) = \emptyset \qquad \mathsf{fn}(\mathbf{0}) = \emptyset$$
$$\mathsf{fv}(P_1 \mid P_2) = \bigcup \mathsf{fv}(P_i) \qquad \mathsf{fn}(P_1 \mid P_2) = \bigcup \mathsf{fn}(P_i)$$
$$\mathsf{fv}((\nu\,u)P) = \mathsf{fv}(P) \setminus \{u\} \qquad \mathsf{fn}((\nu\,u)P) = \mathsf{fn}(P) \setminus \{u\}$$
$$\mathsf{fv}(\texttt{forked}\ e) = \mathsf{fv}(e) \qquad \mathsf{fn}(\texttt{forked}\ e) = \mathsf{fn}(e)$$
$$\mathsf{fv}([\mathsf{go}]\ e\ \texttt{with/to}\ c) = \mathsf{fv}(e) \qquad \mathsf{fn}([\mathsf{go}]\ e\ \texttt{with/to}\ c) = \{c\} \cup \mathsf{fn}(e)$$
$$\mathsf{fv}(\texttt{return}(c)\ e) = \mathsf{fv}(e) \qquad \mathsf{fn}(\texttt{return}(c)\ e) = \{c\} \cup \mathsf{fn}(e)$$

For configurations, stores and class tables:

$$\begin{aligned}
\mathsf{fv}((\nu\,\vec{u})(P,\sigma,\mathtt{CT})) &= (\mathsf{fv}(P) \cup \mathsf{fv}(\sigma) \cup \mathsf{fv}(\mathtt{CT})) \setminus \{\vec{u}\} \\
\mathsf{fn}((\nu\,\vec{u})(P,\sigma,\mathtt{CT})) &= (\mathsf{fn}(P) \cup \mathsf{fn}(\sigma) \cup \mathsf{fn}(\mathtt{CT})) \setminus \{\vec{u}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{fv}(\emptyset) &= \emptyset \\
\mathsf{fn}(\emptyset) &= \emptyset \\
\mathsf{fv}(\sigma \cdot [x \mapsto v]) &= \{x\} \cup \mathsf{fv}(v) \cup \mathsf{fv}(\sigma) \\
\mathsf{fn}(\sigma \cdot [x \mapsto v]) &= \mathsf{fn}(v) \cup \mathsf{fn}(\sigma) \\
\mathsf{fv}(\sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})) &= \mathsf{fv}(\vec{v}) \cup \mathsf{fv}(\sigma) \\
\mathsf{fn}(\sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})) &= \{o, \vec{c}\} \cup \mathsf{fn}(\vec{v}) \cup \mathsf{fn}(\sigma)
\end{aligned}$$

$$\begin{aligned}
\mathsf{fv}(\emptyset) &= \emptyset \\
\mathsf{fn}(\emptyset) &= \emptyset \\
\mathsf{fv}(\mathtt{CT} \cdot [C \mapsto L]) &= \mathsf{fv}(L) \cup \mathsf{fv}(\mathtt{CT}) \\
\mathsf{fn}(\mathtt{CT} \cdot [C \mapsto L]) &= \mathsf{fn}(L) \cup \mathsf{fn}(\mathtt{CT})
\end{aligned}$$

For networks:

$$\begin{aligned}
\mathsf{fv}(\mathbf{0}) &= \emptyset & \mathsf{fn}(\mathbf{0}) &= \emptyset \\
\mathsf{fv}(l[F]) &= \mathsf{fv}(F) & \mathsf{fn}(l[F]) &= \mathsf{fn}(F) \\
\mathsf{fv}(N_1 \mid N_2) &= \bigcup \mathsf{fv}(N_i) & \mathsf{fn}(N_1 \mid N_2) &= \bigcup \mathsf{fn}(N_i) \\
(\nu\,u)N &= \mathsf{fv}(N) \setminus \{u\} & (\nu\,u)N &= \mathsf{fn}(N) \setminus \{u\}
\end{aligned}$$

B Structural equivalence

This section defines the structural equivalences for DJ. They are defined for threads, networks and configurations in Figure B.1. Formally, $\equiv$ is an equivalence relation which includes α-conversion and is generated by the equations in Figure B.1.

The last two rules for configurations define garbage collection of useless store entries, while the last three rules for threads are used to erase runtime value ϵ of the void type. Others rules, including scope opening, are inherited from those of the π-calculus [35], and so are standard.

C Proofs for the Correctness of the Algorithms

This section lists the proofs for Lemma 11.

(Configurations)

$$
\begin{aligned}
(\nu\, u)P, \sigma, \mathtt{CT} &\equiv (\nu\, u)(P, \sigma, \mathtt{CT}) && u \notin \mathsf{fn}(\sigma) \cup \mathsf{fn}(\mathtt{CT}) \\
(\nu\, u)(\nu\, u')F &\equiv (\nu\, u')(\nu\, u)F \\
(\nu\, x)(P, \sigma \cdot [x \mapsto v], \mathtt{CT}) &\equiv P, \sigma, \mathtt{CT} && x \notin \mathsf{fv}(P) \\
(\nu\, o)(P, \sigma \cdot [o \mapsto (C, \ldots)], \mathtt{CT}) &\equiv P, \sigma, \mathtt{CT} && o \notin \mathsf{fn}(P) \cup \mathsf{fn}(\sigma)
\end{aligned}
$$

(Threads)

$$
\begin{aligned}
P \mid \mathbf{0} &\equiv P \\
P \mid P_0 &\equiv P_0 \mid P \\
P \mid (P_0 \mid P_1) &\equiv (P \mid P_0) \mid P_1 \\
(\nu\, u)(P \mid P_0) &\equiv (\nu\, u)P \mid P_0 \quad u \notin \mathsf{fn}(P_0) \\
(\nu\, c)\mathbf{0} &\equiv \mathbf{0} \\
(\nu\, u)(\nu\, u')P &\equiv (\nu\, u')(\nu\, u)P \\
\mathtt{return}(d)\ \epsilon &\equiv \mathtt{return}(d) \\
v; e &\equiv e \\
\mathtt{return}\ \epsilon &\equiv \mathtt{return}
\end{aligned}
$$

(Networks)

$$
\begin{aligned}
N \mid \mathbf{0} &\equiv N \\
N \mid N_0 &\equiv N_0 \mid N \\
N \mid (N_0 \mid N_1) &\equiv (N \mid N_0) \mid N_1 \\
(\nu\, u)(N \mid N_0) &\equiv (\nu\, u)N \mid N_0 \quad u \notin \mathsf{fnv}(N_0) \\
(\nu\, c)\mathbf{0} &\equiv \mathbf{0} \\
(\nu\, u)(\nu\, u')N &\equiv (\nu\, u')(\nu\, u)N \\
l[(\nu\, u)(F)] &\equiv (\nu\, u)l[F]
\end{aligned}
$$

Fig. B.1. Structural equivalence

C.1 Proof of Lemma 11 (1)

(a) Immediate by the definition of og.

(b) Proof is by induction on the length of store σ in both the "if" and "only-if" directions. For the base case of both of these directions, if we assume $\sigma = \emptyset$ (i.e. has zero length) then clearly nothing is reachable. Then by examination of og we see the generated object graph will also be $\emptyset$, meaning no identifiers can be reached from o. This completes the base case.

"If" direction For the inductive step, assume that

$$\mathsf{reachable}(\sigma, o, o') \text{ implies } \mathsf{reachable}(\mathsf{og}(\sigma, o), o, o')$$
$$\mathsf{reachable}(\sigma \cdot [o'' \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})], o, o')$$

We shall prove

$$\mathsf{reachable}(\mathsf{og}(\sigma \cdot [o'' \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})], o), o, o')$$

Fix o', now the interesting case arises when

$$\mathsf{reachable}(\sigma \cdot [o'' \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})], o, o') \text{ but } \neg\mathsf{reachable}(\sigma, o, o')$$

By assumption, $\mathsf{reachable}(\sigma, o, o'')$ hence by applying the inductive hypothesis $\mathsf{reachable}(\mathsf{og}(\sigma, o), o, o'')$. Examining how the object graph func-

tion is defined, we know that for some store $\sigma_x \subseteq \sigma$ we applied $\mathsf{og}(\sigma_x, o'')$. Now since the mapping for o'' is not in σ, this application would have no mapping to copy.

However in the store $\sigma \cdot [o'' \mapsto (C], \vec{f} : \vec{v}, n, \{\vec{c}\})$ above, this application would succeed and copy the o'' mapping. Then we have two cases to consider:

$$o' \in \mathsf{fn}(\vec{v}) \tag{a}$$
$$\text{or there exists } o_i \in \mathsf{fn}(\vec{v}) \text{ and } \mathsf{reachable}(\sigma, o_i, o') \tag{b}$$

For (a), we immediately have that because the object mapping of o'' is now copied, $\mathsf{reachable}(\mathsf{og}(\sigma \cdot [o'' \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})], o), o, o')$. For (b) by examination of the object graph algorithm we see each field of o'' is explored by recursively calling og using a store that is strictly smaller than $\sigma \cdot [o'' \mapsto \ldots]$ (because the mapping for o'' is explicitly removed). Hence by the inductive hypothesis o' is reachable in the object graph computed from $o_i \in \mathsf{fn}(\vec{v})$. As these are combined with the o'' mapping to produce the object graph for o'' we have $\mathsf{reachable}(\mathsf{og}(\sigma \cdot [o'' \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})], o), o, o')$ as required.

"Only-if" direction Immediate by (c) below. Note that reachability is determined entirely by the field structure of objects, and as the algorithm preserves field identities when a mapping is copied, the structure of the computed object graph reflects that of a portion of the larger store.

(c) Straightforward noting that the object graph algorithm copies store mappings setting the lock count to zero and the waiting set to empty.

(d) Obvious noting that the queue of waiting threads in each copied object is set to $\emptyset$, allowing us to strengthen environment Δ successively.

C.2 Proof of Lemma 11 (2)

By induction on length($\mathtt{CT}$).

C.3 Proof of Lemma 11 (3)

Suppose length($\mathtt{CT}_0$) = 0 then by definition, $\mathtt{CT}'$ is complete. Now, assume length($\mathtt{CT}_n$) = n. Given $\mathtt{CT}' = \mathsf{cg}(\mathtt{CT}_n, C)$ for some C is complete by assumption, we either have that $C \in \mathsf{dom}(\mathtt{CT}_n)$ and $\mathtt{CT}' \neq \emptyset$, or $C \notin \mathsf{dom}(\mathtt{CT}_n)$ and $\mathtt{CT}' = \emptyset$. For the inductive step we must show that when the length of the class table is $n+1$ the computed class graph remains complete. Extending the class table can be achieved by appending a new entry giving $\mathtt{CT}_{n+1} = \mathtt{CT}_n \cdot [C' \mapsto L]$

for some $C' \notin \mathsf{dom}(\mathtt{CT}_n)$. We assume that the superclass of C' is present in $\mathtt{CT}_n$, otherwise the new class table would not be complete and so the conclusion would hold by default. Then given $\mathtt{CT}' = \mathsf{cg}(\mathtt{CT}_{n+1}, C)$, if $C \neq C'$ then again $\mathtt{CT}'$ is complete by virtue of being empty. If $C = C'$ then by our assumption that the class table $\mathtt{CT}_{n+1}$ contains the direct superclass of C' then $\mathtt{CT}'$ must also be complete.

D Basic Properties

In this Appendix we shall show some key properties and lemmas that are necessary for the proof of our network invariance and type preservation theorem. Hereafter we often write α for U or S. We also adopt the convention that $\Gamma; \emptyset$ can be written as simply Γ.

D.1 Judgements

Lemma 37 lists some useful properties about judgements. We write $\mathtt{J}$ to stand for any one of the following judgements:

$$\mathtt{J} ::= \mathtt{Env} \mid \sigma : \mathtt{ok} \mid e : \alpha \mid P : \mathtt{thread} \mid F : \mathtt{conf} \mid N : \mathtt{net}$$

This lemma also has the useful property of ensuring that any channels appearing in the channel environment Δ and not in the judgement $\mathtt{J}$ must have the linear type `chan`.

Lemma 37 (Judgements)**.**

(1) $\Gamma; \Delta, c : \tau, c' : \tau', \Delta' \vdash \mathtt{J} \implies \Gamma; \Delta, c' : \tau', c : \tau, \Delta' \vdash \mathtt{J}$.
(2) $\Gamma, u : T, u' : T', \Gamma'; \Delta \vdash \mathtt{J} \implies \Gamma, u' : T', u : T, \Gamma'; \Delta \vdash \mathtt{J}$.
Similarly for `this`.
(3) $\Gamma; \Delta, c : \tau, \Delta' \vdash \mathtt{J} \wedge c \notin \mathsf{fn}(\mathtt{J}) \implies \tau = \mathtt{chan}$.
(4) $\Gamma; \Delta \vdash \mathtt{J} \wedge c \notin \mathsf{dom}(\Delta) \implies \Gamma; \Delta, c : \mathtt{chan} \vdash \mathtt{J}$.
(5) $\Gamma; \Delta \vdash \mathtt{J} \wedge \vdash T : \mathtt{tp} \wedge x \notin \mathsf{dom}(\Gamma) \implies \Gamma, x : T; \Delta \vdash \mathtt{J}$.
(6) $\Gamma; \Delta \vdash \mathtt{J} \wedge \vdash C : \mathtt{tp} \wedge \mathtt{this} \notin \mathsf{dom}(\Gamma) \implies \Gamma, \mathtt{this} : C; \Delta \vdash \mathtt{J}$.
(7) $\Gamma; \Delta, c : \tau \vdash \mathtt{J} \wedge c \notin \mathsf{fn}(\mathtt{J}) \implies \Gamma; \Delta \vdash \mathtt{J}$.
(8) $\Gamma, u : T; \Delta \vdash \mathtt{J} \wedge u \notin \mathsf{fnv}(\mathtt{J}) \implies \Gamma; \Delta \vdash \mathtt{J}$.
(9) $\Gamma, \Gamma'; \Delta, \Delta' \vdash \mathtt{J} \implies \Gamma; \Delta \vdash \mathtt{Env}$.

Proof. By induction on the size of the judgement $\mathtt{J}$. All cases are straightforward. We only list the proof for weakening with the case $\mathtt{J} = P_1 \mid P_2 : \mathtt{thread}$. After applying rule TT-Par we have two cases; we can apply the inductive

hypothesis to either the left branch or the right branch of the parallel composition. For example, choose the left branch. Therefore $\Gamma; \Delta_1, c : \texttt{chan} \vdash P_1 : \texttt{thread}$ and $\Delta_1, c : \texttt{chan} \asymp \Delta_2$ as $c \notin \mathsf{dom}(\Delta_2)$. Apply TT-Par to yield $\Gamma; \Delta_1, c : \texttt{chan} \odot \Delta_2 \vdash P_1 \mid P_2 : \texttt{thread}$. Then $\Gamma; \Delta_1 \odot \Delta_2, c : \texttt{chan} \vdash P_1 \mid P_2 : \texttt{thread}$ by definition of $\odot$. The other case proceeds similarly. □

D.2 Stores

Lemma 38 states properties about the type-safety of store access. Store accesses are defined as adding new variable and object identifier mappings, updating the fields of objects and the value held by a variable, and also retrieving information from variables and object fields. Lemma 38 allows the concatenation of disjoint stores and is useful in typing the $\texttt{deserialize}(e)$ operation.

Lemma 38 (Stores)**.** *Assuming that* $\Gamma; \Delta \vdash \sigma : \texttt{ok}$*. Then:*

(1) *If* $\Gamma \vdash v : T'$ *with* $x \notin \mathsf{dom}(\Gamma)$ *and* $T' <: T$ *then* $\Gamma, x : T; \Delta \vdash \sigma \cdot [x \mapsto v] : \texttt{ok}$.
(2) *Assume* $\Gamma \vdash x : T$ *and* $\Gamma \vdash v : T'$ *with* $T' <: T$*. Then* $\Gamma; \Delta \vdash \sigma[x \mapsto v] : \texttt{ok}$.
(3) $\Gamma \vdash x : T$ *implies* $\Gamma \vdash \sigma(x) : T'$ *with* $T' <: T$.
(4) *If* $\Gamma; \Delta \vdash (C, \vec{f} : \vec{v}, n, \{\vec{c}\}) : \texttt{ok}$ *and* $o \notin \mathsf{dom}(\Gamma)$ *then we have:*
$\Gamma, o : C; \Delta \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})] : \texttt{ok}$.
(5) *If* $\Gamma \vdash o : C$ *and* $\Gamma \vdash v : T_i'$ *with* $\mathsf{fields}(C) = \vec{T}\vec{f}$ *and* $T_i' <: T_i$*, then we have*
$\Gamma; \Delta \vdash \sigma[o \mapsto \sigma(o)[f_i \mapsto v]] : \texttt{ok}$.
(6) *Assume* $\Gamma \vdash o.f_i : T_i$ *with* $\sigma(o) = (C, \vec{f} : \vec{v})$*. Then* $\Gamma \vdash v_i : T_i'$ *where* $T_i' <: T_i$.
(7) *Suppose* $\Gamma, \Gamma'; \Delta, \Delta' \vdash \sigma' : \texttt{ok}$ *with* $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset$*, then* $\Gamma, \Gamma'; \Delta, \Delta' \vdash \sigma \cup \sigma' : \texttt{ok}$.
(8) $\Gamma; \Delta \vdash \sigma : \texttt{ok}$ *and* $\sigma' \subseteq \sigma$ *implies* $\Gamma; \Delta \vdash \sigma' : \texttt{ok}$.
(9) *Suppose* $\Gamma; \Delta \vdash \sigma : \texttt{ok}$ *and* $\sigma' = \mathsf{block}(\sigma, o, c)$ *with* $c \notin \mathsf{dom}(\Delta)$*,* $o \in \mathsf{dom}(\sigma)$*. Then we have that* $\Gamma; \Delta, c : \texttt{chan0}(\texttt{void}) \vdash \sigma' : \texttt{ok}$.
(10) *Suppose* $\Gamma; \Delta, c : \texttt{chan0}(\texttt{void}) \vdash \sigma : \texttt{ok}$ *and* $\sigma' = \mathsf{unblock}(\sigma, o, c)$ *with* $o \in \mathsf{dom}(\sigma)$ *and* $c \notin (\mathsf{fn}(\sigma) \setminus \mathsf{fn}(\sigma(o)))$*. Then we have that* $\Gamma; \Delta \vdash \sigma' : \texttt{ok}$.

Proof. All are mechanical. □

We list the standard lemma for the typability of the method body. The proof is routine.

Lemma 39 (Method body)**.** *Suppose* $\mathsf{mbody}(m, C, \texttt{CT}) = (x, e)$ *and that* $\mathsf{mtype}(m, C) = T \to U$ *with* $\vdash \texttt{CT} : \texttt{ok}$*. Then for some* C' *where* $C <: C'$

and some U' *where* $U' <: U$ *then we have* $x : T, \texttt{this} : C' \vdash e : \texttt{ret}(U')$.

Proof. Straightforward. □

D.3 Structural equivalence: Proof of Lemma 6

An important property to be shown is that the application of the structural equality rules given in Figure B.1 preserves the typing of a term. In order to prove this property, the next lemma is important: it yields natural properties for the composability of environments and is used in many of the later proofs.

Lemma 40 (Commutativity of composition and composability)**.**

(1) $\Delta_1 \asymp \Delta_2$ *and* $(\Delta_1 \odot \Delta_2) \asymp \Delta_3 \iff \Delta_2 \asymp \Delta_3$ *and* $\Delta_1 \asymp (\Delta_2 \odot \Delta_3)$.
(2) $\Delta_1 \asymp \Delta_2$ *and* $(\Delta_1 \odot \Delta_2) \asymp \Delta_3 \implies (\Delta_1 \odot \Delta_2) \odot \Delta_3 = \Delta_1 \odot (\Delta_2 \odot \Delta_3)$.

Proof. In both proofs, without loss of generality we consider singleton environments such that $\Delta_1 = \{c : \texttt{chanI}(U)\}$ and $\Delta_2 = \{c : \texttt{chanO}(U)\}$ with $\Delta_1 \odot \Delta_2 = \{c : \texttt{chan}\}$. For (1), we show only the left-to-right direction, the opposite direction is similar. The only interesting case is that Δ_1 and Δ_2 share the same channels. By the definition of $\asymp$, we know $c \notin \mathsf{dom}(\Delta_3)$. Since $\Delta_2 \odot \Delta_3 = \{c : \texttt{chanO}(U)\} \cup \Delta_3$, we have that $\Delta_2 \asymp \Delta_3$ as required. We can also easily check $\Delta_1 \odot (\Delta_2 \odot \Delta_3)$ is defined, thus by definition of $\asymp$, we have $\Delta_1 \asymp (\Delta_2 \odot \Delta_3)$, as desired. (2) proceeds in a similar manner to (1), adopting the same singleton environments. □

E Proof of invariant properties

E.1 Proofs of Lemma 9

(1) Straightforward by examining the reduction rules that modify class tables: Defrost and Download.

(2) Straightforward by examining the reduction rules, starting from the initial property.

(3) Assume $\mathsf{reachable}(\sigma_{im+1}, P_{im+1}, o)$ with $\sigma_{im+1}(o) = (C, \ldots)$.

Then there are four cases.

(a) Suppose $\mathsf{reachable}(\sigma_{im}, P_{im}, o)$ with $\sigma_{im}(o) = (C, \ldots)$.

Then by the inductive hypothesis, we have two possible situations:

i. $\mathsf{comp}(C, \mathtt{CT}_{im})$, hence by Lemma 9 (1) we have $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.
ii. We have that $P_{im} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \mid Q_{im}$ or that $P_{im} \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \mid Q_{im}$, with a superclass of C in $\vec{C}$. Examining the reduction rules, we see that if the last rule applied was DNothing, then by definition $\mathsf{comp}(C, \mathtt{CT}_{im+1})$. If the last reduction rule applied was Resolve, then we see that $P_{im+1} \equiv E[\mathtt{resolve}\ \vec{C'}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \mid Q_{im}$ with a superclass of C in $\vec{C'}$.

(b) Suppose $\neg\mathsf{reachable}(\sigma_{im}, P_{im}, o)$ with $\sigma_{im}(o) = (C, \ldots)$.
Then the last reduction rule applied must have been Return or Leave. We shall consider the case of the latter; the former is similar. Then we have that $P_{im+1} \equiv o.m(\mathtt{deserialize}(v))\ \mathtt{with}\ c \mid Q_{im+1}$. Since o moved from another location, we can conclude $\mathsf{RMI}(C)$, hence it must have been created at location l_i by New, or was there in the initial network (recall that the store entries for remotely callable object identifiers cannot leak to other locations).

In the case of New, we had that for some step k (with $N_0 \longrightarrow\!\!\!\!\!\rightarrow N_k \longrightarrow\!\!\!\!\!\rightarrow N_m$) when o was created, $\mathsf{comp}(C, \mathtt{CT}_{ik})$. Then by Lemma 9 (1), $\mathsf{comp}(C, \mathtt{CT}_{im+1})$. If o was present in the initial network, then by initial conditions and application of this Lemma again, $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.
(c) Suppose $\mathsf{reachable}(\sigma_{im}, P_{im}, o)$ with $o \notin \mathsf{dom}(\sigma_{im})$.
For this situation to arise, we have o of some class C such that $\mathsf{RMI}(C)$. Since identifiers to remotely callable objects cannot move, this case is complete by contradiction: no reduction to N_{m+1} can occur.
(d) Suppose $\neg\mathsf{reachable}(\sigma_{im}, P_{im}, o)$ with $o \notin \mathsf{dom}(\sigma_{im})$.
The last reduction rule applied was Defrost or New. In the case of the former, we see that $P_{im} \equiv E[\mathtt{defrost}(v;\ v')] \mid Q_{im}$ reduces to $P_{im+1} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \mid Q_{im+1}$ as required. In the case of the latter, we had $\mathsf{comp}(C, \mathtt{CT}_{im})$ and so by Lemma 9 (1) $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.

E.2 Proofs of Lemma 10

Induction on k. Below we assume $c \in \mathsf{blocked}(o, \sigma_{im+1})$ in $\mathtt{waiting}(c)$ ....
(1) (a) The base case is when $k = 1$. To generate $E_1[\mathtt{insync}\ o\ \{e\}], \sigma_{i1}$, it must have been the case that:

$$E_1[\mathtt{sync}\ (o)\ \{e\}], \sigma_{i0} \longrightarrow_l E_1[\mathtt{insync}\ o\ \{e\}], \sigma_{i1}$$

By the initial conditions, $\mathsf{lockcount}(\sigma_{i0}, o) = 0$ and so by application of Sync in we have that $\mathsf{lockcount}(\sigma_{i1}, o) = 1$ as required.

For the inductive case, we assume the hypothesis for N_m and show for N_{m+1}. Suppose we have a configuration of the form:

$$E_1[\ldots E_p[\texttt{insync}\ o\ \{e_p\}]\ldots], \sigma_{im+1},$$
$$\text{with } e_p \neq E'[\texttt{insync}\ o\ \{e'\}], E'[\texttt{waiting}(c)\ \ldots], E'[\texttt{ready}\ o\ \ldots]$$

Then the last reduction was either $e \longrightarrow_l e_p$ for some e or $E_p[\texttt{sync}\ (o)\ \{e_p\}] \longrightarrow_l E_p[\texttt{insync}\ o\ \{e_p\}]$.

By our assumption that $e_p \neq E'[\texttt{insync}\ o\ \{e'\}]$, in the first case the only interesting reduction rule to consider is the application of LeaveCritical. Suppose $e \equiv E[\texttt{insync}\ o\ \{v\}]$, then there are $p+1$ nested acquisitions of the monitor o. By the inductive hypothesis we have that $\mathsf{lockcount}(\sigma_{im}, o) = p+1$. By premise of LeaveCritical $\sigma_{im+1} = \mathsf{setcount}(\sigma_{im}, o, p)$, and so $\mathsf{lockcount}(\sigma_{im+1}, o) = p$ as required. The case for $e \equiv E[\texttt{insync}\ o\ \{\texttt{return}(c)\ v\}]$ is similar.

For the second case, the last reduction rule applied was Sync. Before application there are $p-1$ levels of nested monitors. By the inductive hypothesis it must be the case that $\mathsf{lockcount}(\sigma_{im}, o) = p-1$. Then by the premise of Sync we see that $\sigma_{im+1} = \mathsf{setcount}(\sigma_{im}, o, p)$, and so $\mathsf{lockcount}(\sigma_{im+1}, o) = p$ as required.

(b) Straightforward using (a) and inspecting Notify.

(c) Establishing that $p = n'$ is straightforward using (a).

(2) Base case, $k = 1$. Now suppose: $P_{i0}, \sigma_{i0}, \texttt{CT}_{i0} \longrightarrow_l P_{i1}, \sigma_{i1}, \texttt{CT}_{i1}$. By the initial conditions $\mathsf{lockcount}(\sigma_{i0}, o) = 0$, and by assumption $\mathsf{lockcount}(\sigma_{i1}, o) = n$ with $n > 0$. As no run-time syntax can exist in the network initially, the reduction rule applied was Sync. This means that $P_{i0} \equiv E[\texttt{sync}\ (o)\ \{e\}] \mid Q_{i0}$, and examining the conclusion of the rule $P_{i1} \equiv E[\texttt{insync}\ o\ \{e\}] \mid Q_{i1}$. Again by the initial conditions, e cannot contain $\texttt{insync}\ o\ \{\ldots\}$ as a sub-term, completing this case.

For the inductive step, suppose $P_{im}, \sigma_{im}, \texttt{CT}_{im} \longrightarrow_l P_{im+1}, \sigma_{im+1}, \texttt{CT}_{im+1}$. By assumption $\mathsf{lockcount}(\sigma_{im+1}, o) = p$ and $p > 0$. There are four distinct cases:

(1) $\mathsf{lockcount}(\sigma_{im}, o) = 0$. The last rule applied to derive P_{im} could be either Sync or Ready. To apply the former it must be the case that $P_{im} \equiv E_1[\texttt{sync}\ (o)\ \{e\}] \mid Q_{im}$. By the conclusion of this rule $P_{im+1} \equiv E_1[\texttt{insync}\ o\ \{e\}] \mid Q_{im+1}$ as required. For application of Ready, we have that

$$P_{im} \equiv E_1[\texttt{insync}\ o\ \{\ldots E_x[\texttt{insync}\ o\ \{E[\texttt{ready}\ o\ x]\}]\ldots\}] \mid Q_{im}$$

with $\texttt{insync}\ o\ \{\ldots\}$ not a sub-term of E by 1.(a). Then it remains to show that $x = p$, however this is immediate by inspection of the rule Ready.

(2) $\mathsf{lockcount}(\sigma_{im}, o) = p - 1$. The only rule applicable in this situation is SYNC. Therefore by the inductive hypothesis:

$$P_{im} \equiv E_1[\texttt{insync}\ o\ \{\ldots E_{p-1}[\texttt{insync}\ o\ \{E_p[\texttt{sync}\ (o)\ \{e\}]\}]\ldots\}]\,|\,Q_{im}$$

with $\texttt{insync}\ o\ \{\ldots\}$ not a sub-term of E_p. Examining SYNC, we have that:

$$P_{im+1} \equiv E_1[\texttt{insync}\ o\ \{\ldots E_{p-1}[\texttt{insync}\ o\ \{E_p[\texttt{insync}\ o\ \{e\}]\}]\ldots\}]\,|\,Q_{im+1}$$

By the initial conditions $\texttt{insync}\ o\ \{\ldots\}$ cannot be a sub-term of e, so this completes the case.

(3) $\mathsf{lockcount}(\sigma_{im}, o) = p$. Straightforward.

(4) $\mathsf{lockcount}(\sigma_{im}, o) = p + 1$. Only one rule is applicable in this situation: LEAVECRITICAL. For this rule to have been applied, we must have had that $P_{im} \equiv E_1[\texttt{insync}\ o\ \{\ldots E_{p+1}[\texttt{insync}\ o\ \{e\}]\ldots\}]\,|\,Q_{im}$ with $e = v$ or $e = \texttt{return}(c)\ v$. We consider the case for v, the case for a return is similar. Examining LEAVECRITICAL, we see that P_{im+1} must be $E_1[\texttt{insync}\ o\ \{\ldots E_{p+1}[v]\ldots\}]\,|\,Q_{im+1}$ with $\texttt{insync}\ o\ \{\ldots\}$ not a sub-term of E_{p+1} by our earlier assumption, completing this case. □

E.3 Proofs of the invariant properties

$\mathsf{Inv}(1)$ By Lemma 9 (1).

$\mathsf{Inv}(2)$ Suppose $P_{im} \not\equiv E[\texttt{new}\ C(\vec{v})]\,|\,Q_{im} \longrightarrow_{l_i} P_{im+1} \equiv E[\texttt{new}\ C(\vec{v})]\,|\,Q_{im}$, then one of three possible reduction rules was applied:

(a) We applied CONG. Then $P_{im} \equiv P'_{im}\,|\,Q_{im}$ with $P'_{im} \longrightarrow_{l_i} E[\texttt{new}\ C(\vec{v})]$.
Then if $m = 0$, by the initial condition $\mathsf{Inv}(2)'$ and Lemma 9 (1) we have $\mathsf{comp}(C, \texttt{CT}_{im+1})$, irrespective of the reduction rule applied.
Suppose $m > 0$. Therefore $P'_{im} \equiv E'[\texttt{new}\ C(\vec{v})]$ for some context E'. By the inductive assumption we have $\mathsf{comp}(C, \texttt{CT}_{im})$ and again by Lemma 9 (1) it is the case that $\mathsf{comp}(C, \texttt{CT}_{im+1})$.

(b) We applied DNOTHING. So $P_{im} \equiv E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_j\ \texttt{in}\ \texttt{new}\ C(\vec{v})\,|\,Q_{im}$ with $\vec{C} \in \mathsf{dom}(\texttt{CT}_i)$. Then it must be the case that $m > 0$ since the download expression is not permissible runtime syntax in an initial network. In order to download nothing, it must have been the case that $P_{ik} \equiv E[\texttt{download}\ \vec{D}\ \texttt{from}\ l_j\ \texttt{in}\ e]$ with $C \in \vec{D}$ and $k < m$ (i.e. class C was downloaded at some point in the past). Then examining the rules DOWNLOAD and RESOLVE we can straightforwardly observe that they iterate until all superclasses of C are downloaded. Therefore using Lemma 9 (1) we have trivially that $\mathsf{comp}(C, \texttt{CT}_{im+1})$.

(c) We applied NEWL. So $\mathsf{comp}(C, \texttt{CT}_{im})$ hence $\mathsf{comp}(C, \texttt{CT}_{im+1})$ by Lemma 9 (1) as required.

$\mathsf{Inv}(3)$ There are two interesting sub-cases:
(a) The last applied reduction rule was DOWNLOAD. Then

$$P_{im} \equiv E[\texttt{download}\ C\ \texttt{from}\ l_j\ \texttt{in}\ e] \mid Q_{im}$$
$$\text{and}\ \ P_{im} \longrightarrow_{l_i} E[\texttt{resolve}\ C\ \texttt{from}\ l_j\ \texttt{in}\ e]$$

Since downloading did not fail (the assumption that $N_{m+1} \not\models \mathsf{Err}$), there must exist a location l_j with $C \in \mathsf{dom}(\texttt{CT}_{jm})$. By premise of DOWNLOAD, $C \in \mathsf{dom}(\texttt{CT}_{im+1})$ and $\texttt{CT}_{im+1}(C) = \texttt{CT}_{jm}$ modulo class name labelling as required.
(b) The last applied reduction rule was DEFROST. Then

$$P_{im} \equiv E[\texttt{defrost}(v;\ \lambda(T\ x).(\nu\,\vec{u})(l_j, e, \sigma, \texttt{CT}))]$$

with $\texttt{CT} \subseteq \texttt{CT}_{jk}$ where $k < m$. Straightforwardly, by premise of DEFROST we have that $\texttt{CT}_{im+1} = \texttt{CT}_{im} \cup \texttt{CT}[\vec{C}^m/\vec{C}]$ for some classes $\vec{C}$, and so making a similar argument to the previous sub-case, any duplicate classes *must* have the same definition, modulo class labelling.

$\mathsf{Inv}(4)$ The only interesting cases are those where the set of free variables of a term changes with reduction. There are three such cases. Without loss of generality, we consider only a single thread containing no free variables for a single location with an empty store:
(a) The last applied reduction rule was DEC. Then suppose

$$l_i[E[T\ x = v], \emptyset, \texttt{CT}_i] \longrightarrow\equiv (\nu\,x)(l_i[E[v], [x \mapsto v], \texttt{CT}_i])$$

By $\mathsf{Inv}(12)$, we have $\mathsf{fv}(v) = \emptyset$. Before reduction we have $\mathsf{fv}(E[T\ x = v]) = \emptyset$, after reduction we potentially have that $\mathsf{fv}(E[v]) = \{x\}$, however we see that $\mathsf{dom}([x \mapsto v]) = \{x\}$ and the new identifier is restricted at the network level. Therefore this case is complete.
(b) The last applied reduction rule was METHINVOKE. Then suppose

$$l_i[o.m(v)\ \texttt{with}\ c, \emptyset, \texttt{CT}_i]$$
$$\longrightarrow\equiv (\nu\,x)(l_i[e[o, \texttt{return}(c)/\texttt{this}, \texttt{return}], [x \mapsto v], \texttt{CT}_i])$$

where $\mathsf{mbody}(m, C, \texttt{CT}_i) = (x, e)$, and again by $\mathsf{Inv}(12)$, $\mathsf{fv}(v) = \emptyset$. We must show that $\mathsf{fv}(e) \subseteq [x \mapsto v] \subseteq \{x\}$. However by definition of substitution, we know that $\mathsf{fv}(e[o, \texttt{return}(c)/\texttt{this}, \texttt{return}]) = \mathsf{fv}(e)$. Given that the network configuration is well-typed, it must be the case that $x : T, \texttt{this} : C \vdash e : \texttt{ret}(U)$, i.e. $\mathsf{fv}(e) \subseteq \{x\}$. This concludes the case.

(c) The last applied reduction rule was DEFROST. Suppose

$$\begin{aligned} & l_i[E[\texttt{defrost}(v;\ \lambda(T\ x).(\nu\,\vec{u})(l_j, e, \sigma, \texttt{CT}))], \emptyset, \texttt{CT}_i] \\ \longrightarrow\equiv\ & (\nu\,\vec{u}x)(l_i[E[\texttt{download}\ \vec{F}\ \texttt{from}\ l_j\ \texttt{in sandbox}\ \{e\}], \sigma \cdot [x \mapsto v], \\ & \qquad \texttt{CT} \cup \texttt{CT}_i[\vec{C}^{l_j}/\vec{C}]) \end{aligned}$$

where $\mathsf{fv}(v) = \mathsf{fv}(\lambda(T\ x).(\nu\,\vec{u})(l_j, e, \sigma, \texttt{CT})) = \emptyset$ by $\mathsf{Inv}(12)$. Straightforwardly we have that $\mathsf{fv}(e) \subseteq \mathsf{dom}(\sigma \cdot [x \mapsto v]) \subseteq \{\vec{u}x\}$ to complete this case.

$\mathsf{Inv}(5)$ Here we only need to examine the case when the last applied rule was DEFROST. This case is straightforward: all new identifiers added to the store during defrosting are restricted with fresh names, therefore there can be no overlap of store entries between producer and consumer locations. Moreover, by $\mathsf{Inv}(15)$, the only free object identifiers contained in frozen values are of remoteable objects, hence the rules METHREMOTE and LEAVE cannot move shared store entries across the network.

$\mathsf{Inv}(6)$ We have two cases to consider: when the inductive hypothesis is true because the antecedent of the implication is true (and hence the consequent is also true), and when the antecedent is false. For the former case suppose that:

$$\begin{aligned} & o \in \mathsf{fn}(F_{im+1}) \cap \mathsf{fn}(F_{jm+1}),\ o \in \mathsf{fn}(F_{im}) \cap \mathsf{fn}(F_{jm}), \\ \text{and } & \exists!k.\sigma_{km}(o) = (C, \ldots) \text{ with } \mathsf{RMI}(C) \end{aligned}$$

Then by $\mathsf{Inv}(5)$ we have that o cannot exist in the domain of more than one store. By $\mathsf{Inv}(8)$ we observe that since $o \in \mathsf{fn}(F_{im+1})$, the entry for o cannot have been garbage collected. Hence $\exists\ !k.\sigma_{km+1}(o) = (C, \ldots)$, and since there are no operations to change the "remoteability" of a class, $\mathsf{RMI}(C)$ holds, completing the case.

For the latter case suppose that:

$$o \in \mathsf{fn}(F_{im+1}) \cap \mathsf{fn}(F_{jm+1}), \quad o \notin \mathsf{fn}(F_{im}) \cap \mathsf{fn}(F_{jm}) \tag{E.1}$$

This indicates that a free name moved between two locations. This can happen in two ways: by application of METHREMOTE or RETURN. We show the case of the latter, since the former is proved by the same argument. Assume

$$l_j[\texttt{go}\ v\ \texttt{to}\ c] \,|\, l_i[Q_{im}] \longrightarrow l_j[\ldots] \,|\, l_i[\texttt{return}(c)\ \texttt{deserialize}(v) \,|\, Q_{im}]$$

By typability of F_{im+1} we have that $\Gamma; \Delta \vdash v : \texttt{unit} \to D$ for some class D, i.e. v is a *frozen* value. Assume there exists o such that $o \in \mathsf{fn}(v)$ and (E.1) holds. By $\mathsf{Inv}(8)$, there exists a store entry for o "somewhere" and by $\mathsf{Inv}(5)$ this entry must be unique. Hence there exists unique k such that $\sigma_{km+1}(o) = (C, \ldots)$.

Now by $\mathsf{Inv}(15)$, the only free identifiers in a frozen value must be of remoteable classes, hence $\mathsf{RMI}(C)$. This completes the case.

$\mathsf{Inv}(7)$ There are two sub-cases. First assume that

$$o \in \mathsf{fn}(F_{im+1}) \wedge \exists k.\sigma_{km+1}(o) = (C, \ldots) \wedge \neg\mathsf{RMI}(C)$$

and we shall prove that $k = i$.

(a) Suppose $o \in \mathsf{fn}(F_{im}) \wedge \exists k'.\sigma_{k'm}(o) = (C, \ldots) \wedge \neg\mathsf{RMI}(C) \wedge k' = i$. Then by the inductive assumption we can derive that $o \in \mathsf{dom}(\sigma_{im})$. By $\mathsf{Inv}(8)$ we have that $o \in \mathsf{dom}(\sigma_{im+1})$ which implies $i = k = k'$ as required.

(b) Suppose that $o \notin \mathsf{fn}(F_{im})$, then the last applied reduction step applied must have somehow created a new object identifier o. Examining the reduction rules we have four cases (although two are trivial):

(i) The last applied rule was New.

$$E[\texttt{new}\ C(\vec{v})] \,|\, Q_{im}, \sigma_{im} \longrightarrow_{l_i} (\nu\, o)(E[o] \,|\, Q_{im}, \sigma_{im} \cdot [o \mapsto \ldots])$$

Then straightforwardly $k = i$ because $o \in \mathsf{dom}(\sigma)_{im+1}$.

(ii) The last applied rule was Defrost.

$$\begin{array}{rl} & E[\texttt{defrost}(v;\ \lambda(T\ x).(\nu\,\vec{u})(l_j, e, \sigma, \texttt{CT}))] \,|\, Q_{im}, \sigma_{im} \\ \longrightarrow_{l_i} & (\nu\,\vec{u}x)(E[\ldots] \,|\, Q_{im}, \sigma_{im} \cup \sigma \cdot [x \mapsto v]) \end{array}$$

Then by $\mathsf{Inv}(15)$, the frozen value can contain no free identifiers to non-remotely callable objects and so $o \in \vec{u}$ which entails that $o \in \mathsf{dom}(\sigma)$ and $k = i$.

(iii) The last applied rule was Leave or MethRemote. Then (omitting stores and class tables for clarity)

$$\begin{array}{rl} & l_j[\texttt{go}\ o.m(v)\ \texttt{with}\ c] \,|\, l_i[Q_{im}] \\ \longrightarrow & l_j[\ldots] \,|\, l_i[o.m(\texttt{deserialize}(v))\ \texttt{with}\ c \,|\, Q_{im}] \end{array}$$

By assumption, $\Gamma; \Delta \vdash N : \texttt{net}$ and therefore $\Gamma \vdash v : \texttt{unit} \to D$ for some class D. Then by $\mathsf{Inv}(15)$, we know that any free names appearing inside v must be identifiers for *remoteable* objects. Therefore this case holds vacuously.

$\mathsf{Inv}(8)$ Assume: $o \in \mathsf{fn}(F_{im}) \wedge \exists k\ 1 \leq k \leq n.\ o \in \mathsf{dom}(\sigma_{km})$ and $o \in \mathsf{fn}(F_{im+1})$. Then, by examination of the rules for structural equivalence in Figure B.1, we see that the only way to remove an object identifier is when it does not exist in the free names of the remainder of the network. Since $o \in \mathsf{fn}(F_{im+1})$ by assumption, it must be the case that $\exists k\ 1 \leq k \leq n.\ o \in \mathsf{dom}(\sigma_{km+1})$ as required.

$\mathsf{Inv}(9)$ Only the cases for $R_i \equiv o.m(e)\ \texttt{with}\ c$ and $R_i \equiv E[o.f = e]$ are shown, as the others use the same basic method.

(a) Suppose $P_{im+1} \equiv o.m(e)$ `with` $c \,|\, Q_{im+1}$. Examining the structure of this thread, we see that there were two possible rules applied in the last reduction step:
LeaveSandbox or MethLocal.

(i) Let $P_{im} \equiv E[o.m(v)] \,|\, Q_{im}$. Then this is a local method call, and by the premises of MethLocal, we had $o \in \mathsf{dom}(\sigma_{im})$, hence $\sigma_{im} = (C, \ldots)$. Then by Lemma 9 (3) we have $\mathsf{comp}(C, \mathtt{CT}_{im})$. By monotonicity of stores and class tables, $\sigma_{im+1}(o) = (C, \ldots)$ and $\mathsf{comp}(C, \mathtt{CT}_{im+1})$ as required.

(ii) Let $P_{im} \equiv Q_{im}$. Then this is a remote method call, and by the premises of Leave, we have $o \in \mathsf{dom}(\sigma_{im})$. Then proof proceeds as in the previous case.

(b) Assume $P_{im+1} \equiv E[o.f = e] \,|\, Q_{im+1}$. Then we have two main cases. If $P_{im} \equiv E[o.f = e'] \,|\, Q_{im}$ then by the inductive hypothesis, this case is complete. However suppose $P_{im} \equiv E[e'.f = e] \,|\, Q_{im}$, then we must perform a case analysis on the step $e' \longrightarrow o$.

(i) Suppose the last rule applied was New. Then by $\mathsf{Inv}(2)$ we have that $\mathsf{comp}(C, \mathtt{CT}_{im})$, hence by Lemma 9 (1) $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.

(ii) Suppose the last applied rule was Fld. Then $P_{im} \equiv E[(o'.f').f = e] \,|\, Q_{im}$. Then by typability of N_m we have that $\Gamma, \vec{u} : \vec{T} \vdash (o'.f').f : C$ and $\neg\mathsf{RMI}(C)$. Then by $\mathsf{Inv}(8)$ and $\mathsf{Inv}(7)$ we have that $\sigma_{im}(o) = (C, \ldots)$. Then by Lemma 9 (3), $\mathsf{comp}(C, \mathtt{CT}_{im})$. By monotonicity of class tables and stores, $\sigma_{im+1}(o) = (C, \ldots)$ and $\mathsf{comp}(C, \mathtt{CT}_{im+1})$ as required.

(iii) Suppose the last rule applied was Var. Then $P_{im} \equiv E[x.f = e] \,|\, Q_{im}$. By typability, $\Gamma, \vec{u} : \vec{T} \vdash x.f : C$ with $\neg\mathsf{RMI}(C)$. By reduction, we must have that $[x \mapsto o] \in \sigma_{im}$ and so o is reachable from thread P_{im}. Again by $\mathsf{Inv}(8)$ and $\mathsf{Inv}(7)$ we have that $\sigma_{im}(o) = (C, \ldots)$. Then by Lemma 9 (3), $\mathsf{comp}(C, \mathtt{CT}_{im})$. By monotonicity of class tables and stores, $\sigma_{im+1}(o) = (C, \ldots)$ and $\mathsf{comp}(C, \mathtt{CT}_{im+1})$ as required.

(iv) Suppose the last rule applied was Ass. Then $P_{im} \equiv E[(x = o).f = e] \,|\, Q_{im}$. Then by typability, $\Gamma, \vec{u} : \vec{T} \vdash x = o : C$ for some C such that $\neg\mathsf{RMI}(C)$. Hence by typability $\Gamma, \vec{u} : \vec{T} \vdash o : C'$ such that $C' <: C$. By well-formedness of the class signature, if $\neg\mathsf{RMI}(C)$ then $\neg\mathsf{RMI}(C')$ also. Then this case straightforward as above, by establishing that o must be in the store due to the locality of its class.

(v) Suppose the last rule applied was FldAss. Similar to the case for Ass.

(vi) The last rule applied was LeaveSandbox. This means that we have: $P_{im} \equiv E[\mathtt{sandbox}\ \{o\}.f = e] \,|\, Q_{im}$. Then this is also straightforward, noting that $\Gamma, \vec{u} : \vec{T} \vdash o : C$ with $\neg\mathsf{RMI}(C)$.

$\mathsf{Inv}(10)$ Straightforward by the definition of $\Delta_1 \asymp \Delta_2$.

$\mathsf{Inv}(11)$ Straightforward by the definition of $\Delta_1 \asymp \Delta_2$.

$\mathsf{Inv}(12)$ We investigate the cases where a value comes into a redex position. Assume $P_{im+1} \equiv E[v] \,|\, Q_{im+1}$, then we perform a case analysis as follows.

(a) The last rule was New. Then this case is trivial.
(b) The last rule was Var. Then $P_{im} \equiv E[x] \mid Q_{im}$, and this case is straightforward by $\mathsf{Inv}(13)$.
(c) The last rule was Fld. Then $P_{im} \equiv E[o.f] \mid Q_{im}$, and this case is straightforward by $\mathsf{Inv}(14)$.
(d) The last rule was Freeze. Then $P_{im} \equiv E[\texttt{freeze}[t](T\ x)\{e\}] \mid Q_{im}$. By reduction, $v = \lambda(T\ x).(\nu\, \vec{u})(l_i, e, \sigma, \texttt{CT})$. Examining the definition of fv, we see that $\mathsf{fv}(v) = (\mathsf{fv}(e) \setminus \{x\}) \cup \mathsf{fv}(\sigma) \cup \mathsf{fv}(\texttt{CT})$. By premises of Freeze, we see that σ is derived from σ_{im}. Hence by $\mathsf{Inv}(13)$ and $\mathsf{Inv}(14)$, all values must be closed, so $\mathsf{fv}(\sigma) = \emptyset$. Similarly, $\texttt{CT}$ is derived from $\texttt{CT}_{im}$ and so since $\vdash \texttt{CT}_{im} : \texttt{ok}$, we have that $\mathsf{fv}(\texttt{CT}) = \emptyset$. By typability of P_{im}, we see that $\mathsf{fv}(e) \subseteq \{x\}$, so we can conclude that $\mathsf{fv}(v) = \emptyset$ as required.
(e) The last rule applied was DNothing. This means that we have $P_{im} \equiv E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_j\ \texttt{in sandbox}\ \{v\} \mid Q_{im}$. However this situation only arises after a frozen value has been defrosted. Therefore by the inductive hypothesis, we know that v can contain no free variables.

$\mathsf{Inv}(13)$ For this invariant, we check the cases where new variable mappings are added to the store, or when an existing mapping is changed. Assume $\sigma_{im+1}(x) = v$. Then
(a) Suppose the last reduction rule applied was Dec. Then $P_{im} \equiv E[T\ x = v] \mid Q_{im}$. Then by $\mathsf{Inv}(12)$ we have that $\mathsf{fv}(v) = \emptyset$, so this case is straightforward.
(b) The last rule applied was Defrost. Then $P_{im} \equiv E[\texttt{defrost}(v;\ \lambda(T\ x).(\nu\, \vec{u})(l_j, e, \sigma, \texttt{CT}))] \mid Q_{im}$. However by $\mathsf{Inv}(12)$, we have that the frozen value contains no free variables, so $\mathsf{fv}(\sigma) = \emptyset$ and so any mappings added to σ_{im} are closed.
(c) The last rule was Ass. Again straightforward by $\mathsf{Inv}(12)$.

$\mathsf{Inv}(14)$ For this invariant, check the cases where new object mappings are added to the store, or when an existing mapping is changed. Then, assuming $\sigma_{im+1}(o) = (C, \vec{f} : \vec{v})$, we investigate when the last rule was New, Defrost or FldAss. All are straightforward by application of $\mathsf{Inv}(12)$.

$\mathsf{Inv}(15)$ The only interesting case is when the last applied reduction rule was Freeze. We show only the case when $t = \texttt{eager}$, as the others are similar. Suppose:

$$\begin{aligned} P_{im} &\equiv E[\texttt{freeze}[\texttt{eager}](T\ x)\{e\}] \mid Q_{im} \\ \longrightarrow_{l_i}\ P_{im+1} &\equiv E[\lambda(T\ x).(\nu\, \vec{u})(l_i, e, \sigma, \texttt{CT})] \mid Q_{im} \end{aligned}$$

Then by the premises of Freeze we have:

$$\sigma_y = \{[y \mapsto \sigma_{im}(y)] \mid y \in \mathsf{fv}(e) \setminus \{x\}\} \quad \texttt{CT}' = \mathsf{cg}(\texttt{CT}, \mathsf{icl}(e) \cup \mathsf{icl}(\sigma))$$
$$\sigma = \mathsf{og}(\sigma_{im}, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y \text{ with } \{\vec{u}\} = \mathsf{dom}(\sigma)$$

As a preliminary note, by typability of N_m we have that $\vdash \texttt{CT} : \texttt{ok}$ and so $\mathsf{fn}(\texttt{CT}) = \emptyset$. Therefore: $\mathsf{fn}(\lambda(T\ x).(\nu\,\vec{u})(l_i, e, \sigma, \texttt{CT})) = (\mathsf{fn}(e) \cup \mathsf{fn}(\sigma)) \setminus \{\vec{u}\}$. Now we must show that if $u \in (\mathsf{fn}(e) \cup \mathsf{fn}(\sigma)) \setminus \{\vec{u}\}$ then $\Gamma \vdash u : C$ with $\mathsf{RMI}(C)$. We shall show a contradiction, suppose such a u exists but instead of $\mathsf{RMI}(C)$ we have $\neg\mathsf{RMI}(C)$. By Lemma 11 (1b) we have that u was reachable from the expression e but whose mapping was not present in σ_{im} (if it was, it would have been copied and hence restricted in vector $\vec{u}$). However by conjunction of $\mathsf{Inv}(8)$ and $\mathsf{Inv}(7)$ we have that $\sigma_{im}(u) = (C, \ldots)$ should hold. Hence it would have been present and would have been copied by og. Contradiction.

$\mathsf{Inv}(16)$ Suppose $P_{im+1} \equiv E[\texttt{ready}\ o\ n] \mid Q_{im+1}$. There are only two interesting cases: the last reduction rule applied was Notify, or it was NotifyAll.

(a) Suppose $P_{im} \equiv E'[o.\texttt{notify}] \mid E[\texttt{waiting}(c)\ n] \mid Q_{im}$. By typability of this term we have that $n > 0$, and by the premises of Notify we see that $c \in \mathsf{blocked}(\sigma_{im}, o)$. Then by Lemma 10 we have that $\mathsf{insync}(o, E)$ with n levels of nesting.

(b) Suppose $P_{im} \equiv E'[o.\texttt{notifyAll}] \mid E[\texttt{waiting}(c)\ n] \mid Q_{im}$. Then this case follows in the same way as the previous.

$\mathsf{Inv}(17)$ Assume $P_{im+1} \equiv E[\texttt{waiting}(c)\ n] \mid Q_{im+1}$. Then there is only one interesting case to consider, when $P_{im} \equiv E[o.\texttt{wait}] \mid Q_{im}$. By premise of Wait we have that $\mathsf{insync}(o, E)$, and consequently by Lemma 10, $n > 0$. Since channel c is created fresh, we know that it is stored in the blocked queue of at most one object, and again by the premise of Wait we see that $\sigma_{im+1} = \mathsf{block}(\sigma_{im}, o, c)$, therefore it exists in exactly one place: the blocked queue of o. This completes the case.

E.4 Proof of mutual exclusion

This subsection proves Proposition 17.

By induction on the number of threads synchronised on the object o, written n. The base case is straightforward; take $n = 1$ then $P \equiv E_1[\texttt{insync}\ o\ \{e_1\}] \mid Q$. e_1 can be of any form and still satisfy the condition that at most one thread can execute in its critical section.

For the inductive step, we assume that the property holds for $n - 1$ threads in parallel. Now we write P as follows:

$$P \equiv E_1[\texttt{insync}\ o\ \{e_1\}] \mid \cdots \mid E_{n-1}[\texttt{insync}\ o\ \{e_{n-1}\}] \mid E_n[\texttt{insync}\ o\ \{e_n\}] \mid Q \tag{E.2}$$

We shall show that either:

$$\forall j.1 \leq j \leq n.\ (e_j = E'_j[\texttt{waiting}(c)\ n \ldots] \vee e_j = E'_j[\texttt{ready}\ o\ \ldots]) \quad \text{(E.3)}$$
$$\text{or} \quad \exists! j.1 \leq j \leq n.\ (e_j \neq E'_j[\texttt{waiting}(c)\ n \ldots] \wedge e_j \neq E'_j[\texttt{ready}\ o\ \ldots]) \quad \text{(E.4)}$$

with $c \in \mathsf{blocked}(o, \sigma)$. By the inductive assumption, we have that:

$$\forall j.1 \leq j \leq n-1.\ (e_j = E'_j[\texttt{waiting}(c)\ n \ldots] \vee e_j = E'_j[\texttt{ready}\ o\ \ldots]) \quad \text{(E.5)}$$
$$\text{or} \quad \exists! j.1 \leq j \leq n-1.\ (e_j \neq E'_j[\texttt{waiting}(c)\ n \ldots] \wedge e_j \neq E'_j[\texttt{ready}\ o\ \ldots]) \quad \text{(E.6)}$$

For (E.5) if $e_n = \texttt{waiting}(c)\ n$ or $e_n = \texttt{ready}\ o\ n$ then we can immediately conclude (E.3) as required. Similarly, if e_n is not of this form then we can safely conclude (E.4).

However, if we have the situation (E.6) then the nature of the new thread is important—it cannot be executing inside its critical section. If e_n is waiting or ready, then (E.4) is preserved. However consider that e_n *is* executing within its critical section. We shall show that this situation cannot arise by showing a contradiction.

Without loss of generality, consider only two threads executing in their critical section simultaneously:

$$E_1[\texttt{insync}\ o\ \{e_1\}] \mid E_2[\texttt{insync}\ o\ \{e_2\}]$$

Assume neither e_1 nor e_2 are of the form $E''[\texttt{waiting}(c)\ n]$ or $E''[\texttt{ready}\ o\ n]$. In order to reach such a situation, one thread must have entered its critical section while the other was still active in its. Therefore:

$$E_1[\texttt{insync}\ o\ \{e_1\}] \mid E_2[e'_2], \sigma, \texttt{CT} \longrightarrow_l E_1[\texttt{insync}\ o\ \{e_1\}] \mid E_2[\texttt{insync}\ o\ \{e_2\}], \sigma', \texttt{CT}$$

e'_2 can take two shapes: $e'_2 = \texttt{ready}\ o\ \ldots$ or $e'_2 = \texttt{sync}\ (o)\ \{e_2\}$. In the first case, the only reduction rule applicable is READY, which has $\mathsf{lockcount}(\sigma, o) = 0$ as a premise. However by Lemma 10 we can conclude that $\mathsf{lockcount}(\sigma, o) > 0$, giving rise to an immediate contradiction. The same argument may be made for the second form of e'_2, where SYNC is used.

Using this, we conclude that e_n must be of the form $E'_n[\texttt{waiting}(c)\ o \ldots]$ or $E'_n[\texttt{ready}\ o\ \ldots]$, which establishes (E.4) as required. □

F Proofs for type preservation

F.1 Proofs for Substitution Lemma 12

By induction on the structure of the expression e using Lemma 6. The proof is standard, so we only list one case. Suppose $\Gamma, x : T; \Delta, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c]^U : \texttt{thread}$. Then this is derived from TT-Await with the premise: $\Gamma, x : T; \Delta \vdash E[\ ]^U : \texttt{thread}$ with $c \notin \mathsf{dom}(\Delta)$. We apply the inductive hypothesis obtaining $\Gamma; \Delta \vdash E[\ ]^U[v/x] : \texttt{thread}$. Since Δ is unchanged, the side condition $c \notin \mathsf{dom}(\Delta)$ still holds, and we can apply rule TT-Await to yield $\Gamma; \Delta, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c]^U[v/x] : \texttt{thread}$, as required. □

F.2 Proof of Theorem 13

(1) The proof proceeds by induction on the length of reduction sequence with a case analysis on the final reduction rule applied. When $\sigma = \sigma'$ or $\texttt{CT} = \texttt{CT}'$ we shall omit them.

Case 1 (Var)**.** Use Lemma 38 (3).

Case 2 (Cond)**.** Straightforward by the inductive hypothesis.

Case 3 (While)**.** Standard.

Case 4 (Fld)**.** Straightforward by Lemma 38 (6).

Case 5 (Ass)**.** Straightforward using Lemma 38 (2).

Case 6 (FldAss)**.** Use Lemma 38 (5).

Case 7 (New)**.** Assume $\texttt{new}\ C(\vec{v}), \sigma \longrightarrow_l (\nu\, o)(o, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)])$ and $\Gamma \vdash \texttt{new}\ C(\vec{e}) : C$. To derive this, TE-New must have been applied with premises $\mathsf{fields}(C) = \vec{T}\vec{f}$, $T_i' <: T_i$, $\Gamma \vdash e_i : T_i'$ and $\vdash C : \texttt{tp}$. Using this we can derive that $\Gamma \vdash (C, \vec{f} : \vec{v}, 0, \emptyset) : \texttt{ok}$. Since o is fresh and therefore not in Γ, we can apply Lemma 38 (4) to complete the case.

Case 8 (NewR)**.** Similar to the case for New. Assume $\Gamma \vdash \texttt{new}\ C^l(\vec{e}) : C$, we see TE-New was applied with the important premise that $\vdash C : \texttt{tp}$. Then we can apply TE-ClassLoad to derive $\Gamma \vdash \texttt{download}\ C\ \texttt{from}\ l\ \texttt{in}\ \texttt{new}\ C^l(\vec{e}) : C$ as required.

Case 9 (NewL)**.** As both $\texttt{new}\ C^l(\vec{v})$ and $\texttt{new}\ C(\vec{v})$ are typed by the same rule, this case is immediate.

Case 10 (DEC)**.** Straightforward by Lemma 38 (1).

Case 11 (CONG)**.** Use Lemma 12(3).

Case 12 (RESOLVE)**.** By TE-CLASSLOAD.

Case 13 (DNOTHING)**.** By TE-CLASSLOAD.

Case 14 (READY)**.** Assume $\texttt{ready}\ o\ n, \sigma \longrightarrow_l \epsilon, \sigma'$ with $\Gamma \vdash \texttt{ready}\ o\ n : \texttt{void}$ and $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$. Trivially, $\Gamma \vdash \epsilon : \texttt{ok}$, therefore all that remains is to show that $\Gamma; \Delta_2 \vdash \sigma' : \texttt{ok}$. However since to derive $\texttt{ready}\ o\ n$, we had to apply TE-READY which has the premise that $n > 0$, then we can trivially conclude that $\Gamma; \Delta_2 \vdash \sigma' : \texttt{ok}$.

Case 15 (FREEZE)**.** We assume the `eager`-mode of operation, others are similar. Assume $\texttt{freeze}[\texttt{eager}](T\ x)\{e\}, \sigma, \texttt{CT} \longrightarrow_l \lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \texttt{CT}'), \sigma, \texttt{CT}$ and $\Gamma \vdash \texttt{freeze}[\texttt{eager}](T\ x)\{e\} : T \to U$. To infer this, TE-FREEZE was used with premise $\Gamma, x : T \vdash e : U$. The premises of FREEZE are

$$\sigma' = \mathsf{og}(\sigma, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y \text{ with } \sigma_y = \{[y \mapsto \sigma(y)] \mid y \in \mathsf{fv}(e) \setminus \{x\}\} \quad \text{(a)}$$
$$\{\vec{u}\} = \mathsf{dom}(\sigma') \quad \text{(b)}$$
$$\texttt{CT}' = \mathsf{cg}(\texttt{CT}, \mathsf{icl}(e) \cup \mathsf{icl}(\sigma')) \quad \text{(c)}$$

From (a) we can apply Lemma 11 (1d) to obtain

$$\Gamma; \emptyset \vdash \mathsf{og}(\sigma, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) : \texttt{ok}$$

By definition, we have $\sigma_y \subseteq \sigma$. Then by Lemma 38 (8), $\Gamma; \Delta \vdash \sigma_y : \texttt{ok}$. However σ_y, by construction, only contains mappings from variables, i.e. there are no store objects (and hence no channels) in its co-domain, therefore $\Gamma; \emptyset \vdash \sigma_y : \texttt{ok}$ by Lemma 37 (strengthening). From this knowledge, we can apply Lemma 38 (7) to obtain: $\Gamma; \emptyset \vdash \sigma' : \texttt{ok}$.

Now considering $\texttt{CT}'$, by (c) we see that $\texttt{CT}'$ is constructed using the class dependency algorithm. Trivially if $C \in \mathsf{icl}(e) \cup \mathsf{icl}(\sigma')$ then $C \in \mathsf{dom}(\mathsf{CSig})$. So we apply Lemma 11 (2) to obtain $\vdash \texttt{CT}' : \texttt{ok}$.

Applying TV-FROZEN to e, σ' and $\texttt{CT}'$, we derive $\Gamma' \vdash \lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \texttt{CT}') : T \to U$ where Γ' is a subset of Γ such that $u_i \notin \mathsf{dom}(\Gamma')$. Then we apply Lemma 37 (weakening) to obtain $\Gamma \vdash \lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \texttt{CT}') : T \to U$ as required. Since σ and $\texttt{CT}$ are unchanged, this completes the case.

Case 16 (DEFROST)**.** Assume

$$\begin{array}{r} \texttt{defrost}(v;\ \lambda(T\ x).(\nu\,\vec{u})(m, e, \sigma', \texttt{CT}')), \sigma, \texttt{CT} \\ \longrightarrow_l (\nu\, x\vec{u})(\texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in sandbox}\ \{e[\vec{C^m}/\vec{C}]\}, \\ \sigma \cup \sigma' \cdot [x \mapsto v], \texttt{CT} \cup \texttt{CT}') \end{array} \quad \text{(b)}$$

and $\Gamma \vdash \texttt{defrost}(v;\ \lambda(T\ x).(\nu\,\vec{u})(m, e, \sigma', \texttt{CT}')) : U$. To derive this, the last typing rule applied was TE-DEFROST with premises

$$\Gamma \vdash v : T' \text{ with } T' <: T \text{ and } \Gamma \vdash \lambda(T\ x).(\nu\,\vec{u})(m, e, \sigma', \texttt{CT}') : T \to U \quad \text{(a)}$$

We shall prove

$$\Gamma, x : T, \vec{u} : \vec{T} \vdash \texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in sandbox}\ \{e[\vec{C^m}/\vec{C}]\} : U \quad \text{(b)}$$
$$\Gamma, x : T, \vec{u} : \Delta \vdash \sigma \cup \sigma' \cdot [x \mapsto v] : \texttt{ok} \quad \text{(c)}$$
$$\vdash \texttt{CT} \cup \texttt{CT}' : \texttt{ok} \quad \text{(d)}$$

To infer (a), TV-FROZEN was applied with the premises

$$\Gamma, x : T, \vec{u} : \vec{T} \vdash e : U \quad \Gamma, \vec{u} : \vec{T} \vdash \sigma' : \texttt{ok} \quad \vdash \texttt{CT}' : \texttt{ok}$$

Since C^m and C are typed by the same rule, we infer that

$$\Gamma, x : T, \vec{u} : \vec{T} \vdash e[\vec{C^m}/\vec{C}] : U$$

By application of TE-SANDBOX we have $\Gamma, x : T, \vec{u} : \vec{T} \vdash \texttt{sandbox}\ \{e[\vec{C^m}/\vec{C}]\} : U$.

By the premise of DEFROST, $\{\vec{F}\} = \mathsf{icl}(\sigma') \setminus \mathsf{dom}(\texttt{CT}')$, and in order to judge σ', it must be the case that for all F_i, there exists some mapping $o : F_i$ in $\Gamma, \vec{u} : \vec{T}$. Then by Lemma 37 we have that $\Gamma, \vec{u} : \vec{T} \vdash \texttt{Env}$. To infer this, we must have used the rules for well-formedness of environments (Figure 14), and so can deduce that $\vdash F_i : \texttt{tp}$ for all F_i. Taking this fact, we can apply TE-CLASSLOAD to obtain (b). Now to show that the two stores are compatible, we apply Lemma 38 (7) to derive $\Gamma, \vec{u} : \vec{T} \vdash \sigma \cup \sigma' : \texttt{ok}$. Then by application of Lemma 38 (1) we have (c). (d) then follows from $\mathsf{Inv}(3)$ to complete this case.

(2) Assume $\Gamma; \Delta \vdash F : \texttt{conf}$, $F \longrightarrow_l F'$ and $F' \not\models \mathsf{Err}$. Then we have $\Gamma; \Delta \vdash F' : \texttt{conf}$. To type a configuration, we apply TC-CONF which has the premises $\Gamma; \Delta_1 \vdash P : \texttt{thread}$, $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$, $\vdash \texttt{CT} : \texttt{ok}$, $\texttt{FCT} \subseteq \texttt{CT}$, $\Delta_1 \asymp \Delta_2$ where $\Delta = \Delta_1 \odot \Delta_2$. Proofs proceed from this point, and we omit the store σ and class table $\texttt{CT}$ when they do not change.

Case 17 (LEAVESANDBOX)**.** Straightforward by the inductive hypothesis.

Case 18 (MethLocal)**.** Assume $\Gamma; \Delta_1 \vdash E[o.m(v)] \mid P : \texttt{thread}$ and that $E[o.m(v)] \mid P \longrightarrow_l (\nu\, c)(E[\texttt{await}\ c] \mid o.m(v)\ \texttt{with}\ c \mid P)$ To type this, we applied rule TT-Par with the premises

$$\Gamma; \Delta_{11} \vdash E[o.m(v)] : \texttt{thread} \qquad \Gamma; \Delta_{12} \vdash P : \texttt{thread} \qquad \text{with } \Delta_{11} \asymp \Delta_{12} \tag{a}$$

To type (a), we applied Lemma 12(3) with premises

$$\Gamma \vdash o.m(v) : U \qquad \Gamma; \Delta_{11} \vdash E[\ ]^U : \texttt{thread} \tag{b}$$

Pick a fresh channel c, then apply TT-MethWith to (a) to obtain

$$\Gamma; \texttt{chan0}(U) \vdash o.m(v)\ \texttt{with}\ c : \texttt{thread}$$

With the same fresh channel, apply TT-Await to (b) to obtain

$$\Gamma; \Delta_{11}, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c] : \texttt{thread}$$

By Definition 3, we have $\Delta_{11}, c : \texttt{chanI}(U) \asymp c : \texttt{chan0}(U)$ with

$\Delta_{11}, c : \texttt{chanI}(U) \odot c : \texttt{chan0}(U) = \Delta_{11}, c : \texttt{chan}$. Since c was chosen fresh, $c \notin \mathsf{dom}(\Delta_{12})$ therefore $\Delta_{11}, c : \texttt{chan} \asymp \Delta_{12}$ and we can apply TT-Par twice to get

$$\Gamma; \Delta_{11}, c : \texttt{chan} \odot \Delta_{12} \vdash E[\texttt{await}\ c] \mid o.m(v)\ \texttt{with}\ c \mid P : \texttt{thread}$$

Then by permutation of the environment (Lemma 37) we have $\Delta_{11}, c : \texttt{chan} \odot \Delta_{12} = \Delta_1, c : \texttt{chan}$. Applying TT-Res yields:

$$\Gamma; \Delta_1 \vdash (\nu\, c)(E[\texttt{await}\ c] \mid o.m(v)\ \texttt{with}\ c \mid P) : \texttt{thread}$$

as required.

Case 19 (MethRemote)**.** Assume

$$E[o.m(v)] \mid P \longrightarrow_l (\nu\, c)(E[\texttt{await}\ c] \mid \texttt{go}\ o.m(\texttt{serialize}(v))\ \texttt{with}\ c \mid P)$$

with $o \notin \mathsf{dom}(\sigma)$ and $\Gamma; \Delta_1 \vdash E[o.m(v)] \mid P : \texttt{thread}$. This case is similar to the previous case up to (b). We shall proceed from this point. To type (b), we must have applied TE-Meth, with the premise that $\Gamma \vdash o : C$. By the side condition that $o \notin \mathsf{dom}(\sigma)$, $\mathsf{Inv}(8)$ and $\mathsf{Inv}(6)$, we have $\mathsf{RMI}(C)$. Therefore we can apply TT-GoSer to derive:

$$\Gamma; c : \texttt{chan0}(U) \vdash \texttt{go}\ o.m(\texttt{serialize}(v))\ \texttt{with}\ c : \texttt{thread}$$

Proof then proceeds from this point as in the case for MethLocal to derive $\Gamma; \Delta_1 \vdash (\nu\, c)(E[\texttt{await}\ c] \mid \texttt{go}\ o.m(\texttt{serialize}(v))\ \texttt{with}\ c \mid P) : \texttt{thread}$, as required.

Case 20 (MethInvoke)**.** Assume

$$o.m(v)\ \texttt{with}\ c, \sigma \longrightarrow_l (\nu\, x)(e[o, \texttt{return}(c)/\texttt{this}, \texttt{return}], \sigma \cdot [x \mapsto v])$$

and $\Gamma; \Delta_1 \odot \Delta_2 \vdash o.m(v)\ \texttt{with}\ c, \sigma : \texttt{conf}$. To infer this, we applied TC-Conf with premises

$$\Gamma; \Delta_1 \vdash o.m(v)\ \texttt{with}\ c : \texttt{thread} \qquad \Gamma; \Delta_2 \vdash \sigma : \texttt{ok} \qquad \Delta_1 \asymp \Delta_2$$

By premises of TT-MethWith, we have $\Gamma \vdash o.m(v) : U$ and $\Delta_1 = c : \texttt{chan0}(U)$. By application of MethInvoke in the reduction step, and the fact that to infer the above, we had to apply TE-Meth we have

$$\begin{array}{l} \sigma(o) = (C, \ldots) \qquad \mathsf{mbody}(m, C, \texttt{CT}) = (x, e) \qquad \mathsf{mtype}(m, C) = T \to U \\ \Gamma \vdash o : C \qquad \Gamma \vdash v : T' \qquad T' <: T \end{array} \quad \text{(a)}$$

By assumption that $\vdash \texttt{CT} : \texttt{ok}$, we can apply Lemma 39 to (a) to obtain that $x : T, \texttt{this} : C \vdash e : \texttt{ret}(U')$ with $U' <: U$ for a freshly chosen x. By application of Lemma 12 (substitution) followed by Lemma 37 (strengthening) we have $\Gamma, x : T \vdash e[o/\texttt{this}] : \texttt{ret}(U')$. Then by applying TT-Return we have

$$\Gamma, x : T; c : \texttt{chan0}(U) \vdash e[o, \texttt{return}(c)/\texttt{this}, \texttt{return}] : \texttt{thread}$$

By application of Lemma 38 (1) we then have that $\Gamma, x : T \vdash \sigma \cdot [x \mapsto v] : \texttt{ok}$. To complete the case we then apply TC-Conf followed by TC-ResId giving

$$\Gamma; \Delta_1 \odot \Delta_2 \vdash (\nu\, x)(e[o, \texttt{return}(c)/\texttt{this}, \texttt{return}], \sigma \cdot [x \mapsto v]) : \texttt{conf}$$

as required.

Case 21 (Await)**.** Assume $E[\texttt{await}\ c] \mid \texttt{return}(c)\ v \longrightarrow_l E[v]$ and $\Gamma; \Delta_1 \vdash E[\texttt{await}\ c] \mid \texttt{return}(c)\ v : \texttt{thread}$. To type this, we applied TT-Par with premises:

$$\Gamma; \Delta_{11} \vdash E[\texttt{await}\ c] : \texttt{thread} \qquad \Gamma; \Delta_{12} \vdash \texttt{return}(c)\ v : \texttt{thread} \qquad \Delta_{11} \asymp \Delta_{12} \quad \text{(a)}$$

To type the first conjunct of (a), we must have applied TT-Await. To type the second conjunct, we applied TT-Return. These give us that

$$\begin{array}{ll} \Gamma; \Delta'_{11} \vdash E[\ ]^U : \texttt{thread} & \text{with } \Delta_{11} = \Delta'_{11}, c : \texttt{chanI}(U) \\ \Gamma \vdash \texttt{return}\ v : \texttt{ret}(U') & \text{with } U' <: U \end{array} \quad \text{(b)}$$

To derive (b), we applied TE-Return with the premise that $\Gamma \vdash v : U'$. By Lemma 12(3) we obtain $\Gamma; \Delta'_{11} \vdash E[v] : \texttt{thread}$. To complete the case we apply Lemma 37 (weakening) to the environment Δ'_{11} to give $\Gamma; \Delta_1 \vdash E[v] : \texttt{thread}$.

Case 22 (Fork)**.** Similar to the above case, using Lemma 12(3).

Case 23 (ThreadDeath)**.** Trivial.

Case 24 (Sync)**.** Assume $E[\texttt{sync}\ (o)\ \{e\}], \sigma \longrightarrow_l E[\texttt{insync}\ o\ \{e\}], \sigma'$ with $\Gamma; \Delta_1 \vdash E[\texttt{sync}\ (o)\ \{e\}] : \texttt{thread}$ and $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$.

To derive this, we applied Lemma 12(3) with premises

$$\Gamma; \Delta_1 \vdash E[\ ]^S \qquad \Gamma \vdash \texttt{sync}\ (o)\ \{e\} : S \tag{a}$$

To derive (a), we applied TE-Sync with premises $\Gamma \vdash o : C$ and $\Gamma \vdash e : S$. We can then apply TE-InSync to derive $\Gamma \vdash \texttt{insync}\ o\ \{e\} : S$. Showing the resulting thread is well-typed, $\Gamma; \Delta_1 \vdash E[\texttt{insync}\ o\ \{e\}] : \texttt{thread}$, is straightforward from this point, so all that remains is to show $\Gamma; \Delta_2 \vdash \sigma' : \texttt{ok}$. However this is straightforward, since in the reduction step only the lock count of a store entry is changed. As σ is well-formed, we see that it must be the case that lock counts cannot be set to a negative number, and so we can conclude the case.

Case 25 (Wait)**.** Assume $E[o.\texttt{wait}] \mid P, \sigma \longrightarrow_l (\nu\, c)(E[\texttt{waiting}(c)\ n] \mid P, \sigma')$ and $\Gamma; \Delta_1 \vdash E[o.\texttt{wait}] \mid P : \texttt{thread}$, $\Delta_1 \asymp \Delta_2$ and $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$. To derive this, TT-Par was applied with premises:

$$\Gamma; \Delta_{11} \vdash E[o.\texttt{wait}] : \texttt{thread} \qquad \Gamma; \Delta_{12} \vdash P : \texttt{thread} \quad \text{and} \quad \Delta_{11} \asymp \Delta_{12} \tag{a}$$

To derive the left conjunct of (a), we applied Lemma 12(3) with the premise:

$$\Gamma; \Delta_{11} \vdash E[\ ]^{\texttt{void}} : \texttt{thread} \quad \text{and} \quad \Gamma \vdash o.\texttt{wait} : \texttt{void}$$

By the premise of the reduction rule Wait, we have

$$\mathsf{insync}(o, E) \quad \mathsf{lockcount}(\sigma, o) = n \quad \mathsf{setcount}(\sigma, o, 0) = \sigma'' \quad \mathsf{block}(\sigma'', o, c) = \sigma'$$

By Lemma 10, we have that $n > 0$, and since c is fresh we can apply TT-Waiting to obtain

$$\Gamma; \Delta_{11}, c : \texttt{chanI}(\texttt{void}) \vdash E[\texttt{waiting}(c)\ n]^{\texttt{void}} : \texttt{thread} \tag{h}$$

Since c is fresh, then by Definition 3, $\Delta_{11}, c : \texttt{chanI}(\texttt{void}) \asymp \Delta_{12}$. We then apply TT-Par to yield $\Gamma; \Delta_1, c : \texttt{chanI}(\texttt{void}) \vdash E[\texttt{waiting}(c)\ n] \mid P : \texttt{thread}$. By Lemma 38 (9), $\Gamma; \Delta_2, c : \texttt{chanO}(\texttt{void}) \vdash \sigma' : \texttt{ok}$. By Definition 3 followed

by TC-ResC we can derive $\Gamma; \Delta_1 \odot \Delta_2 \vdash (\nu\, c)(E[\texttt{waiting}(c)\ n] \mid P, \sigma') : \texttt{conf}$, as required.

Case 26 (Notify)**.** Assume

$$E[o.\texttt{notify}] \mid E_1[\texttt{waiting}(c)\ n], \sigma \longrightarrow_l E[\epsilon] \mid E_1[\texttt{ready}\ o\ n], \sigma'$$
$$\Gamma; \Delta_1 \odot \Delta_2 \vdash E[o.\texttt{notify}] \mid E_1[\texttt{waiting}(c)\ n], \sigma : \texttt{conf} \qquad \Delta_1 \asymp \Delta_2$$

To derive the above, we applied TC-Conf with premises

$$\Gamma; \Delta_{11} \odot \Delta_{12} \vdash E[o.\texttt{notify}] \mid E_1[\texttt{waiting}(c)\ n] : \texttt{thread} \qquad \Delta_1 = \Delta_{11} \odot \Delta_{12} \tag{a}$$
$$\Gamma; \Delta_2 \vdash \sigma : \texttt{ok} \tag{b}$$

To derive (a) we applied TT-Par with premises

$$\text{(i) } \Gamma; \Delta_{11} \vdash E[o.\texttt{notify}] : \texttt{thread} \qquad \text{(ii) } \Gamma; \Delta_{12} \vdash E_1[\texttt{waiting}(c)\ n] : \texttt{thread} \tag{c}$$

To derive (c-i), one of two possible rules was applied. Either E contains a return statement, or E is a forked thread. We show the case of the former (the latter is similar). Here the typing rule applied was TT-Return and we have that $\Delta_{11} = d : \texttt{chan0}(U)$ with the premise that $\Gamma \vdash E'[o.\texttt{notify}] : \texttt{ret}(U')$ with $U' <: U$ (E' is the context prior to the substitution of return statements). Then we see that we applied Lemma 12(3) with the premise that $\Gamma \vdash o.\texttt{notify} : \texttt{void}$ and $\Gamma \vdash E'[\ ]^{\texttt{void}} : \texttt{ret}(U')$. Then, to derive $\Gamma \vdash o.\texttt{notify} : C$ we applied TE-Monitor with premise $\Gamma \vdash o : C$. To derive (c-ii) we applied TT-Waiting with premises:

$$\Gamma; \Delta'_{12} \vdash E_1[\ ]^{\texttt{void}} : \texttt{thread}, \quad c \notin \mathsf{dom}(\Delta'_{12}) \quad n > 0 \quad \Delta_{12} = \Delta'_{12}, c : \texttt{chanI}(\texttt{void}) \tag{d}$$

Since $\Gamma \vdash \epsilon : \texttt{void}$, we can safely conclude that $\Gamma; \Delta_{11} \vdash E[\epsilon] : \texttt{ret}(U')$. Then since $\Gamma \vdash o : C$ and $n > 0$ we can apply TE-Ready to deduce $\Gamma \vdash \texttt{ready}\ o\ n : \texttt{void}$. Then taking this fact and (d), we can fill the whole in context E_1 to obtain

$$\Gamma; \Delta'_{12} \vdash E_1[\texttt{ready}\ o\ n] : \texttt{thread}$$

Now by the premise of the reduction rule Notify, we have $c \in \mathsf{blocked}(\sigma, o)$, therefore by typability of (b), we have that $c : \texttt{chan0}(\texttt{void}) \in \Delta_2$. Since $\Delta_1 \asymp \Delta_2$ we cannot have another output on channel c in Δ_{11}, therefore we can safely say that $\Delta_{11} \asymp \Delta'_{12}$, and then apply TT-Par to obtain

$$\Gamma; \Delta_{11} \odot \Delta'_{12} \vdash E[\epsilon] \mid E_1[\texttt{ready}\ o\ n] : \texttt{thread}$$

Now we must show that the new store, σ' is safe. Taking $\Delta_2 = \Delta_2', c : \texttt{chan0}(\texttt{void})$ we have by Lemma 37 that $\Gamma; \Delta_2 \vdash \texttt{Env}$, and so $c \notin \mathsf{dom}(\Delta_2')$. By premise of the reduction rule, we have that $\sigma' = \mathsf{unblock}(\sigma, o, c)$, and so by applying Lemma 38 (10) it must be the case that $\Gamma; \Delta_2' \vdash \sigma' : \texttt{ok}$. Trivially we have $(\Delta_{11} \odot \Delta_{12}') \asymp \Delta_2'$, and can apply TC-Conf to yield (where $\Delta_1' = \Delta_{11}' \odot \Delta_{12}'$)

$$\Gamma; \Delta_1' \odot \Delta_2' \vdash E[\epsilon] \mid E_1[\texttt{ready}\ o\ n], \sigma' : \texttt{conf}$$

Finally we apply TC-Weak to add the channel c that was removed to obtain $\Gamma; \Delta_1 \odot \Delta_2 \vdash E[\epsilon] \mid E_1[\texttt{ready}\ o\ n], \sigma' : \texttt{conf}$.

Case 27 (NotifyAll)**.** Similar to the case for Notify.

Case 28 (NotifyNone)**.** Straightforward.

Case 29 (LeaveCritical)**.** We shall consider the case for return, as the other case is similar. Assume $\texttt{insync}\ o\ \{\texttt{return}(c)\ v\}, \sigma \longrightarrow_l \texttt{return}(c)\ v, \sigma'$ with $\Gamma; c, \texttt{chan0}(U) \vdash \texttt{insync}\ o\ \{\texttt{return}(c)\ v\} : \texttt{thread}$ and $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$. To derive the first judgement, we applied TT-Return with the premise that $\Gamma \vdash \texttt{insync}\ o\ \{\texttt{return}\ v\} : \texttt{ret}(U')$ with $U' <: U$. Therefore we conclude TE-InSync was applied with premises $\Gamma \vdash o : C$ and $\Gamma \vdash \texttt{return}\ v : \texttt{ret}(U')$. Then by applying TT-Return we deduce $\Gamma; \texttt{chan0}(U) \vdash \texttt{return}(c)\ v : \texttt{thread}$, as required. By premise of LeaveCritical we have $\mathsf{lockcount}(\sigma, o) = n$ and $\mathsf{setcount}(\sigma, o, n-1) = \sigma'$, and by the assumption of σ, we have that $n \geq 0$. By Lemma 10 we have that $n \neq 0$ i.e. $n > 0$. When creating σ', we know that the new lock count can not be negative, therefore we have $\Gamma; \Delta_2 \vdash \sigma' : \texttt{ok}$ as required.

Case 30 (RC-Par)**.** Assume $P_1 \mid P_2, \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(P_1' \mid P_2, \sigma', \texttt{CT}')$ with $\vec{u} \notin \mathsf{fnv}(P_2)$ and $\Gamma; \Delta_1 \vdash P_1 \mid P_2 : \texttt{thread}$ with $\Delta_1 \asymp \Delta_2$, $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$ and $\vdash \texttt{CT} : \texttt{ok}$. To derive $P_1 \mid P_2$, TT-Par was applied with premises

$$\Gamma; \Delta_{11} \vdash P_1 : \texttt{thread},\ \Gamma; \Delta_{12} \vdash P_2 : \texttt{thread},\ \Delta_{11} \asymp \Delta_{12}\ \Delta_{11} \odot \Delta_{12} = \Delta_1 \quad \text{(a)}$$

By the premise of RC-Par we have that $P_1, \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(P_1', \sigma', \texttt{CT}')$. Since $\Delta_{11} \asymp \Delta_{12}$, $(\Delta_{11} \odot \Delta_{12}) \asymp \Delta_2$ then by Lemma 40, $\Delta_{11} \asymp \Delta_2$. So applying the inductive hypothesis to P_1 we obtain $\Gamma; \Delta_{11} \odot \Delta_2 \vdash (\nu\, \vec{u})(P_1', \sigma', \texttt{CT}') : \texttt{conf}$. By the fact that $\vec{u} \notin \mathsf{fnv}(P_2)$, we can apply weakening to (a), ensuring that thread P_2 is well typed in the new environment. This allows us to conclude $\Gamma; \Delta_1 \odot \Delta_2 \vdash (\nu\, \vec{u})(P_1' \mid P_2, \sigma', \texttt{CT}') : \texttt{conf}$, as required.

Case 31 (RC-Str)**.** Straightforward by Lemma 6.

Case 32 (RC-Res)**.** We shall prove the case when u is a channel name. The others are similar. Assume $(\nu\, c\vec{u})(P, \sigma, \texttt{CT}) \longrightarrow_l (\nu\, c\vec{u}')(P', \sigma', \texttt{CT}')$ and $\Gamma; \Delta \vdash (\nu\, c\vec{u})(P, \sigma, \texttt{CT}) : \texttt{conf}$. There are two cases: the last applied typing

rule was TC-Weak, or it was TC-ResC. In the case of the former, this rule has the premises

$\Gamma; \Delta' \vdash (\nu\, \vec{u})(P, \sigma, \texttt{CT}) : \texttt{conf}$ with $\Delta = \Delta', c : \texttt{chan}$. Then by the inductive hypothesis, $\Gamma; \Delta' \vdash (\nu\, \vec{u}')(P', \sigma', \texttt{CT}') : \texttt{conf}$. Applying TC-Weak we have $\Gamma; \Delta \vdash (\nu\, c\vec{u}')(P', \sigma', \texttt{CT}') : \texttt{conf}$ as required. When the last reduction rule was TC-ResC, we have the premises: $\Gamma; \Delta, c : \texttt{chan} \vdash (\nu\, \vec{u})(P, \sigma, \texttt{CT}) : \texttt{conf}$. Then again by the inductive hypothesis, $\Gamma; \Delta, c : \texttt{chan} \vdash (\nu\, \vec{u})(P, \sigma, \texttt{CT}) : \texttt{conf}$ and we can apply TC-ResC to conclude the required result.

(3)

Case 33 (RN-Conf)**.** By the premises of RN-Conf, $F \longrightarrow_l F'$. From the structure of N, we see that the last typing rule applied must have been TN-Conf with premise $\Gamma; \Delta \vdash F : \texttt{conf}$. Given this and the assumption that $F \longrightarrow_l F'$ we can apply Theorem 13 (1) to obtain $\Gamma; \Delta \vdash F' : \texttt{conf}$. We can then re-apply TN-Conf to deduce $\Gamma; \Delta \vdash l[F'] : \texttt{net}$ as required.

Case 34 (Download)**.** Assume:

$$\begin{array}{rl} & l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] \\ \longrightarrow & l_1[E[\texttt{resolve}\ \vec{D}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1 \cup \texttt{CT}'] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] \end{array}$$

and $\Gamma; \Delta \vdash l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$. To derive this, we applied TN-Par with premises:

$$\begin{array}{ll} \Gamma; \Delta_1 \vdash l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1] : \texttt{net} \quad \Delta_1 \asymp \Delta_2 & \text{(a)} \\ \Gamma; \Delta_2 \vdash l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net} & \end{array}$$

To type the network location in (a), we had to apply TE-ClassLoad at some point, with the premises that $\Gamma \vdash e : U$ and $\vdash \vec{C} : \texttt{tp}$. Thus we can apply TE-ClassLoad again to derive that $\Gamma \vdash \texttt{resolve}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e : U$ as required. Now all that remains is to show that $\vdash \texttt{CT}_1 \cup \texttt{CT}' : \texttt{ok}$. Again by inspecting the premises of Download, we see that $\texttt{CT}'$ is a subset of $\texttt{CT}_2$ with some substitutions applied. Since these do not affect well-formedness, we deduce that $\vdash \texttt{CT}' : \texttt{ok}$. By $\mathsf{Inv}(3)$ we see that if $\mathsf{dom}(\texttt{CT}_1) \cap \mathsf{dom}(\texttt{CT}') \neq \emptyset$, any shared classes will have the same definition. This means we can immediately derive $\vdash \texttt{CT}_1 \cup \texttt{CT}' : \texttt{ok}$. After this, to complete the case there is merely the mechanical rebuilding of the derivation of the following required result:

$$\Gamma; \Delta \vdash l_1[E[\texttt{resolve}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1 \cup \texttt{CT}'] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$$

Case 35 (Leave)**.** Assume $l_1[\texttt{go}\ o.m(v)\ \texttt{with}\ c \mid P_1, \sigma_1, \texttt{CT}_1] \mid l_2[P_2, \sigma_2, \texttt{CT}_2]$ $\longrightarrow l_1[P_1, \sigma_1, \texttt{CT}_1] \mid l_2[o.m(\texttt{deserialize}(v))\ \texttt{with}\ c \mid P_2, \sigma_2, \texttt{CT}_2]$ and $\Gamma; \Delta \vdash l_1[\texttt{go}\ o.m(v)\ \texttt{with}\ c \mid P_1, \sigma_1, \texttt{CT}_1] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$. In this derivation, we had to judge $\Gamma; c : \texttt{chan0}(U) \vdash \texttt{go}\ o.m(v)\ \texttt{with}\ c : \texttt{thread}$. This is typed

by TT-DeserWith, as is $o.m(\texttt{deserialize}(v))$ $\texttt{with}$ c. Therefore all that remains is to show that the channel environments can be safely composed, however this is straightforward by Lemma 40 and noting that the operator $\odot$ and predicate $\asymp$ are commutative. Hence we obtain

$$\Gamma; \Delta \vdash l_1[P_1, \sigma_1, \texttt{CT}_1] \mid l_2[o.m(\texttt{deserialize}(v)) \texttt{ with } c \mid P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$$

as required.

Case 36 (Return)**.** Similar to case Leave.

Case 37 (SerReturn)**.** Assume

$$l[\texttt{return}(c)\ v \mid P, \sigma, \texttt{CT}] \longrightarrow l[\texttt{go serialize}(v) \texttt{ to } c \mid P, \sigma, \texttt{CT}]$$

and $\Gamma; \Delta \vdash l[\texttt{return}(c)\ v \mid P, \sigma, \texttt{CT}] : \texttt{net}$. To type this, we applied TN-Conf with premise $\Gamma; \Delta \vdash l[\ldots] : \texttt{conf}$. To type this, we applied, TC-Conf. This has the following premises (we omit stores and class tables, since they are invariant under this reduction and therefore trivially well-typed)

$$\begin{array}{l} \Gamma; \Delta_{11} \vdash \texttt{return}(c)\ v : \texttt{thread} \qquad \Delta = \Delta_1 \odot \Delta_2 \qquad \Delta_1 = \Delta_{11} \odot \Delta_{12} \quad \text{(a)} \\ \Gamma; \Delta_{12} \vdash P : \texttt{thread} \qquad \Delta_{11} \asymp \Delta_{12} \end{array}$$

To infer (a), we applied TT-Return with premise

$$\Gamma \vdash \texttt{return}\ v : \texttt{ret}(U') \qquad U' <: U \qquad \Delta_{11} = c : \texttt{chan0}(U) \qquad \text{(b)}$$

To type (b), we applied TE-Return, with the premise that $\Gamma \vdash v : U'$. Then by applying TT-ValTo we have $\Gamma, c : \texttt{chan0}(U) \vdash \texttt{go serialize}(v) \texttt{ to } c : \texttt{thread}$. To complete the case we rebuild the network by applying TC-Conf and TN-Par and obtain $\Gamma; \Delta \vdash l[\texttt{go serialize}(v) \texttt{ to } c \mid P, \sigma, \texttt{CT}] : \texttt{net}$.

Case 38 (RN-Par)**.** Straightforward by the inductive hypothesis.

Case 39 (RN-Res)**.** We consider the case when the restricted name is a channel. Assume $(\nu\, c)N \longrightarrow (\nu\, c)N'$ and $\Gamma; \Delta \vdash (\nu\, c)N : \texttt{net}$. To derive this, we applied TN-ResC with the premise $\Gamma; \Delta, c : \texttt{chan} \vdash N : \texttt{net}$. By premise of RN-Res, $N \longrightarrow N'$ and so by the inductive hypothesis we have that $\Gamma; \Delta, c : \texttt{chan} \vdash N' : \texttt{net}$. Then applying TN-ResC we obtain $\Gamma; \Delta \vdash (\nu\, c)N' : \texttt{net}$ as required.

Case 40 (RN-Str)**.** Straightforward using Lemma 6.

G Code mobility transformations

This appendix contains several auxiliary definitions required to prove the closure of the code mobility transformation rule, (MOB), and its counterpart (AWAIT).

G.1 Extended typing rules for mobile code

In order to ensure that a piece of code can be safely relocated in the network, we must guarantee that any methods it invokes on other objects do not perform operations that could compromise this ability. To do this we use an extended version of the syntax given in § 3, and of the typing system presented in § 5.

We first introduce arrow types which include a *safety annotation*. We write $T \xrightarrow{\mathsf{safe}} U$ for an arrow type assigned to code that is safe to relocate in the network at runtime. Then well-formedness for these new arrow types are given by adding a new rule:

$$\frac{\vdash T : \mathtt{tp} \vee T \in \mathsf{dom}(\mathsf{CSig}) \wedge \neg\mathsf{RMI}(T) \quad \vdash U : \mathtt{tp} \vee U \in \mathsf{dom}(\mathsf{CSig}) \wedge \neg\mathsf{RMI}(U)}{\vdash T \xrightarrow{\mathsf{safe}} U : \mathtt{tp}}$$

For type soundness, we require that this safety property is inherited by subtypes. This is defined by adding the following new rules to the existing subtyping relation for arrow types.

$$\frac{T' <: T \qquad U <: U'}{T \xrightarrow{\mathsf{safe}} U <: T' \longrightarrow U'} \qquad \frac{T' <: T \qquad U <: U'}{T \xrightarrow{\mathsf{safe}} U <: T' \xrightarrow{\mathsf{safe}} U'}$$

To ensure that methods called by our mobile code can have their code safely relocated in the network, such method bodies should be assigned a "safe" arrow type introduced above. This requires modification of the well-formedness rule for methods in class tables as follows:

$$\frac{\begin{array}{c}\mathsf{mtype}(m, C) = T \longrightarrow U \implies \mathtt{this} : C, x : T \vdash^{\mathsf{s}} e : \mathtt{ret}(U') \text{ and } U' <: U \\ \mathsf{mtype}(m, C) = T \xrightarrow{\mathsf{safe}} U \implies x : T \vdash^{\mathsf{s}} e : \mathtt{ret}(U') \text{ and } U' <: U\end{array}}{\mathtt{this} : C \vdash^{\mathsf{s}} U\ m(T\ x)\{e\} : \mathtt{ok\ in}\ C}$$

Here the turnstile $\vdash^{\mathsf{s}}$ indicates that the typing derivation is made using the modified rules below. The jugement of $\vdash^{\mathsf{s}}$ is defined by the above rule and by other rules replacing $\vdash$ with $\vdash^{\mathsf{s}}$. Note that when a method body is "safe", it cannot refer to the receiver `this`. This prevents information from a location covertly leaking into mobile code.

If a piece of code satisfies the mobility predicate, it is safe to relocate around the network. This forms the core of our transformation rules, and in this appendix we give the full definition of that predicate.

Note that the mobility predicate does not include any synchronisation primitives or object instantiation expressions for simplicity of presentation; adding them is not difficult, but requires some unilluminating additional rules.

Values First we introduce the mobility predicate for values. Base values can be moved arbitrarily around the network, and do not leak object identifiers nor use any. An object identifier o can be safely moved around the network provided that it contains no references to remoteable objects and all object identifiers it references (i.e. $\mathsf{dom}(\sigma')$) are brought with it. It leaks no identifiers. Finally a closure can be moved around the network provided there exists some typing environment and store such that, when we evaluate the code e parametrised by formal parameter x in that store, the supplied actual parameter is itself mobile (the first conjunct). Secondly, a closure must contain all classes required by the objects held in its store component, as this avoids the need for a remote site to download classes.

$$
\begin{aligned}
\mathsf{Mobile}_{\Gamma,\sigma}(\mathtt{true}, \mathtt{false}, \mathtt{null}, (), \epsilon, \emptyset, \emptyset) &\text{ iff true} \\
\mathsf{Mobile}_{\Gamma,\sigma}(o, r, \emptyset) &\text{ iff } \sigma(o) = (C, \vec{f} : \vec{v}) \text{ and } \neg\mathsf{RMI}(C) \\
&\quad \text{and } \nexists o'.\mathsf{reachable}(\sigma, o, o') \\
&\qquad \text{with } \Gamma \vdash o' : D \text{ and } \mathsf{RMI}(D) \\
&\quad \text{and } r = \{o' \mid \mathsf{reachable}(\sigma, o, o')\} \cup \{o\} \\
&\quad \text{and } \forall o' \in r.\sigma(o') = (C', \vec{f} : \vec{v}) \implies \\
&\qquad (v_i \in r \text{ or } \mathsf{Mobile}_{\Gamma,\sigma}(v, \emptyset, \emptyset)) \\
\mathsf{Mobile}_{\Gamma,\sigma}(\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma_0, \mathtt{CT}), \emptyset, \emptyset) &\text{ iff } \exists \Gamma', \sigma'.\mathsf{Mobile}_{\Gamma',\sigma'}(x, r', \emptyset) \\
&\quad \text{and } \mathsf{Mobile}_{\Gamma',\sigma_0\cup\sigma'}(e, r, s) \\
&\quad \text{and } \mathsf{icl}(\sigma_0) \subseteq \mathsf{dom}(\mathtt{CT}) \text{ and } \mathsf{ctcomp}(\mathtt{CT})
\end{aligned}
$$

Homomorphic expressions Next we show the homomorphic mappings. The key is to ensure that evaluation of a term does not leak object identifiers used in the evaluation of subsequent terms, and we introduce the following

macro to that end.

$$\begin{aligned}&\mathsf{MobileH}_{\Gamma,\sigma}(e_0,\ldots,e_n,r_0,\ldots,r_n,s_0,\ldots,s_n)\\ \text{iff}\ \ &\textstyle\bigwedge_{0\le i\le n}\mathsf{Mobile}_{\Gamma,\sigma}(e_i,r_i,s_i)\quad\text{and}\quad s_0\cap\bigcup_{1\le j\le n}r_j=\emptyset\end{aligned}$$

Using this macro we can define the mappings for the homomorphic cases.

$$\begin{aligned}
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{if}\ (e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\},\textstyle\bigcup r_i,\bigcup s_i)\ &\text{iff}\ \mathsf{MobileH}_{\Gamma,\sigma}(e_0,e_1,e_2,r_0,r_1,r_2,s_0,s_1,s_2)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{while}\ (e_0)\ \{e_1\},\textstyle\bigcup r_i,\bigcup s_i)\ &\text{iff}\ \mathsf{MobileH}_{\Gamma,\sigma}(e_0,e_1,r_0,r_1,s_0,s_1)\\
\mathsf{Mobile}_{\Gamma,\sigma}(e_0;e_1,\textstyle\bigcup r_i,\bigcup s_i)\ &\text{iff}\ \mathsf{MobileH}_{\Gamma,\sigma}(e_0,e_1,r_0,r_1,s_0,s_1)\\
\mathsf{Mobile}_{\Gamma,\sigma}(T\ x=e_0;e_1,r,s)\ &\text{iff}\ \mathsf{Mobile}_{\Gamma,\sigma}(\texttt{defrost}(e_0;\ \texttt{freeze}[\texttt{eager}](T\ x)\{e_1\}),r,s)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{return}\ e,r,s)\ &\text{iff}\ \mathsf{Mobile}_{\Gamma,\sigma}(e,r,s)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{return},\emptyset,\emptyset)\ &\text{iff true}\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{defrost}(e_0;\ e_1),\textstyle\bigcup r_i,\bigcup s_i)\ &\text{iff}\ \mathsf{MobileH}_{\Gamma,\sigma}(e_0,e_1,r_0,r_1,s_0,s_1)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{fork}(e),r,s)\ &\text{iff}\ \mathsf{Mobile}_{\Gamma,\sigma}(e,r,s)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{download}\ \emptyset\ \texttt{from}\ l\ \texttt{in}\ e,r,s)\ &\text{iff}\ \mathsf{Mobile}_{\Gamma,\sigma}(e,r,s)\\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{sandbox}\ \{e\},r,s)\ &\text{iff}\ \mathsf{Mobile}_{\Gamma,\sigma}(e,r,s)
\end{aligned}$$

Other expressions Here we explain the mobility predicate for non-homomorphic expressions. Mobility of a variable x depends upon mobility of the value it contains, and similarly assignment $x = e$ has a similar condition, requiring mobility for both the variable x with its current value plus mobility of the expression whose result will be assigned.

The case for method call $o.m(e_1)$ demonstrates the importance of the two sets of identifiers r and s. First both expressions must be safe to move, and any identifiers leaked in e_0 must not be used in e_1 as expected. However if this method call is made to a potentially remote party then the set of identifiers leaked by the whole expression includes *all identifiers used by* e_1. If it is a local call then the identifiers leaked by the whole are the union of those leaked by the parts.

Finally a freeze expression is mobile iff there exists some enclosing store in which the resultant closure will be evaluated that ensures the actual parameter supplied for x is safe to move. Then the body of the closure must be mobile. We remove variable x from the current store σ as it is legal to have the program $T\ x = e; \texttt{freeze}[\texttt{eager}](T'\ x)\{e'\}; \cdots$, but the locally allocated variable should be treated as distinct from the formal parameter of the closure.

$$
\begin{aligned}
\mathsf{Mobile}_{\Gamma,\sigma}([\ ]^T, \emptyset, \emptyset) &\text{ iff } \neg\mathsf{RMI}(T) \\
\mathsf{Mobile}_{\Gamma,\sigma}(x, r \cup \{x\}, \emptyset) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(v, r, \emptyset) \\
\mathsf{Mobile}_{\Gamma,\sigma}(x = e, r \cup r', s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(e, r, s) \text{ and } \mathsf{Mobile}_{\Gamma,\sigma}(x, r', \emptyset) \\
\mathsf{Mobile}_{\Gamma,\sigma}(e.f, r, s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(e, r, s) \\
\mathsf{Mobile}_{\Gamma,\sigma}(e_0.f = e_1, \bigcup r_i, \bigcup s_i) &\text{ iff } \mathsf{MobileH}_{\Gamma,\sigma}(e_0, e_1, r_0, r_1, s_0, s_1) \\
\mathsf{Mobile}_{\Gamma,\sigma}(e_0.m(e_1), \bigcup r_i, \bigcup s_i) &\text{ iff } \mathsf{MobileH}_{\Gamma,\sigma}(e_0, e_1, r_0, r_1, s_0, s_1) \\
&\quad \text{and } \Gamma \vdash e_0 : C \text{ with } \mathsf{mtype}(m, C) = T \xrightarrow{\mathsf{safe}} U \\
&\quad \text{and } e_0 \neq o \\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{freeze}[\texttt{eager}](T\ x)\{e\}, r \setminus \{x\}, \emptyset) &\text{ iff } \exists \Gamma', \sigma'. \mathsf{Mobile}_{\Gamma',\sigma'}(x, r', \emptyset) \\
&\quad \text{and } \mathsf{Mobile}_{\Gamma',\sigma\setminus\{x\}\cup\sigma'}(e, r, s) \\
\mathsf{Mobile}_{\Gamma,\sigma}(o.m(e), r, s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(e, r', s') \text{ and } \Gamma \vdash o : C \\
&\quad \text{and } \mathsf{mtype}(m, C) = T \xrightarrow{\mathsf{safe}} T' \\
&\quad \text{and if } \mathsf{RMI}(C) \text{ then } s = r' \text{ and } r = r' \\
&\quad \text{else if } \neg\mathsf{RMI}(C) \text{ then } s = s' \text{ and } \mathsf{Mobile}_{\Gamma,\sigma}(o, r'', \emptyset) \text{ and } r = r' \cup r''
\end{aligned}
$$

Threads The mobility predicate is also defined over threads. The only interesting case if for parallel composition, and this merely says that neither thread leaks identifiers the other uses.

$$
\begin{aligned}
\mathsf{Mobile}_{\Gamma,\sigma}(\mathbf{0}, \emptyset, \emptyset) &\text{ iff true} \\
\mathsf{Mobile}_{\Gamma,\sigma}(P_0 \mid P_1, \bigcup r_i, \bigcup s_i) &\text{ iff } \bigwedge \mathsf{Mobile}_{\Gamma,\sigma}(P_i, r_i, s_i) \\
&\quad \text{and } s_0 \cap r_1 = \emptyset \text{ and } s_1 \cap r_0 = \emptyset \\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{forked}\ e, r, s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(e, r, s) \\
\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{return}(c)\ e, r, s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(e, r, s) \\
\mathsf{Mobile}_{\Gamma,\sigma}(E[\texttt{await}\ c] \mid o.m(v)\ \texttt{with}\ c, r, s) &\text{ iff } \mathsf{Mobile}_{\Gamma,\sigma}(E[o.m(v)], r, s) \\
&\quad \text{and } \Gamma \vdash o : C \text{ with } \neg\mathsf{RMI}(C)
\end{aligned}
$$

The following lemma establishes a useful property of the mobility predicate on expressions.

Lemma 41. *If* $\mathsf{Mobile}_{\Gamma,\sigma}(P, r, s)$ *then we have* $r = \{o \mid \mathsf{reachable}(\sigma, P, o)$ *and* $\Gamma \vdash o : C$ *with* $\neg\mathsf{RMI}(C)\} \cup \mathsf{fv}(P)$ *and* $s \subseteq r$.

Proof. By induction on the size of term P. The only non-trivial case is when $P = o$. Suppose $\mathsf{Mobile}_{\Gamma,\sigma}(o, \mathsf{dom}(\sigma'), \emptyset)$ with $\mathsf{og}(\sigma, o) = \sigma'$. By Lemma 11 (1a)

we have that all objects reachable from o in σ are reachable in σ'. Then because we have the premise that $\mathsf{fnv}(\sigma') \subseteq \mathsf{dom}(\sigma')$ we know that σ' contains no remoteable object identifiers, hence the items reachable in σ' exclude those identifiers, as required. As o has no free variables, this concludes the case. $\square$

G.2.1 Proof of Lemma 31

By induction on the length of the reduction sequence with case analysis of the final rule applied. Most cases are straightforward, so we illustrate only the key ones:

Case 41 (FldAss)**.** Suppose $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(o.f = v, r_0 \cup r_1, s_0 \cup s_1)$ with $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(o, r_0, \emptyset)$ and $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(v, r_1, \emptyset)$, and $o.f = v, \sigma_\alpha, \mathtt{CT} \longrightarrow_l v, \sigma'_\alpha, \mathtt{CT}$. We must show that v is mobile in the updated store σ'_α (σ_α with field f in object o updated to point to v).

If v is a closure or base value, we immediately have $\mathsf{Mobile}_{\Gamma,\sigma'_\alpha}(v, \emptyset, \emptyset)$ as required. If v is an object identifier then because it is mobile, using store locations r_1, the assignment $o.f = v$ cannot increase the size of r_1, hence because $\mathsf{dom}(\sigma_\alpha) \subseteq \mathsf{dom}(\sigma'_\alpha)$ $\mathsf{Mobile}_{\Gamma,\sigma'_\alpha}(v, r'_1, \emptyset)$ with $r'_1 \subseteq r_1$.

Case 42 (Cong)**.** Suppose $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(E[e], r_0 \cup r_1, s_0 \cup s_1)$ and $E[e], \sigma_\alpha, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(E[e'], \sigma'_\alpha, \mathtt{CT}')$. To derive the reduction, we applied Cong with the premise $e, \sigma_\alpha, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(e', \sigma'_\alpha, \mathtt{CT}')$ and $\vec{u} \notin \mathsf{fnv}(E)$. Suppose the mobility predicate is derived from the assumptions $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(E, r_1, s_1)$ and $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(e, r_0, s_0)$ with $s_0 \cap r_1 = \emptyset$. Then by the inductive hypothesis we have $\mathsf{Mobile}_{\Gamma,\sigma'_\alpha}(e', r'_0, s'_0)$, and we know that $s'_0 \subseteq s_0$ (this is easily proved by case analysis on reduction rules), hence $s'_0 \cap r_1 = \emptyset$. This means that, in principle the reduction of the sub-term e has not rendered the remainder of E immobile (since it has not leaked any objects used by E.

Note that r_1 corresponds to the set of all object identifiers reachable from E plus the names of all local variables in E.

Now because E and e are not necessarily disjoint in terms of the memory locations that they touch during execution, evaluation of e may mean that r'_1 (the memory footprint of E after executing e), could have changed. If it remains the same then trivially by applying weakening we have $\mathsf{Mobile}_{\Gamma,\vec{u}:\vec{T},\sigma'_\alpha}(E, r_1, s_1)$ and hence $\mathsf{Mobile}_{\Gamma,\sigma'_\alpha}(E[e'], r'_0 \cup r_1, s'_0 \cup s_1)$. If it shrinks (e.g. execution of e breaks a link from an object found in r_1 by setting a field to `null`), then clearly what was safe to move before is still safe to move now.

The case where it increases in size is slightly more complex, but still easy. Consider that the only way the memory footprint can increase is by field assignment (so an object with identifier in r_1 has a `null` field set to point to another

object). However by the inductive hypothesis, any such assigned object identifier must point to an object at the top of a *fully mobile* object graph, hence all object identifiers in r'_1 are safe to move, so $\mathsf{Mobile}_{\Gamma,\vec{u}:\vec{T},\sigma'_\alpha}(E[e'], r'_0 \cup r'_1, s'_0 \cup s'_1)$ as required.

Case 43 (FREEZE)**.** Assume

$$\mathsf{Mobile}_{\Gamma,\sigma}(\texttt{freeze}[t](T\ x)\{e\}, r \setminus \{x\}, \emptyset) \quad \text{(a)}$$
$$\texttt{freeze}[t](T\ x)\{e\}, \sigma_\alpha, \texttt{CT} \longrightarrow_l \lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma_0, \texttt{CT}'), \sigma_\alpha, \texttt{CT} \quad \text{(b)}$$

Assume (a) is derived from

$$\exists \Gamma', \sigma'.\mathsf{Mobile}_{\Gamma',\sigma'}(x, r', \emptyset) \quad \text{(a-1)}$$
$$\mathsf{Mobile}_{\Gamma',\sigma_\alpha \setminus \{x\} \cup \sigma'}(e, r, s) \quad \text{(a-2)}$$

We must show that if (a-2), then $\mathsf{Mobile}_{\Gamma',\sigma_0 \cup \sigma'}(e, r, s)$. However this is immediate by (b) and the correctness of og as σ_0 is computed as the minimum closure required to execute e. We can give a concrete typing environment for Γ' that just comprises the types of all mappings in σ_0 plus a mapping $x : T$ for the formal parameter. Hence there exist Γ', σ' such that $\mathsf{Mobile}_{\Gamma',\sigma'}(x, r', \emptyset)$ and $\mathsf{Mobile}_{\Gamma',\sigma_0 \cup \sigma'}(e, r, s)$. Thus by examination of the mobility predicate, we have: $\mathsf{Mobile}_{\Gamma,\sigma_\alpha}(\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma_0, \texttt{CT}'), \emptyset, \emptyset)$ as required.

H Mapping for Proposition 34 (2)

We define the encoding from DJ with the methods with multiple parameters into those with a single parameter. The mapping forms $[\![\cdot]\!]^{\Gamma}_{\vec{x},z}$ where $\vec{x}$ is multiple parameters of the source language, while z is a single parameter of the target one. Γ is an environment.

$$[\![\texttt{CT} \cdot [C \mapsto L]]\!] \stackrel{\text{def}}{=} [\![\texttt{CT}]\!] \cdot [C \mapsto L'] \cup \texttt{CT}' \text{ where } (L', \texttt{CT}') = [\![L]\!]$$
$$[\![\mathsf{CSig} \cdot C : \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T_i} \to U_i\}]\!]$$
$$\stackrel{\text{def}}{=} [\![\mathsf{CSig}]\!] \cdot C : \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : C_{mi} \to U_i\} \cdot C_{mi} : \vec{T_i}\vec{\texttt{f}_i}$$
$$[\![\texttt{class}\ C\ \texttt{extends}\ D\,\{\vec{T}\vec{f};\ K\,\vec{M}\}]\!]$$
$$\stackrel{\text{def}}{=} (\texttt{class}\ C\ \texttt{extends}\ D\,\{\vec{T}\vec{f};\ K\,\vec{M}'\}, \texttt{CT}) \text{ where } (\vec{M}', \texttt{CT}) = [\![\vec{M}]\!]_{\texttt{this}:C}$$
$$[\![U\,m\,(\vec{T}\vec{x})\{e\}]\!]_{\texttt{this}:C} \stackrel{\text{def}}{=} (U\,m\,(C_m\ z)\{e'\}, [C_m \mapsto \texttt{class}\ C_m\{\vec{T}\vec{f};\ K\ \}] \cup \texttt{CT})$$
$$\text{where } (e', \texttt{CT}) = [\![e]\!]^{\vec{x},z}_{\texttt{this}:C,\vec{x}:\vec{T}}$$
$$[\![\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})]\!] \stackrel{\text{def}}{=} \lambda(T\ x).(\nu\,\vec{u})(l, e', \sigma', \texttt{CT} \cup \texttt{CT}')$$
$$\text{where } (e', \texttt{CT}_0) = [\![e]\!]_{\vec{x}:\vec{T},\vec{u}:\vec{T'}}, (\sigma', \texttt{CT}_1) = [\![\sigma]\!]_{\vec{x}:\vec{T},\vec{u}:\vec{T'}}, \texttt{CT}' = \texttt{CT}_0 \cup \texttt{CT}_1$$

The encoding of other values is identical. Next we define the main rules for the expressions. Others are just homomorphic like $e_0; e_1$ below. We let

$(e'_i, \mathtt{CT}_i) = [\![e_i]\!]^{\vec{x},z}_{\Gamma}$ for $i \geq 0$ in the following.

$$[\![y]\!]^{x_1,\ldots,x_n,z}_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} (z.\mathtt{f_i}, \emptyset) & \text{if } y = x_i \\ (y, \emptyset) & \text{otherwise} \end{cases} \qquad [\![\mathtt{this}]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (\mathtt{this}, \emptyset)$$

$$[\![y := e_0]\!]^{x_1,\ldots,x_n,z}_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} (z.\mathtt{f_i} := e'_0, \mathtt{CT}_0) & \text{if } y = x_i \\ (y := e'_0, \mathtt{CT}_0) & \text{otherwise} \end{cases} \qquad [\![e_0; e_1]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (e'_0; e'_1, \bigcup \mathtt{CT}_i)$$

$$[\![e_0.m(e_1,\ldots,e_n)]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (e'_0.m(\mathtt{new}\ C_m(e'_1,\ldots,e'_n)), \bigcup \mathtt{CT}_i \cup [C_m \mapsto \mathtt{class}\ C_m\{\vec{T}\vec{f};\ K\}])$$
$$\text{where } \Gamma \vdash e_0 : C$$

The mapping of a configuration, threads and stores are defined as follows.

$$[\![(\nu\ \vec{o}\vec{c}) \prod l_i[(\nu\ \vec{x}_i)(P_i, \sigma_i, \mathtt{CT}_i)]]\!] \stackrel{\text{def}}{=} (\nu\ \vec{o}\vec{c}) \prod l_i[[\![(\nu\ \vec{x}_i)(P_i, \sigma_i, \mathtt{CT}_i)]\!]_{\vec{o}:\vec{C}}]$$

$$[\![(\nu\ \vec{x}\vec{x}_1 \ldots \vec{x}_n)(Q \mid \prod_j P_j, \sigma \cup \bigcup \sigma_j, \mathtt{CT}))]\!]_{\Gamma} \stackrel{\text{def}}{=} (\nu\ \vec{x}\vec{y}\vec{o})(Q' \mid \prod_j P'_j, \sigma' \cup \bigcup \sigma'_j, \mathtt{CT} \cup \mathtt{CT}')$$
$$\text{with } [\![Q]\!]_{\Gamma,\vec{x}:\vec{T}} = (Q', \mathtt{CT}_1) \quad [\![P_j]\!]^{\vec{x}_j,y_j}_{\Gamma,\vec{x}_j:\vec{T}_j} = (P'_j, \mathtt{CT}_j) \quad [\![\sigma]\!]_{\Gamma,\vec{x}:\vec{T}} = (\sigma', \mathtt{CT}_0)$$
$$[\![\sigma_j]\!]^{\vec{x}_j,o_j,m,C}_{\Gamma,\vec{x}_j:\vec{T}_j} = (\sigma'_j, \mathtt{CT}_{0j}) \quad \mathtt{CT}' = \bigcup \mathtt{CT}_{0j} \cup \bigcup \mathtt{CT}_j \cup \mathtt{CT}_1 \cup \mathtt{CT}_0$$

$$[\![\emptyset]\!]_{\Gamma} \stackrel{\text{def}}{=} [\![\emptyset]\!]^{\vec{x},y,o,m,C}_{\Gamma} \stackrel{\text{def}}{=} (\emptyset, \emptyset)$$

$$[\![\sigma \cdot [x \mapsto v]]\!]_{\Gamma} \stackrel{\text{def}}{=} (\sigma' \cdot [x \mapsto v'], \mathtt{CT}' \cup \mathtt{CT})$$

$$[\![\sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})]]\!]_{\Gamma} \stackrel{\text{def}}{=} (\sigma' \cdot [o \mapsto (C, \vec{f} : \vec{v}')], \mathtt{CT}' \cup \mathtt{CT})$$

$$[\![\sigma \cdot [\vec{x} \mapsto \vec{v}]]\!]^{\vec{x},y,o,m,C}_{\Gamma} \stackrel{\text{def}}{=} (\sigma' \cdot [y \mapsto o] \cdot [o \mapsto (C_m, \vec{f} : \vec{v}')], \mathtt{CT}' \cup \mathtt{CT})$$

$$[\![\sigma \cdot [o' \mapsto (C, \ldots)]]\!]^{\vec{x},y,o,m,C}_{\Gamma} \stackrel{\text{def}}{=} [\![\sigma]\!]^{\vec{x},y,o,m,C}_{\Gamma}$$

with $[\![v]\!]_{\Gamma} = (v', \mathtt{CT}')$ and $[\![\sigma]\!]_{\Gamma} = (\sigma', \mathtt{CT})$.

I Proofs of Theorem 36 (3)

This section proves that (RMI3) is equivalent to (Opt3) under the assumption there is no call-back. Let $e'_i = \mathtt{deserialize}(v_i)$ where $v_i = \lambda(\mathtt{unit}\ x).(\nu\ \vec{u})(l, a, \sigma_i)$ is a serialised mobile value at line i in (Opt3) $(3 \leq i \leq 5)$. We show the body of (Opt3) is equivalent with $\mathtt{return}\ r.\mathtt{run}(\mathtt{freeze}\{e[\vec{e}'/\vec{b}]; z\})$. With out loss of generality, we firstly simplify the program with two arguments as follows:

```
1 int m3(RemoteObject r, MyObj a) {
2   return r.g(a, r.f(a));
3 }
```

The corresponding optimised program is:

```
1 int mOpt3(RemoteObject r, MyObj a){
2   ser<MyObj> b1 = serialize(a);
3   ser<MyObj> b2 = serialize(a);
4   thunk<int> t = freeze {
5     r.g(deserialize(b2), r.f(deserialize(b1)));
6   };
7   return r.run(t);
8 }
```

First by Proposition 34 (1), we can ignore an effect of the class downloading. Since there is no call-back, we can also ignore an effect of the method invocation "`r.f`" to the timing of the serialisation "`b2 = serialize(a)`". Hence there is no interleaving between "`b1`" in Line 2 and "`b2`" in Line 3 from the location m. Thus we apply (NI) to the optimised code, and it is equated to:

$$P \stackrel{\text{def}}{=} \mathtt{return}(c)\ r.\mathtt{run}(\mathtt{freeze}\{r.\mathtt{g}(\mathtt{deserialize}(v_2), r.\mathtt{f}(\mathtt{deserialize}(v_1)))\})$$

where $v_i = \lambda(\mathtt{unit}\ x).(\nu\,\vec{u})(l, a, \sigma_i)$ is a serialised mobile value at b_i in (Opt3) above.

First we execute the original program. We omit a surrounding context where it is unnecessary. We also assume the location m (server) contains a store which includes $[r \mapsto (C, ..)]$ and a class table of class C which contains methods `f` and `g`. We also omit C and channel restriction of c and c_1.

$$
\begin{array}{lll}
 & l[\mathtt{return}(c)\ r.\mathtt{g}(a, r.\mathtt{f}(a))] \,|\, m[\mathbf{0}] & \\
\mapsto & l[\mathtt{return}(c)\ r.\mathtt{g}(a, \mathtt{await}\ c_1) \,|\, \mathtt{go}\ r.\mathtt{f}(\mathtt{serialize}(a))\ \mathtt{with}\ c_1] \,|\, m[\mathbf{0}] & (\text{NI}) \\
\mapsto & l[\mathtt{return}(c)\ r.\mathtt{g}(a, \mathtt{await}\ c_1) \,|\, \mathtt{go}\ r.\mathtt{f}(v_1)\ \mathtt{with}\ c_1] \,|\, m[\mathbf{0}] & (\text{FR}) \\
\mapsto^+ & l[\mathtt{return}(c)\ r.\mathtt{g}(a, \mathtt{await}\ c_1)] \,|\, m[r.\mathtt{f}(\mathtt{deserialize}(v_1))\ \mathtt{with}\ c_1] & (\text{NI}) \\
\mapsto^+ & l[...] \,|\, m[(\nu\,\vec{u})(r.\mathtt{f}(a)\ \mathtt{with}\ c_1, \sigma_1)] & (\text{NI}) \\
\rightarrow\!\!\!\!\rightarrow & l[...] \,|\, m[\mathtt{return}(c_1)\ n] & (\ddagger) \\
\mapsto^+ & l[\mathtt{return}(c)\ r.\mathtt{g}(a, n)] \,|\, m[\mathbf{0}] & (\text{MOB}),(\text{L2}),(\text{NI}) \\
\mapsto^+ & l[\mathtt{return}(c)\ \mathtt{await}\ c'] \,|\, m[r.\mathtt{g}(\mathtt{deserialize}(v_2), n)\ \mathtt{with}\ c'] & (\text{NI}),(\text{FR}),(\text{NI}) \\
\mapsto^+ & l[\mathtt{return}(c)\ \mathtt{await}\ c'] \,|\, m[(\nu\,\vec{u})(e[\mathtt{return}(c')/\mathtt{return}], \sigma_2)] & (\text{NI})
\end{array}
$$

At ($\ddagger$), we assume the method invocation `r.f` terminates and returns the answer n. If `r.f` does not terminate in `m3`, as seen in the following, `mOpt3` diverges, thus they are trivially equivalent. Now we look at execution of the

optimised code.

$$
\begin{array}{lll}
 & l[P] \,|\, m[\mathbf{0}] & \\
\mapsto & l[\texttt{return}(c)\ r.\texttt{run}(\lambda().(r.\texttt{g}(\texttt{deserialize}(v_2), r.\texttt{f}(\texttt{deserialize}(v_1))))))] \,|\, m[\mathbf{0}] & \\
\mapsto^{+} & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, (\nu\, \vec{u})(m[\texttt{return}(c')\ r.\texttt{g}(\texttt{deserialize}(v_2), r.\texttt{f}(a)), \sigma_1]) & \\
\rightarrow\!\!\!\rightarrow & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, m[\texttt{return}(c')\ r.\texttt{g}(\texttt{deserialize}(v_2), n)] & (\dagger) \\
\mapsto^{+} & l[\texttt{return}(c)\ \texttt{await}\ c'] \,| & (\textsc{ni}) \\
 & \quad m[(\nu\, \vec{u}c'')(\texttt{return}(c')\ \texttt{await}\ c'' \,|\, e[\texttt{return}(c'')/\texttt{return}], \sigma_2)] & \\
\mapsto & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, m[(\nu\, \vec{u}c'')(\texttt{return}(c')\ \texttt{sandbox}\ \{e[e'/\texttt{return}\ e']\}, \sigma_2)] & (\textsc{l}2) \\
\mapsto & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, m[(\nu\, \vec{u})(e[\texttt{return}(c')/\texttt{return}], \sigma_2)] & (\textsc{l}1)
\end{array}
$$

In the above, we know that we can get n at (†) if and only if n is obtained at (‡). Also the serialisation (FR) in the original code can be derived with the assumption such that v_1 and v_2 are serialised mobile values in Line 2 and Line 3 in the optimised code. Since $\mapsto \subseteq \cong$, (RMI3) is equivalent with (Opt3).