Formalising Java RMI with Explicit Code Mobility

Alexander Ahern and Nobuko Yoshida
Department of Computing, Imperial College London

This paper presents a Java-like core language with primitives for object-oriented distribution 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.

Categories and Subject Descriptors: D.3.1 [**Programming Language**]: Formal Definitions and Theory; D.3.3 [**Programming Language**]: Language Constructs and Features; F.3.2 [**Theory of Computation**]: Semantics of Programming Languages

General Terms: Languages,Theory

Additional Key Words and Phrases: 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. 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 such engineering practice 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 [Microsystems Inc. 2005] (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 further hide 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 from the sender. Such code mobility is important as it reduces the need for strong coupling between communicating parties, while preserving the type safety of the system as a whole.

The implicit code mobility in RMI allows almost transparent use of remote ob-

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

jects and services. However when rigorously analysing the dynamics of distributed programs, or when providing programmers with source-level control over code distribution [Christ 2000], it becomes essential to model their 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* [Bogle and Liskov 1994] or *aggregation* [Yeung and Kelly 2003; Yeung 2004], 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 distributed object-oriented core language with communication primitives (RMI) and distributed runtime. 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 [Milner et al. 1992; Sangiorgi 1992; Honda and Tokoro 1991]. One highlight is the use of a *linear* type discipline [Kobayashi et al. 1996; Honda 1996; Yoshida et al. 2001] 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 object graphs but also executable code. Code passing offers a programmer fine-grained control of type-safe code distribution, improving the safety and performance of their distributed applications. For example, a program fragment accessing a resource remotely could be frozen into 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 [Christ 2000; Bogle and Liskov 1994; Yeung and Kelly 2003; Yeung 2004].

As an application of our formalism, we show that the RMI *aggregation* optimisations proposed in [Yeung and Kelly 2003; Yeung 2004] are type- and semantics-preserving. The generality of the primitive 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. Similar optimisations naturally arise whenever latency and bandwidth are a limiting factor in the performance of distributed 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) A technique to systematically prove type safety for distributed networks using distributed invariants. Not only are they essential for proving type safety but also they are a useful analytical tool for developing consistent typing rules.

(3) Justification of several inter-node RMI optimisations employing explicit code

mobility, using a semantically sound syntactic transformation 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 safety and progress properties using the invariants. 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 a full version of [Ahern and Yoshida 2005b], with complete definitions and detailed proofs. The emphasis is on illustrating correctness of formalising RMI and a use of the distributed invariants for proving the type safety, and developing a theory of an observational congruence for a distributed Java. The present version 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 sequential languages, we can aim to improve execution times by removing redundancy and ensuring our programs 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 [Flanagan 2000] but which is generally applicable to various forms of remote communication, is *aggregation* [Bogle and Liskov 1994; Yeung and Kelly 2003; Yeung 2004]. 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). `x` is returned as the result of the call to `f` from the remote

server, but 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.

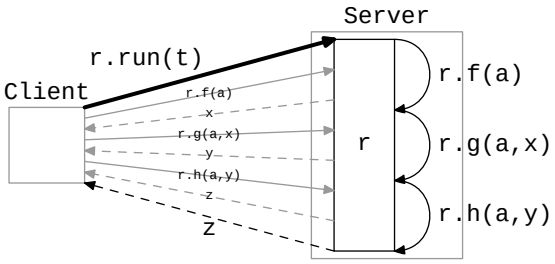
This optimisation methodology is implemented in the Veneer virtual Java Virtual Machine (vJVM) [Yeung and Kelly 2003; Yeung 2004], where sequences of adjacent calls to the same remote object are grouped together into an execution *plan* in bytecode format. This is then uploaded to and executed by the server, with the result of the computation being returned to the client. This simple idea—remote evaluation of code [Stamos and Gifford 1990]—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 [Yeung and Kelly 2003] 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 [Flanagan 2000].

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 [Yeung and Kelly 2003; Yeung 2004], 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 frozen representation of three calls with a closure of free variable a, sending the resulting "thunk" to the server. `thunk<int>` says the frozen code contains an expression of type `int`. We now make only one call across the network to send the frozen



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)

expression, by `r.run(t)`. When the server receives the thunked code, it 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.

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 form of communication-oriented optimisation, which reduces latency and uses bandwidth intelligently, is the idea of *server forwarding* [Yeung and Kelly 2003; Yeung 2004]. 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,
2         RemoteObject r2, int a) {
3    int x1 = r1.f1(a);
4    int y1 = r1.g1(a, x1);
5    int z1 = r1.h1(a, y1);
6    int x2 = r2.f2(z1);
7    int y2 = r2.g2(z1,x2);
8    int z2 = r2.h2(z1,y2);
9    return z2;
10 }
```

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.

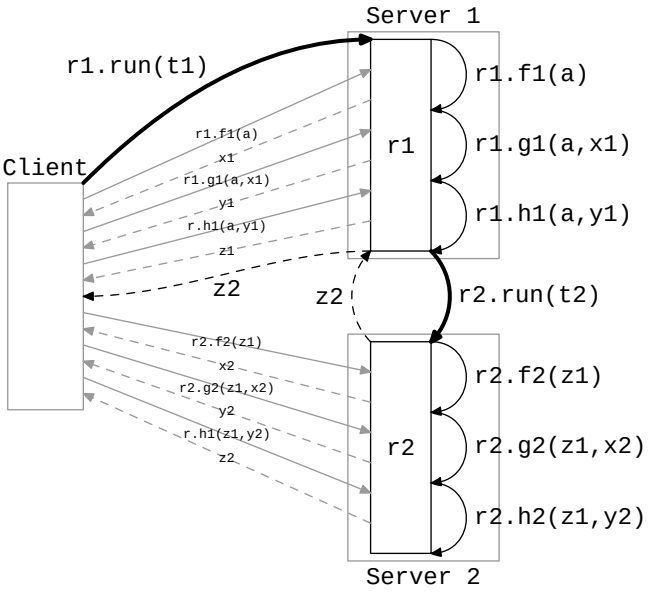


Fig. 2: Example optimisation (2)

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. We use closure passing for representing this optimised code, in which thunked code is *nested*, i.e. we are using higher-order code mobility. Figure 2 gives a diagram of the situation.

```
1  int mOpt2(RemoteObject r1, RemoteObject r2, int a) {
2    thunk<int> t1 = freeze {
3      int x1 = r1.f1(a);
4      int y1 = r1.g1(a, x1);
5      int z1 = r1.h1(a, y1);
6      thunk<int> t2 = freeze {
7        int x2 = r2.f2(z1);
8        int y2 = r2.g2(z1, x2);
9        int z2 = r2.h2(z1, y2);
10       z2;
11     };
12     r2.run(t2);
13   };
14   return r1.run(t1);
15 }
```

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 a from int to class MyObj as in the code in Listing 5:

Here we have two cases:

(1) MyObj is *remote* i.e. when MyObj implements the java.rmi.Remote interface and is therefore remotely callable. In this situation, a is effectively passed by *reference*.

(2) MyObj is *local* i.e. when MyObj is not remotely callable (it 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

Sending a serialised value to a remote consumer can be thought of as passing an object by *value*. Informally, the serialisation process explores the graph under an object in local memory, copying all objects directly or indirectly referred to. When passing such local 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 local class. 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.

```
1  int mOpt3(RemoteObject r, MyObj a) {
2    ser<MyObj> b1 = serialize(a);
3    ser<MyObj> b2 = serialize(a);
4    ser<MyObj> b3 = serialize(a);
5    thunk<int> t = freeze {
6      int x = r.f(deserialize(b1));
7      int y = r.g(deserialize(b2), x);
8      int z = r.h(deserialize(b3), y);
9      z
10   };
11   return r.run(t);
12 }
```

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, since marshaling 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 byte-code 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 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 `r` remote and `x` and `y` local:

```
1  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, 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 [Liang and Bracha 1998; Qian et al. 2000]. 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, 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 sequential 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 [Milner et al. 1992; Sangiorgi 1992; Honda and Tokoro 1991]. For Q3., we summarise our proposal below.

Optimised program 4. Class downloading is a fundamental mechanism in distributed object-oriented 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 thunk 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 following program is able to notice when it is in a high latency situation and act accordingly.

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

Listing 8: Optimised program 5

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 frozen code 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.

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 [Igarashi et al. 2001] and MJ [Bierman et al. 2003], 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*

Syntax occurring only at runtime appears in shaded regions.

(Types)	T ::=	$\texttt{boolean} \mid \texttt{unit} \mid C \mid T \to U$
(Returnable)	U ::=	$\texttt{void} \mid T$
(Classes)	L ::=	$\texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}$
(Constructors)	K ::=	$C(\vec{T}\vec{f})\{\texttt{super}(\vec{f});\texttt{this}.\vec{f} = \vec{f}\}$
(Methods)	M ::=	$U\ m(C\ x)\{e\}$
(Expressions)	e ::=	$v \mid x \mid \texttt{this} \mid \texttt{if}\ (e)\ \{e\}\ \texttt{else}\ \{e\} \mid \texttt{while}\ (e)\ \{e\} \mid e.f \mid e;e$
		$\mid x = e \mid e.f = e \mid \texttt{new}\ C(\vec{e}) \mid e.m(e) \mid T\ x = e$
		$\mid \texttt{return}\ e \mid \texttt{return} \mid \texttt{freeze}[t](T\ x)\{e\} \mid \texttt{defrost}(e;\ e) \mid \texttt{fork}(e)$
		$\mid \texttt{sync}\ (e)\ \{e\} \mid e.\texttt{wait} \mid e.\texttt{notify} \mid e.\texttt{notifyAll} \mid \texttt{new}\ C^l(\vec{e})$
		$\mid \texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e \mid \texttt{resolve}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e \mid \texttt{await}\ c$
		$\mid \texttt{sandbox}\ \{e\} \mid \texttt{insync}\ o\ \{e\} \mid \texttt{ready}\ o\ n \mid \texttt{waiting}(c)\ n \mid \texttt{Error}$
(Tags)	t ::=	$\texttt{eager} \mid \texttt{lazy} \mid \vec{C}$
(Values)	v ::=	$\texttt{true} \mid \texttt{false} \mid \texttt{null} \mid () \mid o \mid \lambda(T\ x).(\nu\ \vec{u})(l, e, \sigma, \texttt{CT}) \mid \epsilon$
(Class Sig.)	CSig ::=	$\emptyset \mid \mathsf{CSig} \cdot [C \mapsto [\texttt{remote}]\ C\ \ \vec{T}\vec{f}\ \{\texttt{m}_\texttt{i} : C_i \to U_i\}]$
(Identifiers)	u ::=	$x \mid o \mid c$
(Threads)	P ::=	$\mathbf{0} \mid P_1 \,|\, P_2 \mid (\nu\, u)P \mid \texttt{forked}\ e \mid \texttt{go}\ e\ \texttt{with}\ c \mid e\ \texttt{with}\ c$
		$\mid \texttt{go}\ e\ \texttt{to}\ c \mid \texttt{return}(c)\ e \mid \texttt{Error}$
(Configurations)	F ::=	$(\nu\ \vec{u})(P, \sigma, \texttt{CT})$
(Networks)	N ::=	$\mathbf{0} \mid l[F] \mid N_1 \,|\, N_2 \mid (\nu\, u)N$
(Stores)	σ ::=	$\emptyset \mid \sigma \cdot [x \mapsto v] \mid \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})]$
(Class tables)	$\texttt{CT}$ ::=	$\emptyset \mid \texttt{CT} \cdot [C \mapsto L]$

Fig. 3: The syntax of the language DJ.

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*. 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. The first primitive, $\texttt{freeze}[t](T\ x)\{e\}$ takes the expression e and, without evaluating it, produces a flattened value representation parameterised 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 $\texttt{freeze}[t]\{e\} \stackrel{\text{def}}{=} \texttt{freeze}[t](\texttt{unit}\ x)\{e\}$ $(x \notin \mathsf{fv}(e))$ and $\texttt{defrost}(e) \stackrel{\text{def}}{=} \texttt{defrost}(();\ e)$ respectively. Note that () denotes a 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 7.1 § 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 agreed interface between 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 "`remote`" if the class is remote. For a class C, the predicate $\mathsf{remote}(C)$ holds iff "`remote`" appears in $\mathsf{CSig}(C)$; otherwise $\mathsf{local}(C)$ holds. Class signatures contain only expected method signatures, not their implementation. This provides a lightweight mechanism for determining the type of remote methods.

3.2 Runtime syntax

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. $\texttt{new}\ C^l(\vec{v})$, $\texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e$ and $\texttt{resolve}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e$ 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 $\texttt{new}\ C^l(\vec{e})$, indicating that the definition of class C can be obtained from a location called l should it need to be instantiated. We write $C^{\text{-}}$ when the treatment of class name C is independent of whether it is labelled or not. $\texttt{await}\ c$ is fundamental to the model of method invocation and can be thought of as the return point for a call. $\texttt{sandbox}\ \{e\}$ represents the execution environment of some code e that originated from a frozen expression.

$\texttt{insync}\ o\ \{e\}$ denotes that expression e has previously acquired the lock on object o. When an expression contains $\texttt{ready}\ o\ n$ as a sub-term it indicates that it is ready to re-acquire the lock on object o. The expression $\texttt{waiting}(c)\ n$ 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 as well as the destination of RMI calls. We shall

often write "o-id" for brevity. Channels c are fundamental to the mechanism of method invocation 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 to compute a value. $\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})$ denotes an expression e frozen with class table `CT` created at l. Expression e is parameterised 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 should attempt to download a copy from l. If σ or `CT` is empty, then we shall omit writing 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 [Milner et al. 1992]. It also includes `Error` which denotes the result of class downloading and 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 $(\nu\,\vec{u})(P, \sigma, \texttt{CT})$ 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 located configurations 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 consists of a mapping from variable names to values, written $[x \mapsto v]$, or 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.`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.

Auxiliary functions. Several auxiliary functions are defined over the syntax of DJ. dom returns the domain of a store and a class table. We also write $\mathsf{dom}(F)$ etc. to denote a 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 $\mathsf{fnv}(N) \stackrel{\text{def}}{=} \mathsf{fv}(N) \cup \mathsf{fn}(N)$. The full definitions is given in the Appendix A.

The set of free class names for a given term is given by the function fcl which

is defined over expressions, threads, configurations and class table entries. The free class names of a value v is defined as $\mathsf{fcl}(v) = \emptyset$. The free class names of an expression and threads are defined recursively as the union of the free class names of all sub-expressions, with the exception that:

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

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

$$\mathsf{fcl}(\texttt{class}\ C\ \texttt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}) = \bigcup \mathsf{fcl}(e_i) \text{ where } M_i = U_i\ m_i(C_i\ x_i)\{e_i\}$$

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

4. OPERATIONAL SEMANTICS

This section presents the formal operational semantics of DJ, extending the standard small step call-by-value reduction of [Pierce 2002; Bierman et al. 2003]. 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 [Milner et al. 1992], written $\equiv$ and given in Appendix B. We define *multi-step* reductions as: $\twoheadrightarrow \stackrel{\text{def}}{=} (\longrightarrow \cup \equiv)^*$ and $\twoheadrightarrow_l \stackrel{\text{def}}{=} (\longrightarrow_l \cup \equiv)^*$.

4.1 Local expressions

The rules for the sequential part of the language are standard [Igarashi et al. 2001; Bierman et al. 2003]. We list the reduction rules in Figure 4. When allocating a new object by New, we explicitly restrict identifiers, which represents "freshness" or "uniqueness" of the address in the store. The auxiliary function $\mathsf{fields}(C)$ is given in Figure 5. It examines the class signature and returns the field declarations for C. We assume the existence of a distinguished class `Object` that is at the top of the inheritance hierarchy and contains no methods and no fields (written $\bullet$).

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.

To reduce the number of computation rules, we make use of the evaluation contexts in Figure 6 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, then any

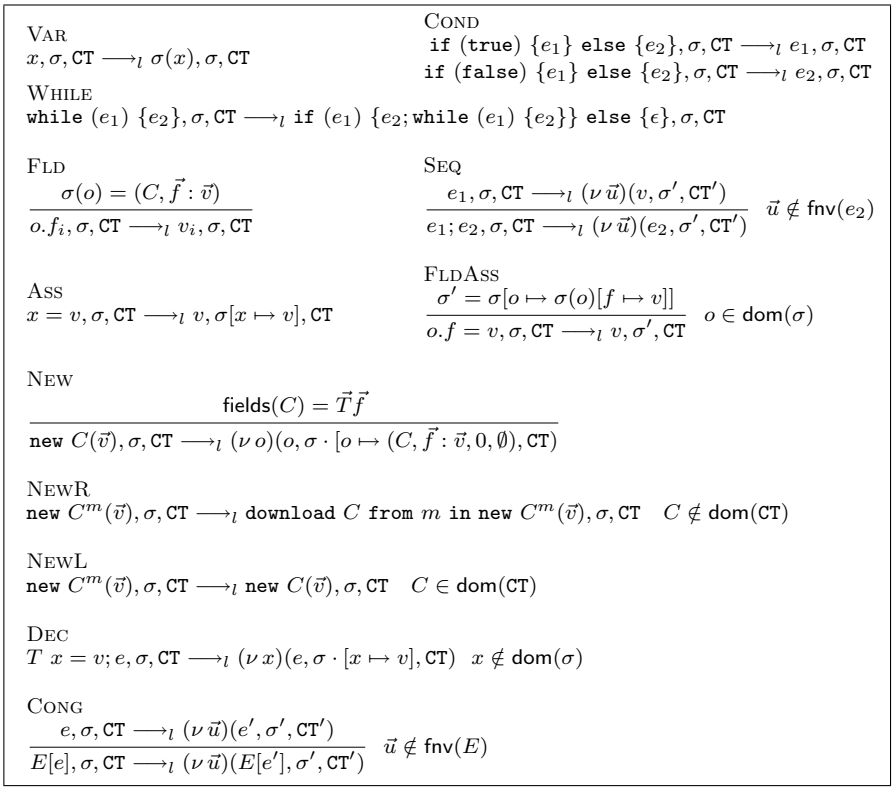


Fig. 4: Local expressions

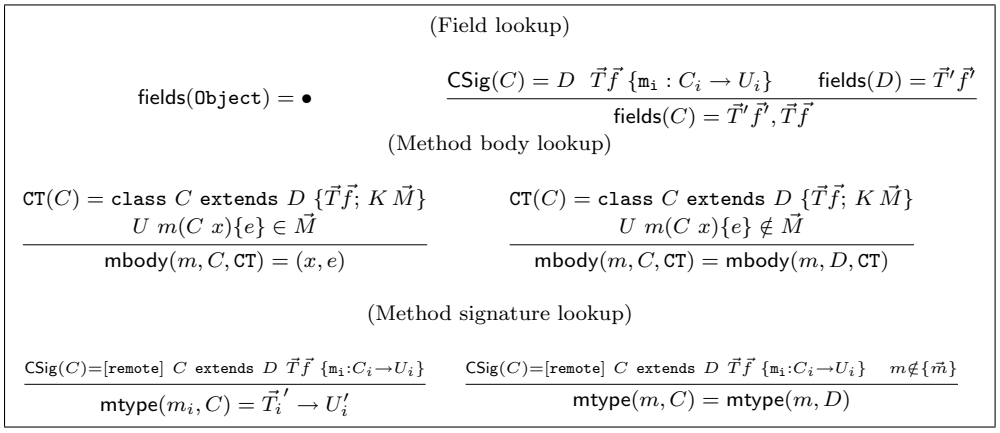


Fig. 5: Lookup functions

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

Fig. 6: Evaluation contexts

RESOLVE
$$\frac{\texttt{CT}(C_i) = \texttt{class}\ C_i\ \texttt{extends}\ D_i\ \{\vec{T}\vec{f};\ K\ \vec{M}\}}{\texttt{resolve}\ \vec{C}\ \texttt{from}\ l'\ \texttt{in}\ e, \sigma, \texttt{CT} \longrightarrow_l \texttt{download}\ \vec{D}\ \texttt{from}\ l'\ \texttt{in}\ e, \sigma, \texttt{CT}}$$

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

DNOTHING
$$\texttt{download}\ \vec{C}\ \texttt{from}\ l'\ \texttt{in}\ e, \sigma, \texttt{CT} \longrightarrow_l e, \sigma, \texttt{CT} \qquad C_i \in \mathsf{dom}(\texttt{CT})$$

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

Fig. 7: Class resolution and downloading

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.

The rules for class downloading in DJ are given in Figure 7 and approximately model the lazy downloading mechanism found in JDK 1.3 without verification [Drossopoulou and Eisenbach 2002]. 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 duplication). We then compute vector $\vec{F}$ from all the free class names mentioned 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 classes. Informally this amounts to downloading immediate superclasses.

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.

We model a failure in this process by the last rule. The rule ERR-CLASSNOTFOUND approximates `ClassNotFoundException` that would occur in the case of the site l_2

not possessing 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 4.2. 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.

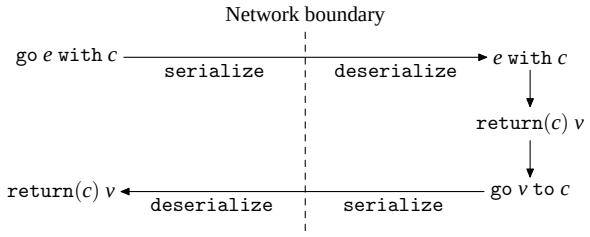
`serialize`(v) and `deserialize`(e) must appear automatically as runtime expressions to serialise parameters and return values of remote method invocations. This is because instances of local classes—those classes without the `remote` keyword in their signature—are incapable of remote method invocation, 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 local 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 [Milner et al. 1992], 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 8(a), which starts from dispatch of a remote method and ends with delivery of its return value. The corresponding formal rules are given in Figure 8(b).

We start our illustration from local method calls. For a method call $o.m(v)$, if



(a) Evaluation steps for a remote call

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

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

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

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

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

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

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

$$\textsc{Err-LostCall}\quad \mathtt{go}\ o.m(v)\ \mathtt{with}\ c, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT} \qquad \textsc{Err-LostReturn}\quad \mathtt{go}\ v\ \mathtt{to}\ c, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$$

(b) Reduction rules

Fig. 8: Remote method invocation

$o \in \mathsf{dom}(\sigma)$ 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.

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 5, 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 $[o/\texttt{this}]$ and a new store entry x is allocated for the formal parameter v. We apply the substitution $e[\texttt{return}(c)/\texttt{return}]$ to indicate that the return value of the method must be sent along channel c. The rule AWAIT is used to communicate the return value to its 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 a local object identifier. 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 $\texttt{go}\ o.m(w)\ \texttt{with}\ c$ 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. In the case of ERR-LOSTCALL, the network becomes partitioned such that a remote method call attempting to reach its destination cannot. 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. The reduction rules are given in Figure 9. We shall focus on the most important rules.

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 executing $E[\texttt{sync}\ (o)\ \{e\}]$, we are attempting to acquire the lock on the object identified by o. To determine whether a lock is taken, the function $\mathsf{getLock}(\sigma, o)$ returns the number of entries to the monitor on object o in store σ. 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 $\mathsf{setLock}(\sigma, o, n')$ with $n' = n + 1$, otherwise execution blocks. The predicates are formally defined as follows.

Definition 4.1. (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:

$$\begin{array}{rcll} \mathsf{getLock}(\sigma, o) & = & n & \qquad \mathsf{blocked}(\sigma, o) = \{\vec{c}\} \\ \mathsf{setLock}(\sigma, o, n') & = & \sigma[o \mapsto (C, \vec{f} : \vec{v}, n', \{\vec{c}\})] & \\ \mathsf{block}(\sigma, o, c) & = & \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \cup \{c\})] & \\ \mathsf{unblock}(\sigma, o, \vec{c}') & = & \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \setminus \{\vec{c}'\})] & \end{array}$$

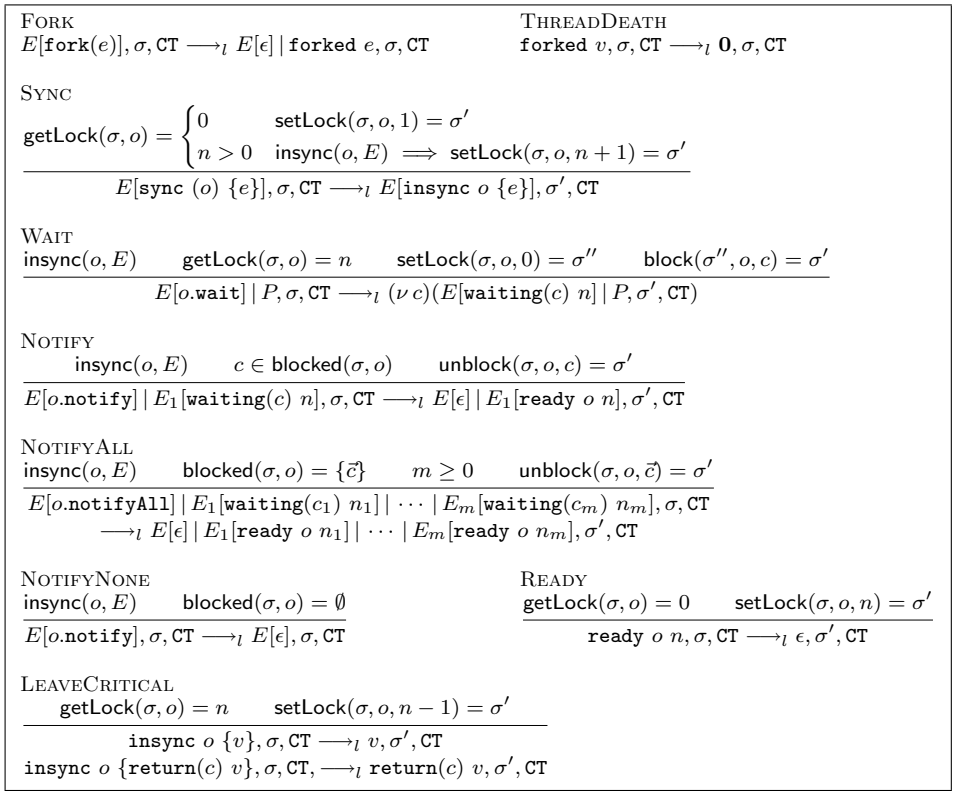


Fig. 9: Concurrency primitives

To temporarily release a lock held on an object o, the command o.`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.

To wake sleeping threads, the commands o.`notify` and o.`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 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 loca-

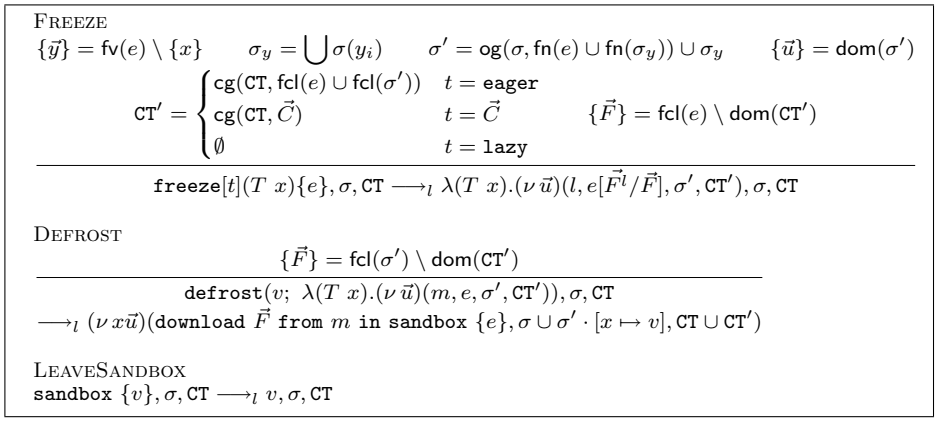


Fig. 10: Creating and executing frozen expressions

tions for evaluation or merely saved for future use. As we have seen in § 3.1, our formulation of the primitives subsumes the serialisation operations 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 10.

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 4.1 to copy all local 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 4.2 to determine the classes that the expression (or the user-specified classes) rely upon.

Finally Figure 11 lists the reduction rules for threads and networks. They are standard using contexts and the structure rules.

4.7 Algorithms

4.7.1 *Object graph.* An object graph is defined as the set of all mappings from an object identifier to store object for every local object transitively referenced by local object identifier v. The lock count, n, for each object is reset to zero when copied and the blocked set $\vec{c}$ emptied to preserve linearity. If the value v refers to a remote object, or a base value such as a boolean, then the object graph is empty.

ALGORITHM 4.1. (Object graph calculation) The function $\mathsf{og}(\sigma, v)$ which computes the object graph of value v in store σ is defined as follows.

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

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 [Ahern and Yoshida 2005a, 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 [Hutchinson et al. 1991] 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 $C \in \mathsf{fcl}(e)$. e indirectly depends upon a class C when $\exists D \in \mathsf{fcl}(e)$ and D is a subclass of C, or C appears free 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 4.2. (Class dependency set calculation)

$$\begin{aligned} \mathsf{cg}(\mathtt{CT}, \vec{M}) &= \bigcup \mathsf{cg}(\mathtt{CT}, \mathsf{fcl}(e_i)) \text{ with } M_i = U_i\ m_i(C_i\ x_i)\{e_i\} \\ \mathsf{cg}(\mathtt{CT}, C) &= \begin{cases} \emptyset & \text{if } C \notin \mathsf{dom}(\mathtt{CT}) \vee C \in \mathsf{dom}(\mathtt{FCT}) \\ \mathsf{cg}(\mathtt{CT}, \mathtt{CT}(C)) & \text{otherwise} \end{cases} \\ \mathsf{cg}(\mathtt{CT}, \vec{C}) &= \bigcup \mathsf{cg}(\mathtt{CT}, C_i) \\ \mathsf{cg}(\mathtt{CT}, L) &= \mathsf{cg}(\mathtt{CT} \setminus C, D) \cup\ \mathsf{cg}(\mathtt{CT} \setminus C, \vec{M}) \cup [C \mapsto L] \\ &\quad \text{where } L = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{\vec{T}\,\vec{f};\ K\ \vec{M}\} \end{aligned}$$

4.7.3 *Correctness of algorithms.* In this subsection we show one of the key results that are used in the proof in § 6, the correctness of the graph algorithms.

First 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. This is defined as follows, where $\sigma(o) = (C, \vec{f} : \vec{v})$:

$$\mathsf{reachable}(\sigma, o, o') \iff (o' \in \mathsf{fn}(\vec{v}) \vee \exists o'' \in \mathsf{fn}(\vec{v}).\mathsf{reachable}(\sigma, o'', o'))$$

We extend this predicate to arbitrary thread P:

$$\mathsf{reachable}(\sigma, P, o) \iff \exists o' \in \{\vec{o}\}.\mathsf{reachable}(\sigma, o', o)$$

where $\{\vec{x}\} = \mathsf{fv}(P)$ and $\{\vec{o}\} = \bigcup \mathsf{fn}(\sigma(x_i)) \cup \mathsf{fn}(P)$.

With this predicate we define the relation $RCH(\sigma)$ which contains all reachable pairs of objects in a store σ as follows:

$$RCH(\sigma) = \{(o, o') \mid \forall o, o' \in \mathsf{dom}(\sigma).o \neq o' \wedge \mathsf{reachable}(\sigma, o, o')\}$$

Our object graph algorithm must, to be correct, preserve the tree structure of the store when copying objects, hence it must preserve this reachability relation.

For a store σ and an object graph σ_g computed from that store, the predicate $\mathsf{ogcomp}(\sigma, \sigma_g)$ (*completeness of an object graph*) holds if the computed graph preserves the reachability relation for all object identifiers in its domain. Given $RCH(\sigma)$ and $RCH(\sigma_g)$, we define:

$$\mathsf{ogcomp}(\sigma, \sigma_g) \stackrel{\mathsf{def}}{=} \forall o \in \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma_g).(o, o') \in RCH(\sigma) \iff (o, o') \in RCH(\sigma_g)$$

This property ensures all links are correctly copied to the graph σ_g, and no new links are created.

4.7.4 *Class dependency algorithm.* The correctness of the class dependency algorithm relies upon the definition of the following predicate:

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

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.

With these preliminaries dealt with, we have the following lemma:

LEMMA 4.1. (Correctness of algorithms)

(1) $\Gamma; \Delta \vdash \sigma : \mathtt{ok}$ *and* $\sigma' = \mathsf{og}(\sigma, o)$ *implies* $\Gamma; \emptyset \vdash \sigma' : \mathtt{ok}$*; for all* $o' \in \mathsf{dom}(\sigma')$*, we have* $\sigma'(o') = (C, \ldots)$ *with* $\mathsf{local}(C)$*; and for all* $o' \in (\mathsf{fn}(\sigma') \setminus \mathsf{dom}(\sigma'))$*,* $\Gamma \vdash o' : C$ *with* $\mathsf{remote}(C)$.

(2) $\sigma' = \mathsf{og}(\sigma, v)$ *implies* $\mathsf{ogcomp}(\sigma, \sigma')$.

(3) *Suppose* $\vdash \mathtt{CT} : \mathtt{ok}$ *with* $C_i \in \mathsf{dom}(\mathsf{CSig})$*. Then we have* $\vdash \mathsf{cg}(\mathtt{CT}, \vec{C}) : \mathtt{ok}$.

(4) $\mathsf{ctcomp}(\mathtt{CT})$ *and* $\mathtt{CT}' = \mathsf{cg}(\mathtt{CT}, C)$ *imply* $\mathsf{ctcomp}(\mathtt{CT}' \cup \mathtt{FCT})$.

PROOF. See Appendix C. □

4.8 Examples of executions

This subsection gives three small examples of the dynamic semantics, focusing on distributed primitives. Since no concurrent threads exist, we omit the counter 0 and an empty queue $\emptyset$ from store entries $[o \mapsto (C, .., 0, \emptyset)]$.

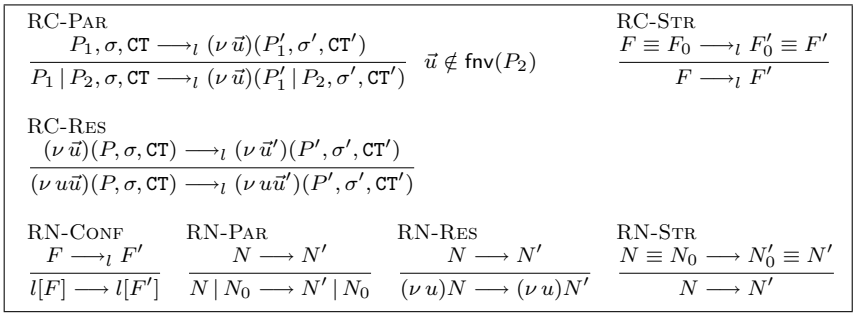


Fig. 11: Network and thread

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:

$$\lambda(\mathtt{int}\ x).(\nu\, o_1, o_2, y)(l, x + y + o_1.f, \sigma_1, \mathtt{CT}_1)$$

where $\sigma_1 = [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \ldots)] \cdot [y \mapsto 6]$ and $\mathtt{CT}_1 = [B \mapsto \ldots]$

```
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 o1 = new A(5, new B());
8  freeze[B] int x){x + y + o1.f};
```

Listing 9: Example of a program using freeze

To defrost a frozen value $\lambda(T\ x).(\nu\, \vec{u})(m, e, \sigma_1, \mathtt{CT}_1)$ we apply Defrost. Firstly, any classes supplied with the frozen value are appended to the current class table. Any class names appearing free in e are tagged with their originating location: $\mathtt{new}\ C(\vec{e})$ becomes $\mathtt{new}\ C^l(\vec{e})$. During execution of the newly defrosted code, when an expression such as the above $\mathtt{new}\ C^l(\vec{v})$ 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 $\mathtt{download}\ \vec{F}\ \mathtt{from}\ m\ \mathtt{in}\ \mathtt{sandbox}\ \{e\}$. This accurately mimics the mechanism employed by the `RMIClassLoader` class used in Java RMI. When sending marshaled

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 `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 l, where it is important to notice that a variable y is already in scope: 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:

$$
\begin{aligned}
& \texttt{defrost}(5;\ t), [y \mapsto \texttt{true}], \texttt{CT} \\
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y_2)(\texttt{dl}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y_2 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_2) \\
\text{with}\ & \sigma_2 = [y \mapsto \texttt{true}] \cdot [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \epsilon)] \cdot [y_1 \mapsto 6] \cdot [x \mapsto 5] \\
\text{and}\ & \texttt{CT}_2 = \texttt{CT} \cdot [B \mapsto \ldots] \\
\longrightarrow_m\ & (\nu\, x, o_1, o_2, y_1)(\texttt{resolve}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y_1 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_3) \\
\text{with}\ & \texttt{CT}_3 = \texttt{CT}_2 \cdot [A \mapsto \ldots]
\end{aligned}
$$

Assuming that the superclass of A is *Object*, this should already be present in the local class table.

$$
\begin{aligned}
& \longrightarrow_m (\nu\, x, o_1, o_2, y_1)(\texttt{dl}\ \mathit{Object}\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y_1 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_3) \\
& \longrightarrow_m (\nu\, x, o_1, o_2, y_1)(\texttt{sb}\ \{x + y_1 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_3) \\
& \twoheadrightarrow_m \texttt{sb}\ \{16\}, [y \mapsto \texttt{true}], \texttt{CT}_3 \quad \longrightarrow_m\ 16, [y \mapsto \texttt{true}], \texttt{CT}_3
\end{aligned}
$$

In the final steps, we 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{ ...}
2 class B { }
3 class C {D m(){return new D()}}
4 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 $\texttt{CT}_5 = \texttt{CT}_3 \cdot [C \mapsto \texttt{class}\ C\ \{D\ \texttt{d}()\{\texttt{return new}\ D^l()\}\}]$. 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 remote class R and classes A and B are local.

```
1 // location l
2 class B { }
3 // Program
4 A o1 = new A(5, new B()); return r.f(o1);
5 // location m
6 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:

$$(\nu\, o_1, o_2)(E[\texttt{return}\ r.f(o_1)], \sigma_1', \texttt{CT}_1')$$

with $\texttt{CT}_1' = [A \mapsto ...] \cdot [B \mapsto ...]$ and $\sigma_2 = [o_1 \mapsto (A, f : 5, g : o_2)] \cdot [o_2 \mapsto (B, \epsilon)]$. This reduces to:

$$\begin{array}{l} \longrightarrow_l (\nu\, o_1, o_2, c)(E[\texttt{await}\ c] \,|\, \texttt{go}\ r.f(\texttt{serialize}(o_1))\ \texttt{with}\ c, \sigma_1', \texttt{CT}_1') \\ \longrightarrow_l (\nu\, o_1, o_2, c)(E[\texttt{await}\ c] \,|\, \texttt{go}\ r.f(v)\ \texttt{with}\ c, \sigma_1', \texttt{CT}_1') \end{array}$$

with $v = \lambda(\texttt{unit}\ x).(\nu\, o_1, o_2)(o_1, \sigma_1')$. 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 to the location m, v is deserialised, starting downloading classes A and B as follows. Below we assume $\texttt{CT}_2' = [R \mapsto ...]$ and $\sigma_2' = [r \mapsto (R, \emptyset)]$:

$$\begin{array}{ll} & r.f(\texttt{deserialize}(v))\ \texttt{with}\ c, \sigma_2', \texttt{CT}_2' \\ \longrightarrow_l & (\nu\, o_1, o_2)(r.f(o_1)\ \texttt{with}\ c, \sigma_1' \cdot \sigma_2', \texttt{CT}_2') \\ \longrightarrow_l & (\nu\, o_1, o_2)(r.f(\texttt{dl}\ A, B\ \texttt{from}\ l\ \texttt{in sb}\ \{o_1\})\ \texttt{with}\ c, \sigma_1' \cdot \sigma_2', \texttt{CT}_2') \\ \rightarrow\!\!\!\!\rightarrow_l & (\nu\, o_1, o_2)(r.f(o_1)\ \texttt{with}\ c, \sigma_1' \cdot \sigma_2', \texttt{CT}_2' \cdot \texttt{CT}_1') \\ \longrightarrow_l & (\nu\, o_1, o_2, x)(\texttt{return}(c)\ x.f(o_1) + 1, \sigma_1' \cdot \sigma_2' \cdot [x \mapsto o_1], \texttt{CT}_2' \cdot \texttt{CT}_1') \\ \rightarrow\!\!\!\!\rightarrow_l & \texttt{return}(c)\ 6, \sigma_2', \texttt{CT}_2' \cdot \texttt{CT}_1' \quad \longrightarrow_l\ \texttt{go}\ 6\ \texttt{to}\ c, \sigma_2', \texttt{CT}_2' \cdot \texttt{CT}_1' \end{array}$$

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 $E[6]$ by Await, as required.

5. TYPING SYSTEM

This section presents the key 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 12.

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 empty meaning. We write $C <: D$ when class C is a subtype of class D. Our notion of subtyping is mostly standard (we assume $<:$ causes no cycle as in [Igarashi et al. 2001; Bierman et al. 2003]). It is given in Figure 12. The arrow type is standard.

$$
\begin{array}{lll}
T & ::= \texttt{boolean} \mid \texttt{unit} \mid C \mid T \to U & \text{(Types)} \\
U & ::= \texttt{void} \mid T & \text{(Returnable types)}
\end{array}
$$

Extended types not appearing in program text:

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

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

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

Fig. 12: Types and subtyping

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 ($U\ m(C\ x)\{e\}$ 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 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 12, $\texttt{chanI}(U)$ 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.

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

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

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

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 5.1. The commutative, partial, binary composition operator on channel types, $\odot$, 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:

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

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 `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 $P \mid Q$ 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 in Figure 13. There are six kinds of judgement, and all are interrelated. In the following we assume α ranges over either τ, S, U or C $\texttt{extends}$ D $\vec{T}\vec{f}$ $\{\texttt{m}_\texttt{i} : C_i \to U_i\}$.

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

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

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:

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

where α ranges over T, U and S, and the typing rules are given in Figure 14. The typing judgement is lightweight or 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 CT are well-typed, 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

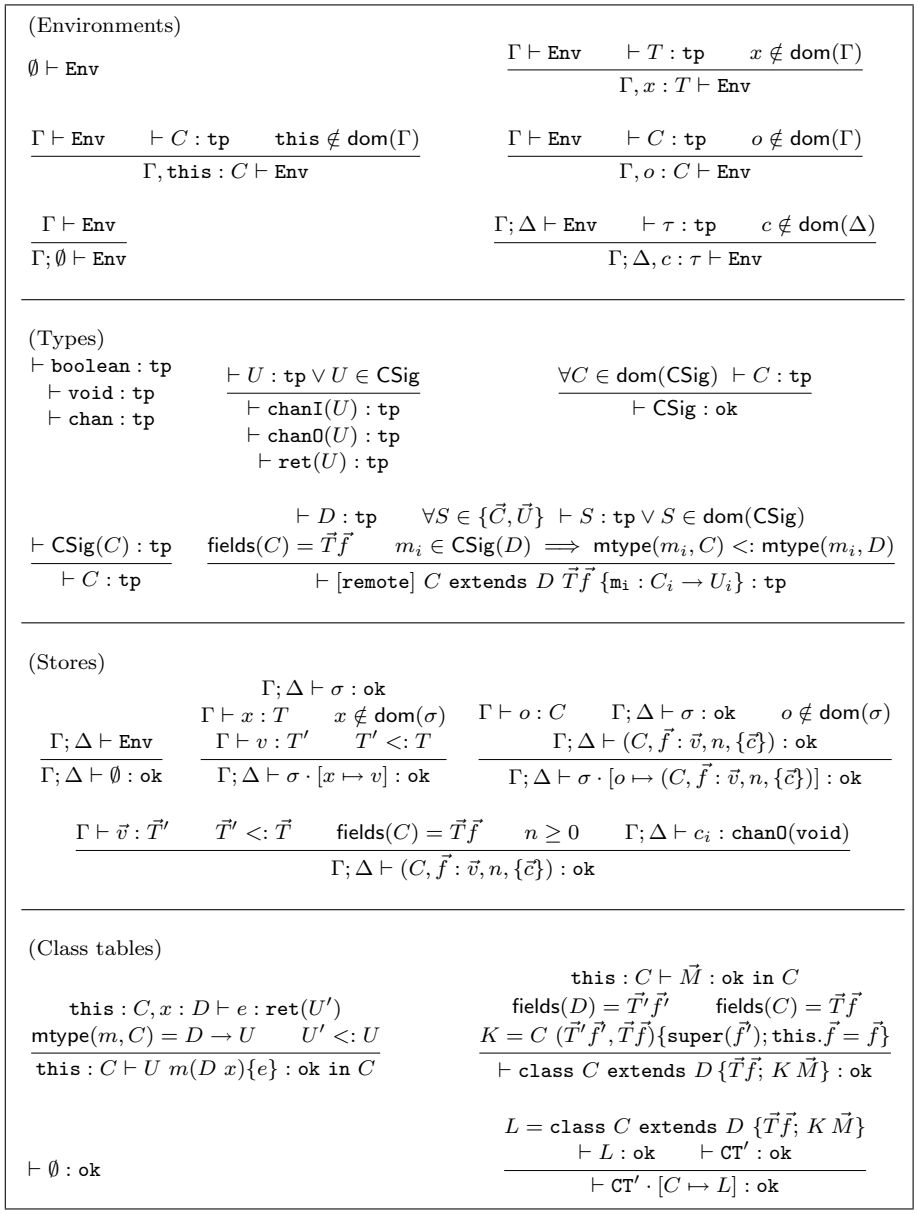


Fig. 13: Well-formedness

assumption that runtime values are created under the invariants defined in § 6. By combining with the invariants, we shall see:

- Instances of remote classes are not contained in σ, i.e. if $o \in \mathsf{dom}(\sigma)$, then we have $\sigma(o) = (C, \ldots)$ with $\mathsf{local}(C)$. 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 local object-identifiers: for exam-

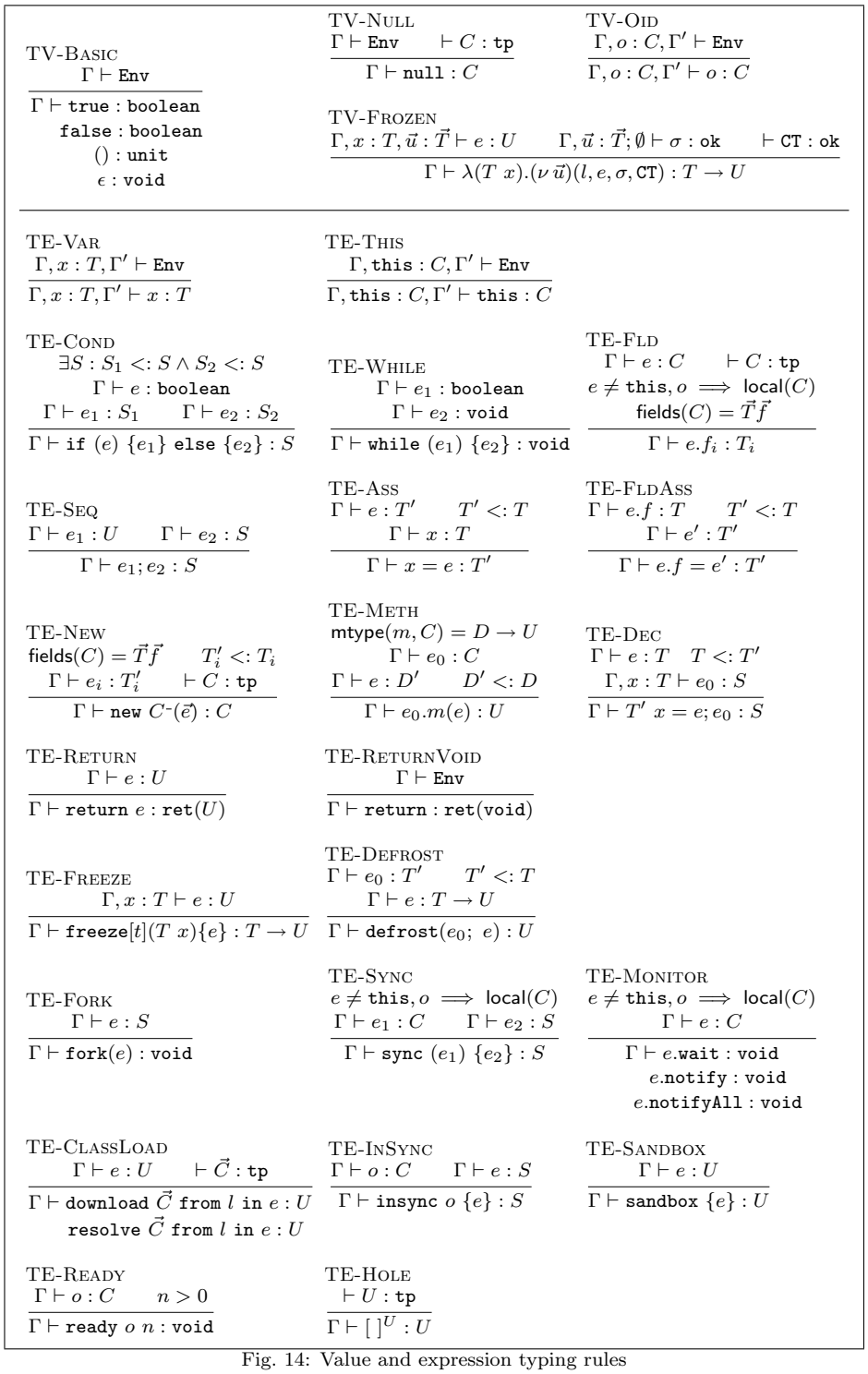


Fig. 14: Value and expression typing rules

ple, 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.

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

5.3.1 *Locality for field 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 remote object in a different location. Hence the following should be prohibited even if class C is remote.

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

However we wish to allow to type the following with class C remote:

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

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

In order to propose a typing rule to prevent remote field access statically but allow field access on remote 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 6.3. The rule TE-Fld restricts field accesses only for local classes if e is neither `this` or o. The special expression `this` is allowed to have a remote 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 field access is always local.

The second restriction with respect to Java implementation is on thread synchronisation: performing thread synchronisation on a remote object is undefined behaviour. In Java it is possible to synchronise on the *stub* to a remote object, but this is not the same as synchronising on the actual remote object, since it does not acquire the lock on the underlying object held at the remote site and does not prevent other clients in the network from accessing that resource.

```
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 the remote 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. Combining the invariants of locality, then we can now 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 are given in Figure 15. 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 $\Delta_1 \asymp \Delta_2$.

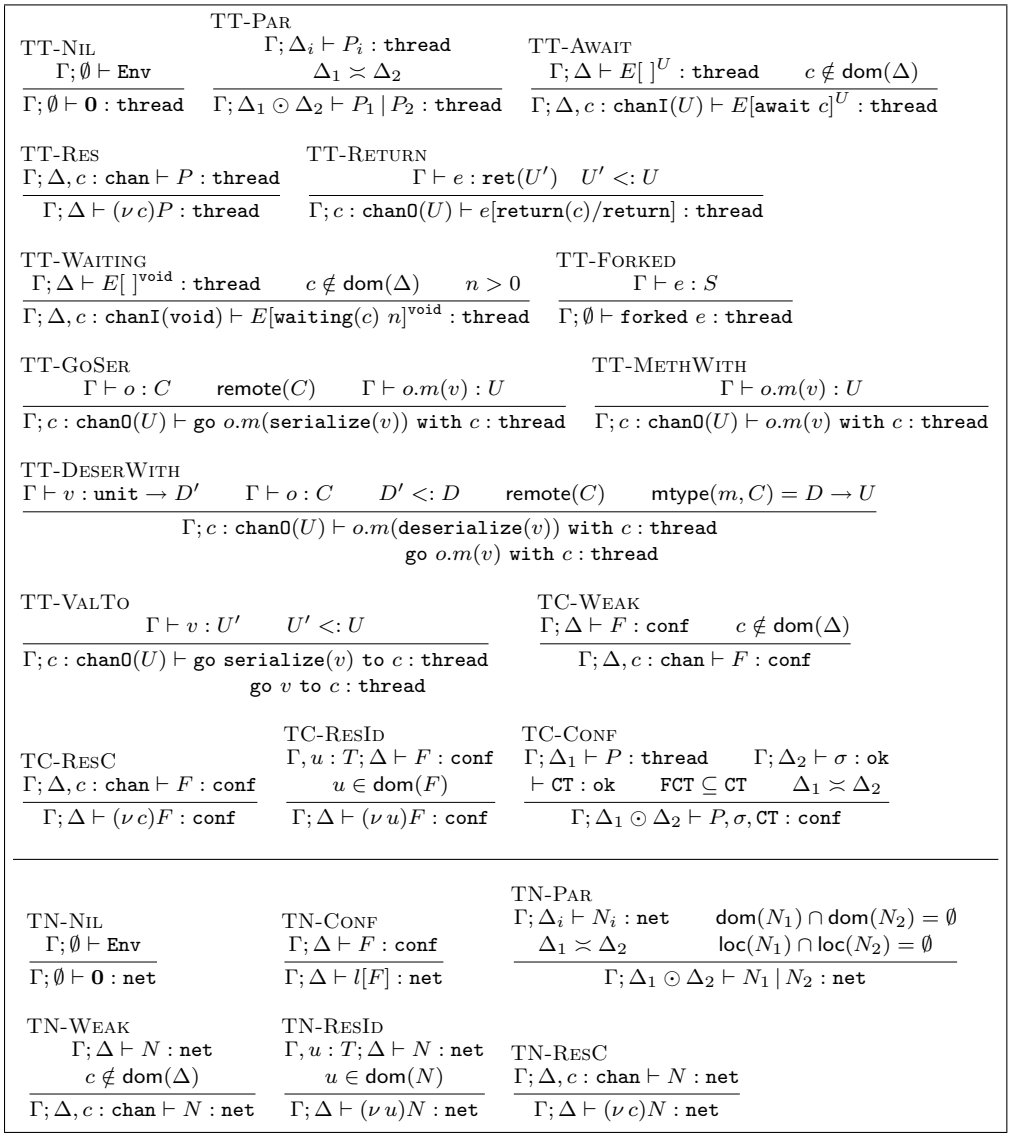


Fig. 15: Thread and network typing rules

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 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 $\mathsf{loc}((\nu\, u)N) = \mathsf{loc}(N)$. We also use $\Delta_1 \asymp \Delta_2$ as TC-Conf

to check the composability.

6. NETWORK INVARIANTS AND TYPE SOUNDNESS

This section presents the main technical results of the present paper. 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 safety guarantees.

6.1 Network invariants

We start from the definition of a property over networks, given in Definition 6.1.

Definition 6.1. (Properties) Let ψ denote a property over networks (i.e. ψ is a subset of 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 $\mathtt{Error}$ as subexpression, i.e. $\mathsf{Err} = \{N \mid N \equiv (\nu\,\vec{u})(l[E[\mathtt{Error}] \mid P, \sigma, \mathtt{CT}] \mid N')\}$. 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 typable 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 6.1. (Canonical forms) *Suppose that* $\Gamma; \Delta \vdash N : \mathtt{net}$ *then we have* $N \equiv (\nu\,\vec{u})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ *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 6.2. (Structural equivalence preserves typability)

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

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 the preservation of safety, we require certain properties to remain invariant.

Definition 6.2. (Network invariants) Given network $N \equiv (\nu\,\vec{u})(\prod_{0 \le i < n} l_i[F_i])$ with $F_i = (P_i, \sigma_i, \mathtt{CT}_i)$, and assuming $0 \le j < n$, $i \ne j$ 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*

$\mathsf{Inv}(1)$ $\texttt{FCT} \subseteq \texttt{CT}_i$

$\mathsf{Inv}(2)$ $P_i \equiv E[\texttt{new}\ C(\vec{v})] \mid Q_i \implies \mathsf{comp}(C, \texttt{CT}_i)$

$\mathsf{Inv}(3)$ $C \in \mathsf{dom}(\texttt{CT}_i) \cap \mathsf{dom}(\texttt{CT}_j) \implies$
$\texttt{CT}_i(C) = \texttt{CT}_j(C) \vee \texttt{CT}_i(C) = \texttt{CT}_j(C)[\vec{D}^{l_i}/\vec{D}]$ with $\mathsf{fcl}(\texttt{CT}_i(C)) = \{\vec{D}\}$

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) { this.i = i; }
5 }
```

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) { this.i = i; }
4 }
```

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 [Greanier 2005].[1]

6.1.2 *Locality*

$\mathsf{Inv}(4)$ $\mathsf{fv}(P_i) \subseteq \mathsf{dom}(\sigma_i) \subseteq \{\vec{u}\}$

$\mathsf{Inv}(5)$ $\mathsf{dom}(\sigma_i) \cap \mathsf{dom}(\sigma_j) = \emptyset$

$\mathsf{Inv}(6)$ $o \in \mathsf{fn}(F_i) \cap \mathsf{fn}(F_j) \implies \exists!k.\ \sigma_k(o) = (C, \ldots) \wedge \mathsf{remote}(C)$

$\mathsf{Inv}(7)$ $o \in \mathsf{fn}(F_i) \wedge \exists k.\ \sigma_k(o) = (C, \ldots) \wedge \mathsf{local}(C) \implies k = i$

$\mathsf{Inv}(8)$ $o \in \mathsf{fn}(F_i) \implies \exists k\ 1 \leq k \leq n.\ o \in \mathsf{dom}(\sigma_k)$

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

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

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 Inv(4) and 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 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 *thread P inputs at c* if $P \equiv E[\texttt{await}\ c] \mid R$ or $P \equiv E[\texttt{waiting}(c)\ n] \mid R$ for some E and 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.

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.

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

Inv(12) $P_i \equiv E[v] \mid Q_i$ then $\mathsf{fv}(v) = \emptyset$

Inv(13) $\sigma_i(x) = v \implies \mathsf{fv}(v) = \emptyset$

Inv(14) $\sigma_i(o) = (C, \vec{f} : \vec{v}) \implies \mathsf{fv}(v_j) = \emptyset$

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

Locks

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

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

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. Inv(16) ensures that a thread that is ready to reacquire a lock will set that lock's count to a non-zero number. 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. It can, however, contain parallel threads distributed among

locations; these have been generated by compiling multiple user-defined main programs. Definition 6.3 states these conditions formally.

Definition 6.3. (Initial network) We call network $N \equiv (\nu\, \vec{u})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ an *initial network* if it satisfies the following conditions (called *initial properties*):

- it contains no runtime expressions or values except o-ids 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) $\mathsf{fcl}(P_i) \subseteq \mathsf{dom}(\mathtt{CT}_i)$,
  (b) $C \in \mathsf{fcl}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_i) \implies \mathsf{comp}(C, \mathtt{CT}_i)$ and
  (c) $\sigma_i(o) = (C, \dots) \implies \mathsf{comp}(C, \mathtt{CT}_i)$.
- we also strengthen the $\mathsf{Inv}(9)$ by replacing the reduction context E by the arbitrary context $\mathcal{C}$ in R_i.

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

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

First A should be defined in $\mathtt{CT}$ (this is ensured by (a) in $\mathsf{Inv}(2')$); secondly B should be also defined in $\mathtt{CT}$ (this is ensured by (a) and (b): since $A \in \mathsf{dom}(\mathtt{CT})$, we have $\mathsf{comp}(A, \mathtt{CT})$, which implies $B \in \mathsf{dom}(\mathtt{CT})$); and thirdly, C should be defined in $\mathtt{CT}$ too since $\mathtt{new}\ C()$ appears after the method invocation at $\mathtt{m}$. This condition is ensured by (b) since $C \in \mathsf{fcl}(\mathtt{CT})$. 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 6.3 and prove the invariant $\mathsf{Inv}(2)$. The initial condition of $\mathsf{Inv}(9)$ is similarly understood as (c).

6.3 Type soundness 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 soundness 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* $\Gamma; \Delta \vdash N_0 : \mathtt{net}$ *and* N_0 *satisfies the initial properties. Then* $N_0 \longrightarrow N_1$ *implies* $N_1 \models \mathsf{Inv}(r)$ *for each* $1 \le r \le 17$ *if* $N_1 \not\models \mathsf{Err}$.

(ii) *Assume* $\Gamma; \Delta \vdash N_m : \mathtt{net}$ $(m \ge 1)$ *and* $N_m \models \mathsf{Inv}(r)$ *for all* $1 \le r \le 17$. *Then* $N_m \longrightarrow N_{m+1}$ *implies* $N_{m+1} \models \mathsf{Inv}(r)$ *for each* $1 \le r \le 17$ *if* $N_{m+1} \not\models \mathsf{Err}$.

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

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.

6.4 Proofs of network invariants

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

LEMMA 6.3. (Class table properties) *Assume:*

$\Gamma;\Delta \vdash N_k : \mathtt{net}$ *for* $0 \leq k \leq m$, $\quad N_0 \in \mathsf{Init}$, $\quad N_k \in \mathsf{Inv}(r)$ *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*) $C \in \mathsf{fcl}(P_{im+1})$ *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* $\neg\mathsf{reachable}(\sigma_{im+1}, Q_{im+1}, o)$
(*4*) $\exists N_k \equiv (\nu\, \vec{u}_k)(\prod_{0\leq i<n} l_i[P_{ik}, \sigma_{ik}, \mathtt{CT}_{ik}])$ *with* $P_{ik} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{ik}$ *and* $0 \leq j < n$ *implies*

  $\forall C_z \in \{\vec{C}\}.\forall C'.C_z <: C'.\exists N_m.N_k \twoheadrightarrow N_m$ *and*

  $P_{im} \equiv\ E[\mathtt{resolve}\ \vec{D}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im}$ *with* $C' \in \{\vec{D}\}$ *and* $C' \in \mathsf{dom}(\mathtt{CT}_{im})$

Lemma 6.3 says (1) the class table at each location always increases; (2) if a class name appears free 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 is in the middle of downloading. (4) all superclasses will be eventually downloaded if no error occurs.

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

LEMMA 6.4. (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* $\mathsf{Err} \not\models N_k$.

(*1*) *If* $P_{im+1} \equiv E_1[\mathtt{insync}\ o\ \{\ldots E_p[\mathtt{insync}\ o\ \{e\}]\ldots\}] \,|\, Q_{im+1} \wedge e \neq E'[\mathtt{insync}\ o\ \{e'\}]$ *then:*
  (*a*) $e \neq E[\mathtt{waiting}(c)\ n']$ *with* $c \in \mathsf{blocked}(o, \sigma_{im+1})$ *and* $e \neq E[\mathtt{ready}\ o\ \ldots]$. *implies* $\mathsf{getLock}(\sigma_{im+1}, o) = p$,
  (*b*) $e = E[\mathtt{ready}\ o\ n'] \implies p = n'$,

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

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

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

Summary of the one-step invariant proof. We summarise the proofs of **Step 1** for each invariant. We use the 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 6.3 (1). For $\mathsf{Inv}(3)$, we analyse two rules, Download and Defrost, which changes the class table. $\mathsf{Inv}(4)$ 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 6.3 (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 4.1. For $\mathsf{Inv}(16)$ and $\mathsf{Inv}(17)$, we use Lemma 6.4.

6.5 Proofs of type soundness

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

Lemma 6.5. (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) $\Gamma, \texttt{this} : C \vdash e : \alpha$ *and* $\Gamma \vdash o : C'$ *with* $C' <: C$ *imply* $\Gamma \vdash e[o/\texttt{this}] : \alpha'$ *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 the expression e using Lemma 6.2. See Appendix F.1. (3) is by induction on the structure on E. All proofs are mechanical. □

Now we achieve the main theorem.

Theorem 6.1. **(Subject reduction)**

(*1*) *Assume* $\Gamma, \vec{u} : \vec{T} \vdash e : \alpha$, $\Gamma, \vec{u} : \vec{T} \vdash \sigma : \texttt{ok}$ *and* $\vdash \texttt{CT} : \texttt{ok}$. *Suppose* $(\nu\, \vec{u})(e, \sigma, \texttt{CT}) \longrightarrow_l (\nu\, \vec{u}')(e', \sigma', \texttt{CT}')$ *and* $e' \not\models \mathsf{Err}$. *Then we have* $\Gamma, \vec{u}' : \vec{T}' \vdash e' : \alpha'$ *for some* $\alpha' <: \alpha$, $\Gamma, \vec{u}' : \vec{T}' \vdash \sigma' : \texttt{ok}$ *and* $\vdash \texttt{CT}' : \texttt{ok}$.

(*2*) *Assume* $\Gamma; \Delta \vdash F : \texttt{conf}$, $F \longrightarrow_l F'$ *and* $F' \not\models \mathsf{Err}$. *Then we have* $\Gamma; \Delta \vdash F' : \texttt{conf}$.

(*3*) *Assume* $\Gamma; \Delta \vdash N : \texttt{net}$, $N \longrightarrow N'$ *and* $N' \not\models \mathsf{Err}$. *Then we have* $\Gamma; \Delta \vdash N' : \texttt{net}$.

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.

As a corollary we derive:

COROLLARY 6.1. ***(Network invariant)*** $\wedge_{1 \le r \le 17}\, \mathsf{Inv}(r)$ *is a network invariant with the initial network properties* Init *defined in Definition 6.3.*

6.6 Progress and Linearity Properties

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

Definition 6.4. (Progress invariants) Given network $N \equiv (\nu\, \vec{u})(\prod_{0 \le i < n} l_i[P_i, \sigma_i, \texttt{CT}_i])$, and assuming $0 \le k < n$, we define property $\mathsf{Prog}(r)$ as a set which satisfy the following conditions.

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

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

$\mathsf{Prog}(3)$ $P_i \equiv E[\texttt{resolve}\ \vec{C}\ \texttt{from}\ m\ \texttt{in}\ e]\,|\,Q_i \implies \vec{C} \in \mathsf{dom}(\texttt{CT}_i)$
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 they are local to.

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

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

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

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

THEOREM 6.2. (Progress, locality and linearity)
$\wedge_{1\leq r\leq 10}$ Prog(r) *is a network invariant with the initial network properties* Init *defined in Definition 6.3.*

PROOF. Immediately Prog(1) is derived from Inv(2). Prog(2) is by the monotonicity of the class tables proved in Lemma 6.3 (1). Prog(3) is obvious by DOWNLOAD. Prog(4) and Prog(5) are proved by Inv(9). Prog(6) and Prog(7) are obvious by Inv(4). Prog(8) is derived from Inv(9). Prog(9) is by combining Inv(8) and Inv(5). Prog(10) is straightforward by combining Inv(10) and Inv(11). □

6.7 Progress with synchronisation and normal forms

In this last subsection, we first investigate a simple progress property in the presence of the synchronisation primitives. Then we will show the normal form of DJ—the form of a whole network when computation terminates. We start from the first proposition which says that one monitor is held by only one thread.

PROPOSITION 6.1. (Mutual exclusion) *For a location* $l[P, \sigma, \texttt{CT}]$*, suppose*

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

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 exit the monitor if the monitor is only hold by threads who are going to exit.

PROPOSITION 6.2. (Progress with synchronisation) *For a location* $l[P, \sigma, \texttt{CT}]$*,*

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

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

PROOF. (1) If P_i satisfies the assumption, then by Proposition 6.1, $E[\texttt{insync}\ o\ \{E'[e]\}]$ is only the thread which holds the monitor o. Hence progress is obvious by the definition of $\longrightarrow$. (2) is by $\mathsf{getLock}(\sigma, o) = 0$. □

In the presence of synchronisation, there would be no progress in the program even if it is well-typed and does not reach an error state. For example, threads may

deadlock naturally by requesting monitors in a certain order, stopping them from proceeding forever: a simple example is
$E[\texttt{insync}\ o\ \{\texttt{sync}\ (o)\ \{e\}\}]$. 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 shall leave this topic to a forthcoming exposition. We now concludes the normal form theorem.

THEOREM 6.3. **(Normal forms)** *Assume* $N_0 \models \mathsf{Init}$ *and* $N_0 \twoheadrightarrow N \not\longrightarrow$ *and* $N \not\models \mathsf{Err}$. *Then we have* $N \equiv (\nu\,\vec{u})(\prod_{0\leq i<n} l_i[\prod_{0\leq j_i<n_i} P_{j_i}, \sigma_i, \mathtt{CT}_i])$ *with* P_{j_i} *is either* $\texttt{go}\ v\ \texttt{to}\ c$ *or* $E[\texttt{insync}\ o\ \{e\}]$ *with* $e \in \{\texttt{waiting}(o)\ n, \texttt{ready}\ o\ n, \texttt{sync}\ (o')\ \{e'\}\}$.

PROOF. By induction on 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 $P_i \equiv o.m(\vec{v})\ \texttt{with}\ c\,|\,Q_i$. If o is the local object id, then $N \longrightarrow N'$ by $\mathsf{Prog}(8)$. Assume that o is a remote o-id and $o \notin \mathsf{dom}(\sigma_i)$. This time by METHREMOTE, $N \longrightarrow N'$, contradiction. Next suppose there exists P_i such that $P_i \equiv \texttt{go}\ v\ \texttt{to}\ c\,|\,Q_i$ with $c \in \{\vec{u}\}$ or $c : \texttt{chan} \in \Delta$. Then by $\mathsf{Prog}(10)$, there exists k such that $P_k \equiv E[\texttt{await}\ c]\,|\,Q_k$. 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. JUSTIFICATION OF OPTIMISATION

We prove the correctness of the optimised code in § 2 using sound syntactic transformation rules over programs and runtime. The key idea is a use of the following *noninterference property* [Jones 1993; Reynolds 1978] to justify the correctness of these rules. Let us write $N \mapsto N'$ for a transformation rule of the optimisation from N to N'. Once we check $N \mapsto N'$ is type-preserving and satisfies the following noninterference property then N and N' are immediately observationally equal, hence the transformation is semantics-preserving.

> if $N \twoheadrightarrow N_1$ and $N \mapsto^* N_2$, then $N_1 \equiv N_2$ or there exists N' such that $N_1 \mapsto^* N'$ and $N_2 \twoheadrightarrow N'$.

For tractable reasoning, we introduce several syntactic transformation rules satisfying this property. By the use of these equational laws, which come from those of the linear types of mobile processes [Kobayashi et al. 1996; Yoshida et al. 2001], justifications are carried out syntactically using $\mapsto$.

7.1 Observational congruence

We define an observational congruence over the typed language and runtime by applying the equational theory of process algebra [Honda and Yoshida 1995]. Hereafter we assume all networks are typed and started executing from the initial condition Init in Definition 6.3.

Definition 7.1. (Typed relations) 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* is a typed relation $\mathcal{R}$ which is an equivalence closed under all typed contexts and the structure rules, i.e. $\equiv \subseteq \mathcal{R}$.

By the subject reduction theorem, we immediately know $\longrightarrow$ is a typed relation.

The formulation of behavioural equality is based on two conditions: reduction-closedness and an observational predicate. In the distributed setting, terms can effectively change meaning (for example by side-effecting a store), so we define "equality" to mean that two equated programs go to an equated state again. The second condition comes from the concept of observation in mobile process theory [Honda and Yoshida 1995]. For an observation, we take the output ("go") to channel c.

Definition 7.2. (Reduction-closedness and the observational predicate)

- A typed congruence $\mathcal{R}$ on networks is *reduction-closed* whenever $\Gamma;\Delta \vdash N_1 \mathcal{R} N_2$, $N_1 \longrightarrow\!\!\!\!\!\rightarrow N_1' \not\models \mathsf{Err}$ implies, for some N_2', $N_2 \longrightarrow\!\!\!\!\!\rightarrow N_2'$ with $\Gamma;\Delta \vdash N_1' \mathcal{R} N_2'$; and its symmetric case.
- We define the observational predicate $\downarrow_c$ and $\Downarrow_c$ as follows.

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

We say $\mathcal{R}$ respects the observational predicate if $\Gamma;\Delta \vdash N_1 \mathcal{R} N_2$ with $c : \mathtt{chan0}(U) \in \Delta$ implies $N_1 \Downarrow_c$ iff $N_2 \Downarrow_c$.

Now we define the observational congruence.

Definition 7.3. (Observational congruence) A typed congruence $\mathcal{R}$ is *sound* if it is reduction-closed and respects the observational predicate.

- We write $\cong$ for the maximum sound equality over a network invariant, i.e. $\cong$ is defined over a set which excludes the error states $\{N \mid \exists N_0.(N_0 \models \mathsf{Init},\ N_0 \longrightarrow\!\!\!\!\!\rightarrow N,\ N \not\models \mathsf{Err})\}$.
- We write $\cong^{\bullet}$ for the maximum sound equality over untyped networks which include error states.

7.2 Transformation

We introduce a set of tractable conversion rules which can quickly check the equivalence of distributed networks. First we formally define the noninterference property.

Definition 7.4. Let us assume $\mapsto$ is a typed relation closed under name restriction, parallel composition and the structure rules. We say $\mapsto$ satisfies a *noninterference property*, i.e. if $N \longrightarrow\!\!\!\!\!\rightarrow N_1$ and $N \mapsto^* N_2$, then $N_1 \equiv N_2$ or there exists N' such that $N_1 \mapsto^* N'$ and $N_2 \longrightarrow\!\!\!\!\!\rightarrow N'$.

Lemma 7.1. *Suppose $\mapsto$ satisfies a noninterference property and $\mapsto$ respects the observational predicate. Then $N_1 \mapsto N_2 \not\models \mathsf{Err}$ implies $N_1 \cong N_2$.*

Proof. By taking $\mathcal{R} = \{(N, M) \mid N \mapsto^* M \not\models \mathsf{Err}\}$ and showing it is sound. By the assumption, we know $\mapsto$ respects the action predicate and so does $\mapsto^*$. Then it remains to show that $\mathcal{R}$ is reduction closed. Suppose $N \mathcal{R} M$ and $N \longrightarrow\!\!\!\!\!\rightarrow N'$ then by

the definition of noninterference, $N' \equiv M$ or $M \twoheadrightarrow M'$ such that $N' \mapsto^* M$. Since $\equiv \subset \mapsto^* \subset \mathcal{R}$, this completes the case. Now suppose $N\mathcal{R}M$ and $M \twoheadrightarrow M'$ then by Definition 7.3 we have that $N\mathcal{R}M \twoheadrightarrow M'$ and so $N \twoheadrightarrow N' \equiv M'$. $\square$

Code that can move safely. The transformation rules should reduce the number of communications and class downloadings, while preserving meaning. For this, we need to identify what kinds of code and programs can safely move from one location to another. Below predicate $\mathsf{Mobile}_\Gamma(e)$ is true if $\mathsf{fv}(e) = \emptyset$ and $o \in \mathsf{fn}(e)$ implies $\Gamma \vdash o : C$ with $\mathsf{remote}(C)$; i.e. e does not contain any free variables or local o-ids under environment Γ; in addition it does not contain any of the following terms as a subterm (since they break the locality invariants, see § 5.3.1 and § 6.1.2).

$$\{o.f, o.f = v, \texttt{sync}\ (o)\ \{e'\}, \texttt{insync}\ o\ \{e'\}, o.\texttt{notify}, o.\texttt{notifyAll}, o.\texttt{wait}, \texttt{ready}\ o\ n\ \}$$

If $\mathsf{Mobile}_\Gamma(e)$, e can *move* from one location to another preserving its meaning.

Transformation Rules. We define the key translation rules below, assuming that the right hand side is typed under $\Gamma; \Delta$. We omit surrounding context where it is unnecessary. Assume the right hand side is typed under $\Gamma; \Delta$. We omit the surrounding context where it is unnecessary.

Linearity

(l1) $\texttt{return}(c)\ E[\texttt{sandbox}\ \{e_1; \ldots; e_n\}] \mapsto e_1; \ldots; \texttt{return}(c)\ E[e_n]$

(l2) $E[\texttt{await}\ c] \,|\, e[\texttt{return}(c)/\texttt{return}] \mapsto E[\texttt{sandbox}\ \{e[e'/\texttt{return}\ e']\}]$

(l1) is standard. (l2) means that a method body e can be evaluated inline. This is ensured by linearity of channel c.

Class

(cm) $l[P, \sigma, \texttt{CT}] \mapsto l[P, \sigma, \texttt{CT} \cup \texttt{CT}']$ $\quad \mathsf{ctcomp}(\texttt{CT}'),\ \vdash \texttt{CT}' : \texttt{ok}$

(cm) says that a complete class table can always move.

Closed

(cr) $(\nu\, x)(E[x] \,|\, P, \sigma \cdot [x \mapsto v]) \mapsto (E[v], \sigma)$ $\quad$ when $\{x := e\} \notin P \cup E[\ ]$

(fr) $(\nu\, \vec{u})(E[\texttt{freeze}[t](T\ x)\{e\}] \,|\, P, \sigma, \texttt{CT}) \mapsto (\nu\, \vec{u})(E[\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \texttt{CT}')] \,|\, P, \sigma, \texttt{CT})$

where in (fr), $\mathsf{dom}(\sigma') \cap (\mathsf{fnv}(P) \cup \mathsf{fnv}(E)) = \emptyset$ and $\mathsf{fnv}(\sigma') \subseteq \mathsf{dom}(\sigma')$ and σ' and $\texttt{CT}'$ are given following FREEZE. (cr) says that the timing of reading a value is unimportant if x only does not write in its scope. (fr) means that the timing of freezing an expression is ignored, provided it shares no information with other parties. Note "$\nu\, x$" in (cr) and "$\nu\, \vec{u}$" in (fr) ensure x and u_i are not shared.

Method Invocation

(mi) $l[E[o.m(v)], \sigma, \texttt{CT}] \,|\, m[Q, \sigma', \texttt{CT}']$

$\quad \mapsto l[E[\texttt{defrost}(v;\ \lambda(T\ x).(m, e[o/\texttt{this}], \emptyset, \emptyset))], \sigma, \texttt{CT}] \,|\, m[Q, \sigma', \texttt{CT}']$

where $\mathsf{Mobile}_\Gamma(e[v, o/x, \texttt{this}])$, $\mathsf{Mobile}_\Gamma(v)$, $[o \mapsto (C, \ldots)] \in \sigma'$, $\mathsf{mtype}(m, C) = T \to U$, and $\mathsf{mbody}(m, C, \texttt{CT}') = (x, e)$. This rule means we can fetch a closure of the mobile method body from the remote site safely.

Mobile Method Bodies

(rm) $l[E[\texttt{await}\ c] \,|\, P, \sigma_l] \,|\, m[(\nu\, \vec{x})(R \,|\, Q, \sigma_m \cdot [\vec{x} \mapsto \vec{v}])]$

$\quad \mapsto l[(\nu\, \vec{x})(E[\texttt{await}\ c] \,|\, R \,|\, P, \sigma_l \cdot [\vec{x} \mapsto \vec{v}], \texttt{CT}_l)] \,|\, m[Q, \sigma_m]$

where $\mathsf{Mobile}_\Gamma(e[\vec{v}/\vec{x}])$, $\mathsf{Mobile}_\Gamma(v_i)$, $R \equiv e[\texttt{return}(c)/\texttt{return}]$ and $\vec{x} \notin \mathsf{fv}(Q)$. This

rule says a mobile thread with a store can move from the remote site safely.

Synchronisation

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

(sync) $l[P \mid E[\texttt{insync}\ o\ \{e\}] \mid P, \sigma_2, \texttt{CT}] \mapsto (\nu\, \vec{u})l[E[\texttt{insync}\ o\ \{e'\}] \mid P, \sigma_2', \texttt{CT}']$
if $l[E[e] \mid P, \sigma_1, \texttt{CT}] \longrightarrow (\nu\, \vec{u})l[E[e'] \mid P, \sigma_1', \texttt{CT}']$,
$\sigma_2 = \mathsf{setLock}(\sigma_1, o, \mathsf{getLock}(\sigma, o) + 1)$, $\sigma_2' = \mathsf{setLock}(\sigma_1', o, \mathsf{getLock}(\sigma, o) + 1)$

The reduction under synchronisation is deterministic if there is no wait, notify and notifyall in the program.

Deterministic Rule

(ni) $N \longrightarrow N' \implies N \mapsto N'$

if the last reduction rule applied was not generated by variable read (Var), variable assignment (Ass), field access (Fld), field assignment (FldAss), freezing (Freeze), the signal to a waiting thread (Notify), monitor entry (Sync) or the reduction for reacquiring a monitor (Ready). The transformation rule $N \mapsto N'$ is defined as a binary relation generated by the above rules closed under parallel composition, restriction and structure rules (as in Figure 11).

Theorem 7.1.

(*1*) **(noninterference)** $\mapsto$ *satisfies a noninterference property and respects the observational predicate under a network invariant.*

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

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

Proof. (1) is mechanical by investigating each of the above rules in turn. The most interesting rule is (mi), which is derived by (ni, rm, l2). (sync) is by Propositon 6.2 (1). (2) is straightforward by noting that $\longrightarrow$ is a typed relation, and that transformed terms can be related by it. (3) uses (1) and (2) together with Lemma 7.1. □

Note that by Definition 6.1, Theorem 7.1 excludes the error statement. 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 7.1.

(*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 7.1 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 G for the encoding and proofs. □

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

Definition 7.5. Let us write $c \rightsquigarrow c'$ if there exists a chain of channels such that

$$\texttt{return}(c)\ E_1[\texttt{await}\ c_1] \mid \texttt{return}(c_1)\ E_2[\texttt{await}\ c_2] \cdots \mid \texttt{return}(c_n)\ E_n[\texttt{await}\ c']$$

Suppose

$$N \twoheadrightarrow (\nu\,\vec{u})(l_1[\texttt{go}\ o.m(\vec{v})\ \texttt{with}\ c \mid E[\texttt{await}\ c] \mid P, \sigma_1, \texttt{CT}_1] \mid l_2[Q, \sigma_2, \texttt{CT}_2] \mid M) \stackrel{\text{def}}{=} N'$$

with $o \in \mathsf{dom}(\sigma_2)$. We say there exists *a call-back from* l_1 *to* l_2 if we have:
$N' \twoheadrightarrow (\nu\,\vec{u})(l_1[E[\texttt{await}\ c] \mid P', \sigma_1', \texttt{CT}_1'] \mid l_2[\texttt{go}\ o'.m'(\vec{v'})\ \texttt{with}\ c' \mid Q', \sigma_2', \texttt{CT}_2'] \mid M')$
where $o' \in \mathsf{dom}(\sigma_1')$ and there exists $c \rightsquigarrow c'$ in Q'.

The definition means: suppose "$E[\texttt{await}\ c]$" is created in l_1 by the remote method invocation "$o.m$" to l_2. Then the computation of $E[\texttt{await}\ c]$ is blocked until some method invocation "$o'.m'$" returned from l_2. 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. Note that there is no guarantee that the caller and the call-back can be synchronised correctly in a naïve program, as they cannot hold the same lock (see § 5.3.1). For this reason, since most RMI programs do not use call-backs, we do not investigate them.

7.3 Correctness of the 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 $o.\texttt{mOpt1}(r, n)\ \texttt{with}\ c$,

$$(\nu\,a)(\texttt{thunk}\langle\texttt{int}\rangle\ t = \texttt{freeze}[\texttt{lazy}]\{e; z\}; \texttt{return}(c)\ r.\texttt{run}(t), [a \mapsto n])$$

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

$$\begin{array}{lll}
\mapsto & (\nu\,a)(\texttt{thunk}\langle\texttt{int}\rangle\ t = v; \texttt{return}(c)\ r.\texttt{run}(t), [a \mapsto n]) & \text{(fr)} \\
\mapsto^+ & (\nu\,t)(\texttt{return}(c)\ r.\texttt{run}(t), [t \mapsto v]) & \text{(ni)} \\
\mapsto & \texttt{return}(c)\ r.\texttt{run}(v), \emptyset & \text{(cr)} \\
\mapsto & \texttt{return}(c)\ \texttt{defrost}(v;\ \lambda(T\ x).(l, \texttt{defrost}(x), \emptyset, \emptyset)), \emptyset & \text{(mi)} \\
\mapsto^+ & (\nu\,a)(\texttt{return}(c)\ \texttt{sandbox}\ \{e; z\}, [a \mapsto n]) & \text{(ni)} \\
\mapsto & (\nu\,a)(e; \texttt{return}(c)\ z, [a \mapsto n]) & \text{(l1)}
\end{array}$$

The last line is identical to (RMI1) after the method invocation by $o.\texttt{m1}(r, n)\ \texttt{with}\ c$. 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 $\mathsf{Mobile}_{\emptyset}(v)$, we can apply (mi) in the forth 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). Then we apply a similar transformation with the above to derive (RMI3). See Appendix H 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:

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

Finally by Proposition 7.1 (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 7.1 (1) is not applicable. To summarise, we have:

Theorem 7.2. **(Correctness of the Optimisations)**

(*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 [Liang and Bracha 1998; Qian et al. 2000]. Our formulation of class downloading is modular, so it is adaptable to model other linking strategies [Drossopoulou and Eisenbach 2002; Drossopoulou et al. 2003], see § 4.2. We set the class invariant $\mathsf{Inv}(3)$ in Definition 6.2. 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 [Krintz et al. 1999] propose a class splitting and pre-fetching 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 [Cardelli 1994] is a distributed object-based, 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 [Merro et al. 2002] 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 [Nestmann et al. 2002] formalised a typing system for a core Obliq calculus and studied different kinds of object aliasing. Briais and Nestmann [Briais and Nestmann 2002] then strengthened the safety result in [Merro et al. 2002] 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 [Cardelli 1994] refers to one in the sense 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 [Hutchinson et al. 1991] 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 [Gordon and Hankin 1999] extend the object calculus [Abadi and Cardelli 1996] 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 [Jeffrey 2000] treats an extension of [Gordon and Hankin 1999] 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 [Jeffrey 2000] treats transactional serialisation as in [Cardelli 1994]), which are among the key elements for analysis of RMI and code mobility in Java.

Scope and runtime formalisms for Java. Zhao et al [Zhao et al. 2004] 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 6.2 ensures that identifiers for local 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 [Gordon and Syme 2001] and Yu et al [Yu et al. 2004].

The JavaSeal [Vitek et al. 1998] 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 [Ohori and Kato 1993] 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 [Acute 2005] 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 [Taha and Sheard 1997] 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 [Kamin et al. 2003] 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 [Zook et al. 2004] 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 runtime for 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 soundness 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 [Bogle and Liskov 1994; Yeung and Kelly 2003; Yeung 2004], 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 [Yeung and Kelly 2003; Yeung 2004] cannot aggregate code in which new object generation is inserted, such as:

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

```
5   return z;
6 }
```

where `MyObj` is a local 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 [Yeung and Kelly 2003; Yeung 2004], 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 [Hutchinson et al. 1991; Cardelli 1994], 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 [Igarashi et al. 2001; Bierman et al. 2003], exceptions [Ancona et al. 2002] and parametric polymorphism [Igarashi et al. 2001]; although these features can be represented by extension of the present syntax and types, their precise interplay with distributed language constructs requires examination.

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 [McGraw and Morrisett 2000; Bogle and Liskov 1994]. 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 [Wallach et al. 1997]. 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 [Yoshida 2004].

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* [Honda et al. 1998; Vasconcelos et al. 2004] 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 [Microsystems Inc. 2005]) 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 [Tejani 2005]. 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-preoder theory developed for process algebra [Arun-Kumar and Hennessy 1992] to compare program performance.

Acknowledgements. We deeply thank Paul Kelly for his discussions about the Veneer vJVM and the RMI optimisations in [Yeung and Kelly 2003; Yeung 2004]. Comments from the anonymous reviewers helped to revise the submission. 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 and EPSRC GR/T04724/01.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag.

ACUTE. 2005. Acute home page. http://www.cl.cam.ac.uk/users/pes20/acute.

AHERN, A. AND YOSHIDA, N. 2005a. Formal Analysis of a Distributed Object-Oriented Language and Runtime. Tech. Rep. 2005/01, Department of Computing, Imperial College London: Available at: http://www.doc.ic.ac.uk/~aja/dcbl.html.

AHERN, A. AND YOSHIDA, N. 2005b. Formalising java rmi with explicit code mobility. In *OOP-*

SLA '05 (to appear), the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM Press.

Ancona, D., Lagorio, G., and Zucca, E. 2002. Simplifying types in a calculus for Java exceptions. Tech. rep., DISI - Università di Genova.

Arun-Kumar, S. and Hennessy, M. 1992. An efficiency preorder for processes. *Acta Inf. 29,* 8, 737–760.

Bierman, G., Parkinson, M., and Pitts, A. 2003. MJ: An imperative core calculus for Java and Java with effects. Tech. Rep. 563, University of Cambridge Computer Laboratory. April.

Bogle, P. and Liskov, B. 1994. Reducing cross domain call overhead using batched futures. In *OOPSLA '94*. ACM Press, 341–354.

Briais, S. and Nestmann, U. 2002. Mobile objects "must" move safely. In *FMOODS 2002*. Kluwer Academic Publishers, 129–146.

Cardelli, L. 1994. Obliq: A language with distributed scope. Tech. Rep. 122, Systems Research Center, Digital Equipment Corporation.

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

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

Drossopoulou, S., Lagorio, G., and Eisenbach, S. 2003. Flexible Models for Dynamic Linking. In *12th European Symposium on Programming*, P. Degano, Ed. Lecture Notes in Computer Science, vol. 2618. Springer-Verlag, 38–53.

Flanagan, D. 2000. *Java Examples in a Nutshell*. O'Reilly UK.

Gordon, A. D. and Hankin, P. D. 1999. A concurrent object calculus: Reduction and typing. Tech. Rep. 457, University of Cambridge Computer Laboratory. February.

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

Greanier, T. 2005. Discover the secrets of the Java Serialization API. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

Honda, K. 1996. Composing processes. In *Proceedings of POPL'96*. 344–357.

Honda, K. and Tokoro, M. 1991. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*.

Honda, K., Vasconcelos, V. T., and Kubo, M. 1998. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*. Lecture Notes in Computer Science, vol. 1381. 22–138.

Honda, K. and Yoshida, N. 1995. On reduction-based process semantics. *TCS 151,* 2, 385–435.

Hutchinson, N. C., Raj, R. K., Black, A. P., Levy, H. M., and Jul, E. 1991. The Emerald Programming Language. Tech. rep., Department of Computer Science, University of British Columbia, Vancouver BC, Canada V6T 1Z2. October.

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

Jeffrey, A. 2000. A Distributed Object Calculus. In *FOOL*. ACM Press.

Jones, C. 1993. Constraining interference in an object-based design method. In *Proceedings of TAPSOFT'93*. LNCS 668. Springer Verlag, 136–150.

Kamin, S., Clausen, L., and Jarvis, A. 2003. Jumbo:run-time code generation for java and its applications. In *CGO03*. IEEE.

Kobayashi, N., Pierce, B., and Turner, D. 1996. Linear types and π-calculus. In *Proceedings of POPL'96*. 358–371.

Krintz, C., Calder, B., and Hölzle, U. 1999. 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*. ACM Press, 276–291.

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

McGraw, G. and Morrisett, G. 2000. Attacking malicious code: a report to the infosec research council. *IEEE Software 17,* 5, 33–44.

Merro, M., Kleist, J., and Nestmann, U. 2002. Mobile objects as mobile processes. *Information and Computation 177,* 2, 195–241.

Microsystems Inc., S. 2005. Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi/.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, parts I and II. *Info. & Comp. 100,* 1.

Nestmann, U., Hüttel, H., Kleist, J., and Merro, M. 2002. Aliasing models for mobile objects. *Inf. Comput. 177,* 2, 195–241.

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

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

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

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

Sangiorgi, D. 1992. Expressing mobility in process algebras: First-order and higher order paradigms. Ph.D. thesis, University of Edinburgh.

Stamos, J. W. and Gifford, D. K. 1990. Implementing remote evaluation. *IEEE Trans. Softw. Eng. 16,* 7, 710–722.

Taha, W. and Sheard, T. 1997. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM Press, 203–217.

Tejani, F. MEng. Computing Final Year Project, Imperial College London, to appear in June, 2005. Implementation of a distributed mobile java. M.S. thesis, Imperial College London.

Vasconcelos, V. T., Ravara, A., and Gay, S. 2004. Session types for functional multithreading. In *CONCUR'04*. Lecture Notes in Computer Science, vol. 3170. 497–511.

Vitek, J., Bryce, C., and Binder, W. 1998. Designing JavaSeal or how to make Java safe for agents. *Electronic Commerce Objects*.

Wallach, D. S., Balfanz, D., Dean, D., and Felten, E. W. 1997. Extensible security architectures for Java. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*. ACM Press, 116–128.

Yeung, K. 2004. Dynamic performance optimisation of distributed java applications. Ph.D. thesis, Imperial College London.

Yeung, K. and Kelly, P. 2003. Optimizing Java RMI programs by communication restructuring. In *Middleware'03*. Lecture Notes in Computer Science, vol. 2672. 324–343.

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

Yoshida, N., Berger, M., and Honda, K. 2001. Strong Normalisation in the π-Calculus. In *Proc. LICS'01*. IEEE, 311–322. The full version in *Journal of Inf. & Comp..*, 191 (2004) 145–202, Elsevier.

Yu, D., Kennedy, A., and Syme, D. 2004. 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*. ACM Press, 39–51.

Zhao, T., Noble, J., and Vitek, J. 2004. Scoped types for real-time Java. In *Proceedings of the 25th Annual IEEE Symposium on Real-time Systems*.

Zook, D., Huang, S. S., and Smaragdakis, Y. 2004. Generating AspectJ Programs with Meta-AspectJ. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering*. Lecture Notes in Computer Science, vol. 3286. Springer-Verlag GmbH, 1–19.

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.

$$\mathsf{dom}(\emptyset) = \emptyset,\ \mathsf{dom}(\sigma \cdot [x \mapsto \dots]) = \mathsf{dom}(\sigma) \cup \{x\},\ \mathsf{dom}(\sigma \cdot [o \mapsto \dots]) = \mathsf{dom}(\sigma) \cup \{o\}$$
$$\mathsf{dom}(\mathtt{CT} \cdot [C \mapsto \dots]) = \mathsf{dom}(\mathtt{CT}) \cup \{C\}$$
$$\mathsf{dom}(\mathsf{CSig} \cdot [C \mapsto \dots]) = \mathsf{dom}(\mathsf{CSig}) \cup \{C\}$$
$$\mathsf{dom}((\nu\, \vec{u})(P, \sigma, \mathtt{CT})) = \mathsf{dom}(\sigma) \setminus \{\vec{u}\}$$
$$\mathsf{dom}(\mathbf{0}) = \emptyset,\ \mathsf{dom}(l[F]) = \mathsf{dom}(F),\ \mathsf{dom}(N_1 \mid N_2) = \mathsf{dom}(N_1) \cup \mathsf{dom}(N_2)$$
$$\mathsf{dom}((\nu\, u)N) = \mathsf{dom}(N) \setminus \{u\}$$

A.2 Free variables and names

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

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

For values:

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

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.

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

(Configurations)

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

(Threads)

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

(Networks)

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

Fig. 16: Structural equivalence

For threads:

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

For networks:

$\mathbf{0}$	$\emptyset$	$\emptyset$
$l[F]$	$\mathsf{fv}(F)$	$\mathsf{fn}(F)$
$N_1 \mid N_2$	$\bigcup \mathsf{fv}(N_i)$	$\bigcup \mathsf{fn}(N_i)$
$(\nu\, u)N$	$\mathsf{fv}(N) \setminus \{u\}$	$\mathsf{fn}(N) \setminus \{u\}$
$\emptyset$	$\emptyset$	$\emptyset$
$\sigma \cdot [x \mapsto v]$	$\{x\} \cup \mathsf{fv}(v) \cup \mathsf{fv}(\sigma)$	$\mathsf{fn}(v) \cup \mathsf{fn}(\sigma)$
$\sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$	$\mathsf{fv}(\vec{v}) \cup \mathsf{fv}(\sigma)$	$\{o, \vec{c}\} \cup \mathsf{fn}(\vec{v}) \cup \mathsf{fn}(\sigma)$
$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{CT} \cdot [C \mapsto L]$	$\mathsf{fv}(L) \cup \mathsf{fv}(\mathtt{CT})$	$\mathsf{fn}(L) \cup \mathsf{fn}(\mathtt{CT})$

B. STRUCTURAL EQUIVALENCE

This section defines the structural equivalences for DJ. They are defined for threads, networks and configurations in Figure 16. Formally, $\equiv$ is an equivalence relation which includes α-conversion and is generated by the equations in Figure 16.

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 [Milner et al. 1992], and so are standard.

C. PROOFS FOR THE CORRECTNESS OF THE ALGORITHMS

This section lists the proofs for Lemma 4.1. We show (1, 2) by induction on the length of σ, and (3, 4) by induction on the length of $\mathtt{CT}$.

In the subsequent proofs, we use *length functions* for stores and class tables defined by:

$$\text{length}(\emptyset) = 0 \qquad \text{length}(\sigma \cdot [o \mapsto (C, \ldots)]) = \text{length}(\sigma \cdot [x \mapsto v]) = \text{length}(\sigma) + 1$$

$$\text{length}(\emptyset) = 0 \quad \text{length}(\mathtt{CT} \cdot [C \mapsto \mathtt{class}\ C\ \mathtt{extends}\ D\ \{\vec{T}\vec{f};\ K\ \vec{M}\}]) = \text{length}(\mathtt{CT}) + 1$$

(1) By induction on length(σ). For the base case assume length(σ) = 0. Then $\sigma = \emptyset$ and $\Gamma; \emptyset \vdash \emptyset : \mathtt{ok}$ immediately. The two conjunctions hold vacuously, therefore this case is complete.

For the inductive case, we assume that the property holds for length(σ) $< n$ and prove for length(σ) $= n$. Examining the algorithm, there are two sub-cases. If $v \notin \mathsf{dom}(\sigma)$ or $v \in \mathsf{dom}(\sigma)$ and v is an instance of a class C such that $\mathsf{remote}(C)$, then $\sigma' = \emptyset$. Then, as in the base case, the property holds straightforwardly.

Now for the second sub-case, i.e. $\mathsf{og}(\sigma, o) = [v \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)] \cup \mathsf{og}(\sigma_i, o_i)$, examining Algorithm 4.1, we have:

$$\sigma(v) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\}) \text{ with } \mathsf{local}(C),\ \{\vec{o}\} = \mathsf{fn}(\vec{v}) \tag{5}$$

$$\sigma_1 = \sigma \setminus \{v\} \tag{6}$$

$$\sigma_{i+1} = \sigma_i \setminus \mathsf{dom}(\mathsf{og}(\sigma_i, o_i)) \tag{7}$$

From because $v \in \mathsf{dom}(\sigma)$ and (6) we have that length(σ_1) $< n$. Examining (7), length(σ_i) $\leq$ length(σ_1) $<$ length(σ) and so by the inductive hypothesis:

$$\Gamma; \emptyset \vdash (\sigma'_i =)\mathsf{og}(\sigma_i, o_i) : \mathtt{ok}, \quad \forall o' \in \mathsf{dom}(\sigma'_i).\ \sigma'_i(o') = (C, \ldots) \text{ with } \mathsf{local}(C)$$
$$\forall o' \in (\mathsf{fn}(\sigma'_i) \setminus \mathsf{dom}(\sigma'_i)).\ \Gamma \vdash o' : C \text{ with } \mathsf{remote}(C)$$

By applying Lemma D.2 (7), and by set-union we have:

$$\Gamma; \emptyset \vdash (\sigma'' =)\bigcup \mathsf{og}(\sigma_i, o_i) : \mathtt{ok}, \quad \forall o' \in \mathsf{dom}(\sigma'').\ \sigma''(o') = (C, \ldots) \text{ with } \mathsf{local}(C)$$
$$\forall o' \in (\mathsf{fn}(\sigma'') \setminus \mathsf{dom}(\sigma'')).\ \Gamma \vdash o' : C \text{ with } \mathsf{remote}(C)$$

By typability of σ, we have that $\Gamma; \Delta \vdash (C, \vec{f} : \vec{v}, n, \{\vec{c}\}) : \mathtt{ok}$. Therefore we can straightforwardly deduce: $\Gamma; \emptyset \vdash (C, \vec{f} : \vec{v}, 0, \emptyset) : \mathtt{ok}$, and by examination of (5), $\{\vec{o}\} = \mathsf{fn}(\vec{v})$ and any identifiers in $\vec{o}$ that point to instances of local classes will have been gathered by recursive application of the algorithm, and hence in $\mathsf{dom}(\sigma'')$. Thus we finish the case.

(2) Assume $\sigma' = \mathsf{og}(\sigma, v)$. The base case is where length(σ) $= 0$ i.e. $\sigma = \emptyset$. Examining the algorithm we see that $\sigma' = \emptyset$. Therefore trivially $\mathsf{ogcomp}(\sigma, \sigma')$.

For the inductive step, assume that $\sigma' = \mathsf{og}(\sigma, v)$ and $\mathsf{ogcomp}(\sigma, \sigma')$ for length(σ) $<$ n. Now setting length(σ) $= n$ there are two sub-cases:

(a) $\sigma' = \emptyset$. Trivially, there are no pairs in the reachability relation $RCH(\sigma')$ and so we have $\mathsf{ogcomp}(\sigma, \sigma')$ vacuously.

(b) $\sigma' = [v \mapsto \sigma(v)] \bigcup \mathsf{og}(\sigma_i, o_i)$, where $\sigma(v) = (C, \vec{f} : \vec{v})$, $\{\vec{o}\} = \mathsf{fn}(\vec{v})$, $\sigma_1 = \sigma \setminus \{o\}$ and $\sigma_{i+1} = \sigma_i \setminus \mathsf{dom}(\mathsf{og}(\sigma_i, o_i))$.

Clearly, length(σ_i) $< n$ by the initial removal of o from σ_1. Write $\sigma'_i = \mathsf{og}(\sigma_i, o_i)$. Further examination of the algorithm shows that σ'_{i+1} is computed from σ_i less the elements collected in σ'_i. Therefore $\mathsf{dom}(\sigma'_i) \cap \mathsf{dom}(\sigma'_{i+1}) = \emptyset$ so by the inductive hypothesis $\mathsf{ogcomp}(\sigma_1, \bigcup \mathsf{og}(\sigma_i, o_i))$. Recall that $\sigma_1 = \sigma \setminus \{v\}$. By adding $[v \mapsto \sigma(v)]$ to each side, we add the same number of reachable states and therefore $\mathsf{ogcomp}(\sigma, \sigma')$.

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

(4) Suppose length($\mathtt{CT}_0$) $= 0$ then by definition, $\mathtt{CT}'$ is complete. Now, take 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 soundness 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 D.1 lists some useful properties about judgements. We write $\mathtt{J}$ to stand for any one of the following judgements:

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

Lemma D.1linearity 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 $\mathtt{chan}$.

LEMMA D.1. (Judgements)

(Permutation of environments)
(*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* $\mathtt{this}$.

(Linearity of channels)
(*3*) $\Gamma; \Delta, c : \tau, \Delta' \vdash \mathtt{J} \wedge c \notin \mathsf{fn}(\mathtt{J}) \implies \tau = \mathtt{chan}$.

(Weakening)
(*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*) $\Gamma; \Delta \vdash \mathtt{J} \wedge \vdash C : \mathtt{tp} \wedge \mathtt{this} \notin \mathsf{dom}(\Gamma) \implies \Gamma, \mathtt{this} : C; \Delta \vdash \mathtt{J}$.

(Strengthening)

(7) $\Gamma; \Delta, c : \tau \vdash \mathtt{J} \wedge c \notin \mathsf{fn}(\mathtt{J}) \implies \Gamma; \Delta \vdash \mathtt{J}$.
(8) $\Gamma, u : T; \Delta \vdash \mathtt{J} \wedge u \notin \mathsf{fnv}(\mathtt{J}) \implies \Gamma; \Delta \vdash \mathtt{J}$.

(Implied judgements)
(9) $\Gamma, \Gamma'; \Delta, \Delta' \vdash \mathtt{J} \implies \Gamma; \Delta \vdash \mathtt{Env}$.

Proof. By induction on the size of the judgement 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 $\Gamma; \Delta_1, c : \mathtt{chan} \vdash P_1 : \mathtt{thread}$ and $\Delta_1, c : \mathtt{chan} \asymp \Delta_2$ as $c \notin \mathsf{dom}(\Delta_2)$. Apply TT-Par to yield $\Gamma; \Delta_1, c : \mathtt{chan} \odot \Delta_2 \vdash P_1 \mid P_2 : \mathtt{thread}$. Then $\Gamma; \Delta_1 \odot \Delta_2, c : \mathtt{chan} \vdash P_1 \mid P_2 : \mathtt{thread}$ by definition of $\odot$. The other case proceeds similarly. □

D.2 Stores

Lemma D.2 states properties about the type-safety of store access. Store access 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 D.2 allows the concatenation of disjoint stores and is useful in typing the $\mathtt{deserialize}(e)$ operation.

Lemma D.2. (Stores) *Assuming that* $\Gamma; \Delta \vdash \sigma : \mathtt{ok}$. *Then:*

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

Proof. All are mechanical. □

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

Lemma D.3. (Method body) *Suppose* $\mathsf{mbody}(m, C, \mathtt{CT}) = (x, e)$ *and* $\mathsf{mtype}(m, C) = D \to U$ *with* $\vdash \mathtt{CT} : \mathtt{ok}$. *Then for some* C' *where* $C <: C'$ *and some* U' *where* $U' <: U$ *then we have* $x : D, \mathtt{this} : C' \vdash e : \mathtt{ret}(U')$.

D.3 Structural equivalence: Proof of Lemma 6.2

An important property to be shown is that the application of the structural equality rules given in Figure 16 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 D.4. *(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 6.3

(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. $\mathsf{comp}(C, \texttt{CT}_{im})$, hence by Lemma 6.3 (1) we have $\mathsf{comp}(C, \texttt{CT}_{im+1})$.
ii. $P_{im} \equiv E[\texttt{download}\,\vec{C}\,\texttt{from}\,l_j\,\texttt{in}\,e] \,|\, Q_{im}$ or $P_{im} \equiv E[\texttt{resolve}\,\vec{C}\,\texttt{from}\,l_j\,\texttt{in}\,e] \,|\, Q_{im}$, 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 $\mathsf{comp}(C, \texttt{CT}_{im+1})$. If the last reduction rule applied was RESOLVE, then we see that $P_{im+1} \equiv E[\texttt{resolve}\,\vec{C}'\,\texttt{from}\,l_j\,\texttt{in}\,e] \,|\, Q_{im}$ 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 $P_{im+1} \equiv o.m(\texttt{deserialize}(v))\ \texttt{with}\ c \,|\, Q_{im+1}$. Since o moved from another location, we can conclude $\mathsf{remote}(C)$, hence it must have been created at location l_i by NEW, or was there in the initial network (recall that remote object identifiers cannot leak to other locations, nor can their store entry be carried in a frozen value).

In the case of NEW, we had that for some step k (with $N_0 \rightarrow\!\!\!\!\rightarrow N_k \rightarrow\!\!\!\!\rightarrow N_m$) when o was created, $\mathsf{comp}(C, \texttt{CT}_{ik})$. Then by Lemma 6.3 (1), $\mathsf{comp}(C, \texttt{CT}_{im+1})$. If o was

present in the initial network, then by initial conditions and application of this Lemma again, $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.

When o becomes reachable, all objects that it points to (both directly and indirectly) now become reachable. Suppose without loss of generality that $[o \mapsto (C, f : o')] \in \sigma_{im}$, i.e. we only have one field, that points to a newly reachable object o'. If $o' \notin \mathsf{dom}(\sigma_{im})$, then it must be remote, and so the property holds immediately.

If $o' \in \mathsf{dom}(\sigma_{im})$, we must investigate further. Suppose there exists some step k such that $P_{ik} \equiv E[\mathtt{new}\ C(o')] \,|\, Q_{ik} \longrightarrow_{l_i} E[o] \,|\, Q_{ik+1}$. Since o' is a non-remote identifier, we have that it was either instantiated by New (hence the class it belongs to is complete), or it is the result of a defrost operation. By examining the rule Defrost, before this assignment could take place, we must have downloaded all superclasses of o''s class, hence it must be complete. Similarly if $P_{ik} \equiv E[o.f = o'] \,|\, Q_{ik}$.

If o was added to the current location by Defrost, then by the completeness of the object graph of the frozen object, we had that o' was also added in the same step. Hence after downloading all superclasses of o, we must have also downloaded all superclasses of o'.

(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 $\mathsf{remote}(C)$. Since remote object identifiers 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 Download or New. In the case of the former, we see that $P_{im} \equiv E[\mathtt{defrost}(v;\ \ldots)] \,|\, Q_{im}$ reduces to $P_{im+1} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im+1}$ as required. In the case of the latter, we had $\mathsf{comp}(C, \mathtt{CT}_{im})$ and so by Lemma 6.3 (1) $\mathsf{comp}(C, \mathtt{CT}_{im+1})$.

(4) Obvious by repeating Download and Resolve until we reach resolution of C'. Note that these reductions terminate as the inheritance relations in a well-formed class table is acyclic.

E.2 Proofs of Lemma 6.4

Induction on k. Below we assume $c \in \mathsf{blocked}(o, \sigma_{im+1})$ in $\mathtt{waiting}(c)\ \ldots.$

(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, $\mathsf{getLock}(\sigma_{i0}, o) = 0$ and so by application of Sync in we have that $\mathsf{getLock}(\sigma_{i1}, o) = 1$ 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:

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

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

$E_p[\texttt{insync}\ o\ \{e_p\}]$.

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 $\mathsf{getLock}(\sigma_{im}, o) = p+1$. By premise of LeaveCritical $\sigma_{im+1} = \mathsf{setLock}(\sigma_{im}, o, p)$, and so $\mathsf{getLock}(\sigma_{im+1}, o) = p$ 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 $\mathsf{getLock}(\sigma_{im}, o) = p-1$. Then by the premise of Sync we see that $\sigma_{im+1} = \mathsf{setLock}(\sigma_{im}, o, p)$, and so $\mathsf{getLock}(\sigma_{im+1}, o) = p$ 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: $P_{i0}, \sigma_{i0}, \mathtt{CT}_{i0} \longrightarrow_l P_{i1}, \sigma_{i1}, \mathtt{CT}_{i1}$. By the initial conditions $\mathsf{getLock}(\sigma_{i0}, o) = 0$, and by assumption $\mathsf{getLock}(\sigma_{i1}, o) = n$ with $n > 0$. As no run-time syntax can exist in the network initially, the reduction rule applied was Sync. This means that $P_{i0} \equiv E[\texttt{sync}\ (o)\ \{e\}] \,|\, Q_{i0}$, and examining the conclusion of the rule $P_{i1} \equiv E[\texttt{insync}\ o\ \{e\}] \,|\, Q_{i1}$. Again by the initial conditions, e cannot contain $\texttt{insync}\ o\ \{\dots\}$ as a sub-term, completing this case.

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

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

$$P_{im} \equiv E_1[\texttt{insync}\ o\ \{\dots E_x[\texttt{insync}\ o\ \{E[\texttt{ready}\ o\ x]\}]\dots\}] \,|\, Q_{im}$$

with $\texttt{insync}\ o\ \{\dots\}$ 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) $\mathsf{getLock}(\sigma_{im}, o) = p-1$. The only rule applicable in this situation is Sync. Therefore by the inductive hypothesis:

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

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

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

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

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

(4) $\mathsf{getLock}(\sigma_{im}, o) = p+1$. Only one rule is applicable in this situation: LeaveCritical. To apply this, P_{im} must be $E_1[\texttt{insync}\ o\ \{\dots E_{p+1}[\texttt{insync}\ o\ \{e\}]\dots\}] \,|\, Q_{im}$ with $e = v$ or $e = \texttt{return}(c)\ v$. We consider the case for v, the case for a return is similar. Examining LeaveCritical, P_{im+1} must be $E_1[\texttt{insync}\ o\ \{\dots E_{p+1}[v]\dots\}] \,|\, Q_{im+1}$ with $\texttt{insync}\ o\ \{\dots\}$ not a sub-term of E_{p+1} by our earlier assumption, completing this case. □

E.3 Proofs of the Invariant properties

Inv(1) By Lemma 6.3 (1).

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 two 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 Inv(2)′ and Lemma 6.3 (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 6.3 (1) it is the case that $\mathsf{comp}(C, \texttt{CT}_{im+1})$.

(b) We applied DNothing. Then $P_{im} \equiv E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_j\ \texttt{in}\ \texttt{new}\ C(\vec{v}) \,|\, Q_{im}$ 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 6.3 (1) we have trivially that $\mathsf{comp}(C, \texttt{CT}_{im+1})$.

Inv(3) There are two interesting sub-cases:

(a) The last applied reduction rule was Download. Then

$$\begin{aligned} & P_{im} \equiv E[\texttt{download}\ C\ \texttt{from}\ l_j\ \texttt{in}\ e] \,|\, Q_{im} \\ \text{and} \quad & P_{im} \longrightarrow_{l_i} E[\texttt{resolve}\ C\ \texttt{from}\ l_j\ \texttt{in}\ e] \end{aligned}$$

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 the premise of Download, $C \in \mathsf{dom}(\texttt{CT}_{im+1})$ and $\texttt{CT}_{im+1}(C) = \texttt{CT}_{jm}$ modulo class 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.

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 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

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

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 $\mathsf{fv}(e[o/\texttt{this}][\texttt{return}(c)/\texttt{return}]) = \mathsf{fv}(e)$. Given that the network configuration is well-typed, it must be the case that $x : D, \texttt{this} : C \vdash e : \texttt{ret}(U)$, i.e. $\mathsf{fv}(e) \subseteq \{x\}$. This concludes the case.

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

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

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 $\mathsf{Inv}(15)$, frozen values contain only remote identifiers, hence the rules METHREMOTE and LEAVE cannot move shared store entries across the network.

$\mathsf{Inv}(6)$ For the first case suppose that:

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

Then by $\mathsf{Inv}(5)$ we have that o cannot exist as an entry in 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 "remoteness" of a class, $\mathsf{remote}(C)$ holds, completing the case.
For the second case suppose that:

$$o \in \mathsf{fn}(F_{im+1}) \cap \mathsf{fn}(F_{jm+1}), \quad o \notin \mathsf{fn}(F_{im}) \cap \mathsf{fn}(F_{jm}) \tag{8}$$

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 (8) 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 remote classes, hence $\mathsf{remote}(C)$. This completes the case.

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

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

and we shall prove that $k = i$.

(a) Suppose $o \in \mathsf{fn}(F_{im}) \wedge \exists k'.\sigma_{k'm}(o) = (C, \ldots) \wedge \mathsf{local}(C) \wedge k' = i$. 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.

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

Then by $\mathsf{Inv}(15)$, the frozen value can contain no free local object identifiers 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 conciseness)

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

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 *remote* object identifiers. 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 16, 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 $P_{im+1} \equiv o.m(e)\ \texttt{with}\ c \,|\, Q_{im+1}$. 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 6.3 (3) we have $\mathsf{comp}(C, \texttt{CT}_{im})$. By monotonicity of stores and class tables, $\sigma_{im+1}(o) = (C, \ldots)$ and $\mathsf{comp}(C, \texttt{CT}_{im+1})$ 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 $\mathsf{comp}(C, \texttt{CT}_{im})$, hence by Lemma 6.3 (1) $\mathsf{comp}(C, \texttt{CT}_{im+1})$.

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 $\mathsf{local}(C)$. Then by $\mathsf{Inv}(8)$ and $\mathsf{Inv}(7)$ we have that $\sigma_{im}(o) = (C, \ldots)$. Then by Lemma 6.3 (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 $P_{im} \equiv E[x.f = e] \mid Q_{im}$. By typability, $\Gamma, \vec{u} : \vec{T} \vdash x.f : C$ with $\mathsf{local}(C)$. 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 6.3 (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 $P_{im} \equiv E[(x = o).f = e] \mid Q_{im}$. Then by typability, $\Gamma, \vec{u} : \vec{T} \vdash x = o : C$ for some C such that $\mathsf{local}(C)$. Hence by typability $\Gamma, \vec{u} : \vec{T} \vdash o : C'$ such that $C' <: C$. By our convention, it must be the case that $\mathsf{local}(C')$ 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. Then $P_{im} \equiv E[\mathtt{sandbox}\ \{o\}.f = e] \mid Q_{im}$. Then this is also straightforward, noting that $\Gamma, \vec{u} : \vec{T} \vdash o : C$ with $\mathsf{local}(C)$.

$\mathsf{Inv}(10)$ 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 $P_{im+1} \equiv E[v] \mid Q_{im+1}$, 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 $P_{im} \equiv E[\mathtt{freeze}[t](T\ x)\{e\}] \mid Q_{im}$. By reduction, $v = \lambda(T\ x).(\nu\, \vec{u})(l_i, e, \sigma, \mathtt{CT})$. Examining the definition of fv, we see that $\mathsf{fv}(v) = (\mathsf{fv}(e) \setminus \{x\}) \cup \mathsf{fv}(\sigma) \cup \mathsf{fv}(\mathtt{CT})$. 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, $\mathtt{CT}$ is derived from $\mathtt{CT}_{im}$ and so since $\vdash \mathtt{CT}_{im} : \mathtt{ok}$, we have that $\mathsf{fv}(\mathtt{CT}) = \emptyset$. 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 was DNOTHING. Then $P_{im} \equiv E[\mathtt{download}\, \vec{C}\ \mathtt{from}\, l_j\ \mathtt{in}\ \mathtt{sandbox}\ \{v\}] \mid Q_{im}$. 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 $P_{im} \equiv E[\mathtt{defrost}(v;\ \lambda(T\ x).(\nu\, \vec{u})(l_j, e, \sigma, \mathtt{CT}))] \mid Q_{im}$. However by $\mathsf{Inv}(12)$, we

have that the frozen value contains no free variables, hence $\mathsf{fv}(\sigma) = \emptyset$ 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. 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:

$$P_{im} \equiv E[\texttt{freeze}[\texttt{eager}](T\ x)\{e\}] \,|\, Q_{im} \longrightarrow_{l_i} P_{im+1} \quad \equiv E[\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma, \texttt{CT})] \,|\, Q_{im}$$

Then by the premises of Freeze we have:

$$\{\vec{y}\} = \mathsf{fv}(e) \setminus \{x\} \text{ with } \sigma_y = \bigcup \sigma_{im}(y_i), \quad \texttt{CT}' = \mathsf{cg}(\texttt{CT}, \mathsf{fcl}(e) \cup \mathsf{fcl}(\sigma))$$
$$\sigma = \mathsf{og}(\sigma_{im}, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y \text{ with } \{\vec{u}\} = \mathsf{dom}(\sigma)$$

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:$\mathsf{fn}(\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma, \texttt{CT})) = (\mathsf{fn}(e) \cup \mathsf{fn}(\sigma)) \setminus \{\vec{u}\}$. By Lemma 1 (1), and that $\{\vec{u}\} = \mathsf{dom}(\sigma)$:

$$\forall u \in (\mathsf{fn}(\mathsf{og}(\sigma_{im}, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y))) \setminus \{\vec{u}\}).\ \Gamma \vdash u : C \wedge \mathsf{remote}(C)$$

By Lemma 2 (2), all *local* object identifiers reachable from e or σ_y must be included in the computed object graph and hence in $\vec{u}$. Therefore all remaining free names in e or σ_y must point to remote objects. So:

$$\begin{aligned} &\forall u \in (\mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \setminus \{\vec{u}\}.\ \Gamma \vdash u : C \wedge \mathsf{remote}(C) \wedge (9) \\ \Longrightarrow\ &\forall u \in (\mathsf{fn}(e) \cup \mathsf{fn}(\sigma)) \setminus \{\vec{u}\}.\ \Gamma \vdash u : C \wedge \mathsf{remote}(C) \\ \Longrightarrow\ &\forall u \in \mathsf{fn}(\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma, \texttt{CT})).\ \Gamma \vdash u : C \wedge \mathsf{remote}(C) \end{aligned}$$

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

(a) Suppose $P_{im} \equiv E'[o.\texttt{notify}] \,|\, E[\texttt{waiting}(c)\ n] \,|\, Q_{im}$. 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 6.4 we have that $\mathsf{insync}(o, E)$ with n levels of nesting.

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

$\mathsf{Inv}(17)$ Assume $P_{im+1} \equiv E[\texttt{waiting}(c)\ n] \,|\, Q_{im+1}$. Then there is only one interesting case to consider, when $P_{im} \equiv E[o.\texttt{wait}] \,|\, Q_{im}$. By premise of Wait we have that $\mathsf{insync}(o, E)$, and consequently by Lemma 6.4, $n > 0$. 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 Proofs for progress with synchronisation primitives

This subsection proves Lemma 6.1.

By induction on the number of threads synchronised on the object o, written n. The base case is straightforward; take $n = 1$ then $P \equiv E_1[\texttt{insync}\ o\ \{e_1\}]\,|\,Q$. 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:

$$P \equiv E_1[\texttt{insync}\ o\ \{e_1\}]\,|\,\cdots\,|\,E_{n-1}[\texttt{insync}\ o\ \{e_{n-1}\}]\,|\,E_n[\texttt{insync}\ o\ \{e_n\}]\,|\,Q \quad (9)$$

We shall show that either:

$$\forall j.1 \leq j \leq n.\ (e_j = E'_j[\texttt{waiting}(c)\ o\ldots] \vee e_j = E'_j[\texttt{ready}\ o\ \ldots]) \quad (10)$$

$$\text{or}\quad \exists! j.1 \leq j \leq n.\ (e_j \neq E'_j[\texttt{waiting}(c)\ o\ldots] \wedge e_j \neq E'_j[\texttt{ready}\ o\ \ldots]) \quad (11)$$

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

$$\forall j.1 \leq j \leq n-1.\ (e_j = E'_j[\texttt{waiting}(c)\ o\ldots] \vee e_j = E'_j[\texttt{ready}\ o\ \ldots]) \quad (12)$$

$$\text{or}\quad \exists! j.1 \leq j \leq n-1.\ (e_j \neq E'_j[\texttt{waiting}(c)\ o\ldots] \wedge e_j \neq E'_j[\texttt{ready}\ o\ \ldots]) \quad (13)$$

For (12) if $e_n = \texttt{waiting}(c)\ o\ldots$ or $e_n = \texttt{ready}\ o\ \ldots$ then we can immediately conclude (10) as required. Similarly, if e_n is not of this form then we can safely conclude (11).

However, if we have the situation (13) 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 (11) 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:

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

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

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

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 $\mathsf{getLock}(\sigma, o) = 0$ as a premise. However by Lemma 6.4 we can conclude that $\mathsf{getLock}(\sigma, o) > 0$, 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, it is possible to 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 (11) as required. □

F. PROOFS FOR TYPE SOUNDNESS

F.1 Proofs for Substitution Lemma 6.5

By induction on the structure of the expression e using Lemma 6.2. 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 6.1

(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 Var. Use Lemma D.2 (3).

Case Cond. Straightforward by the inductive hypothesis.

Case While. Standard.

Case Fld. Straightforward by Lemma D.2 (6).

Case Seq. Assume $e_1; e_2, \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(e_2, \sigma', \texttt{CT}')$ with $\vec{u} \notin \mathsf{fnv}(e_2)$ and $\Gamma \vdash e_1; e_2 : S$. To derive this, TE-Seq was applied with the premises that $\Gamma \vdash e_1 : U$ and $\Gamma \vdash e_2 : S$. By premises of Seq we have $e_1, \sigma, \texttt{CT} \longrightarrow_l v, \sigma', \texttt{CT}'$, and so by the inductive hypothesis we have $\Gamma, \vec{u} : \vec{T} \vdash v : U'$ with $U' <: U$ and $\Gamma, \vec{u} : \vec{T}; \Delta \vdash \sigma' : \texttt{ok}$, $\vdash \texttt{CT}' : \texttt{ok}$. Then to complete the case we apply weakening to derive $\Gamma, \vec{u} : \vec{T} \vdash e_2 : S$.

Case Ass. Straightforward using Lemma D.2 (2).

Case FldAss. Use Lemma D.2 (5).

Case 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 D.2 (4) to complete the case.

Case 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 Dec. Straightforward by Lemma D.2 (1).

Case Cong. Use Lemma 6.5(3).

Case Resolve. By TE-ClassLoad.

Case DNothing. Assume By TE-ClassLoad.

Case Freeze. We shall 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

$$\sigma' = \mathsf{og}(\sigma, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y, \quad \text{with } \{\vec{y}\} = \mathsf{fv}(e) \setminus \{x\},\ \sigma_y = \bigcup \sigma(y_i) \tag{a}$$

$$\{\vec{u}\} = \mathsf{dom}(\sigma') \tag{b}$$

$$\texttt{CT}' = \mathsf{cg}(\texttt{CT}, \mathsf{fcl}(e) \cup \mathsf{fcl}(\sigma')) \tag{c}$$

From (a) we can apply Lemma 4.1 (1) 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 D.2 (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 D.1 (strengthening). From this knowledge, we can apply Lemma D.2 (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 $C \in \mathsf{fcl}(e) \cup \mathsf{fcl}(\sigma')$ then $C \in \mathsf{dom}(\mathsf{CSig})$. So we apply Lemma 4.1 (3) 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 D.1 (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 DEFROST. Assume

$$\texttt{defrost}(v;\ \lambda(T\ x).(\nu\, \vec{u})(m, e, \sigma', \texttt{CT}')), \sigma, \texttt{CT} \tag{b}$$

$$\longrightarrow_l (\nu\, x\vec{u})(\texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in sandbox}\ \{e[\vec{C^m}/\vec{C}]\}, \sigma \cup \sigma' \cdot [x \mapsto v], \texttt{CT} \cup \texttt{CT}')$$

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

$$\Gamma \vdash v : T' \text{ with } T' <: T \text{ and } \Gamma \vdash \lambda(T\ x).(\nu\, \vec{u})(m, e, \sigma', \texttt{CT}') : T \to U \tag{a}$$

We shall prove

$$\Gamma, x : T, \vec{u} : \vec{T} \vdash \texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in sandbox}\ \{e[\vec{C^m}/\vec{C}]\} : U \tag{b}$$

$$\Gamma, x : T, \vec{u} : \Delta \vdash \sigma \cup \sigma' \cdot [x \mapsto v] : \texttt{ok} \tag{c}$$

$$\vdash \texttt{CT} \cup \texttt{CT}' : \texttt{ok} \tag{d}$$

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, $\{\vec{F}\} = \mathsf{fcl}(\sigma') \setminus \mathsf{dom}(\texttt{CT}')$, 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 D.1 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 13), and so can deduce that $\vdash F_i : \mathtt{tp}$ 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 D.2 (7) to derive $\Gamma, \vec{u} : \vec{T} \vdash \sigma \cup \sigma' : \mathtt{ok}$. Then by application of Lemma D.2 (1) we have (c). (d) then follows from $\mathsf{Inv}(3)$ to complete this case.

(2) Assume $\Gamma; \Delta \vdash F : \mathtt{conf}$, $F \longrightarrow_l F'$ and $F' \not\models \mathsf{Err}$. Then we have $\Gamma; \Delta \vdash F' : \mathtt{conf}$. To type a configuration, we apply TC-Conf which has the premises $\Gamma; \Delta_1 \vdash P : \mathtt{thread}$, $\Gamma; \Delta_2 \vdash \sigma : \mathtt{ok}$, $\vdash \mathtt{CT} : \mathtt{ok}$, $\mathtt{FCT} \subseteq \mathtt{CT}$, $\Delta_1 \asymp \Delta_2$ where $\Delta = \Delta_1 \odot \Delta_2$. Proofs proceed from this point, and we omit the store σ and class table $\mathtt{CT}$ when they do not change.

Case LeaveSandbox. Straightforward by the inductive hypothesis.

Case MethLocal. Assume $E[o.m(v)] \,|\, P \longrightarrow_l (\nu\, c)(E[\mathtt{await}\ c] \,|\, o.m(v)\ \mathtt{with}\ c \,|\, P)$ and $\Gamma; \Delta_1 \vdash E[o.m(v)] \,|\, P : \mathtt{thread}$. To type this, rule TT-Par was applied with the premises

$$\Gamma; \Delta_{11} \vdash E[o.m(v)] : \mathtt{thread} \qquad \Gamma; \Delta_{12} \vdash P : \mathtt{thread} \qquad \text{with } \Delta_{11} \asymp \Delta_{12} \qquad \text{(a)}$$

To type (a), we applied Lemma 6.5(3) with premises

$$\Gamma \vdash o.m(v) : U \qquad \Gamma; \Delta_{11} \vdash E[\ ]^U : \mathtt{thread} \qquad \text{(b)}$$

Pick a fresh channel c, then apply TT-MethWith to (a) to obtain

$$\Gamma; \mathtt{chan0}(U) \vdash o.m(v)\ \mathtt{with}\ c : \mathtt{thread}$$

With the same fresh channel, apply TT-Await to (b) to obtain

$$\Gamma; \Delta_{11}, c : \mathtt{chanI}(U) \vdash E[\mathtt{await}\ c] : \mathtt{thread}$$

By Definition 5.1, we have $\Delta_{11}, c : \mathtt{chanI}(U) \asymp c : \mathtt{chan0}(U)$ with

$\Delta_{11}, c : \mathtt{chanI}(U) \odot c : \mathtt{chan0}(U) = \Delta_{11}, c : \mathtt{chan}$. Since c was chosen fresh, $c \notin \mathsf{dom}(\Delta_{12})$ therefore $\Delta_{11}, c : \mathtt{chan} \asymp \Delta_{12}$ and we can apply TT-Par twice to get

$$\Gamma; \Delta_{11}, c : \mathtt{chan} \odot \Delta_{12} \vdash E[\mathtt{await}\ c] \,|\, o.m(v)\ \mathtt{with}\ c \,|\, P : \mathtt{thread}$$

Then by permutation of the environment (Lemma D.1) we have $\Delta_{11}, c : \mathtt{chan} \odot \Delta_{12} = \Delta_1, c : \mathtt{chan}$. Applying TT-Res yields:

$$\Gamma; \Delta_1 \vdash (\nu\, c)(E[\mathtt{await}\ c] \,|\, o.m(v)\ \mathtt{with}\ c \,|\, P) : \mathtt{thread}$$

as required.

Case MethRemote. Assume

$$E[o.m(v)] \,|\, P \longrightarrow_l (\nu\, c)(E[\mathtt{await}\ c] \,|\, \mathtt{go}\ o.m(\mathtt{serialize}(v))\ \mathtt{with}\ c \,|\, P)$$

with $o \notin \mathsf{dom}(\sigma)$ and $\Gamma; \Delta_1 \vdash E[o.m(v)] \,|\, P : \mathtt{thread}$. 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 $\mathsf{remote}(C)$. Therefore we can apply TT-GoSer to derive:

$$\Gamma; c : \mathtt{chan0}(U) \vdash \mathtt{go}\ o.m(\mathtt{serialize}(v))\ \mathtt{with}\ c : \mathtt{thread}$$

Proof then proceeds from this point as in the case for MethLocal to derive $\Gamma; \Delta_1 \vdash (\nu\, c)(E[\mathtt{await}\ c] \,|\, \mathtt{go}\ o.m(\mathtt{serialize}(v))\ \mathtt{with}\ c \,|\, P) : \mathtt{thread}$, as required.

Case MethInvoke. Assume

$$o.m(v)\ \mathtt{with}\ c, \sigma \longrightarrow_l (\nu\, x)(e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}], \sigma \cdot [x \mapsto v])$$

and $\Gamma; \Delta_1 \odot \Delta_2 \vdash o.m(v)\ \mathtt{with}\ c, \sigma : \mathtt{conf}$. To infer this, we applied TC-Conf with premises

$$\Gamma; \Delta_1 \vdash o.m(v)\ \mathtt{with}\ c : \mathtt{thread} \qquad \Gamma; \Delta_2 \vdash \sigma : \mathtt{ok} \qquad \Delta_1 \asymp \Delta_2$$

By premises of TT-MethWith, we have $\Gamma \vdash o.m(v) : U$ and $\Delta_1 = c : \mathtt{chan0}(U)$. By application of MethInvoke in the reduction step, and the fact that to infer the above, we had to apply TE-Meth we have

$$\begin{array}{l} \sigma(o) = (C, \ldots) \qquad \mathsf{mbody}(m, C, \mathtt{CT}) = (x, e) \qquad \mathsf{mtype}(m, C) = D \to U \\ \Gamma \vdash o : C \qquad \Gamma \vdash v : D' \qquad D' <: D \end{array} \tag{a}$$

By assumption that $\vdash \mathtt{CT} : \mathtt{ok}$, we can apply Lemma D.3 to (a) to obtain that $x : D, \mathtt{this} : C \vdash e : \mathtt{ret}(U')$ with $U' <: U$ for a freshly chosen x. By application of Lemma 6.5 (substitution) followed by Lemma D.1 (strengthening) we have $\Gamma, x : D \vdash e[o/\mathtt{this}] : \mathtt{ret}(U')$. Then by applying TT-Return we have

$$\Gamma, x : D; c : \mathtt{chan0}(U) \vdash e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}] : \mathtt{thread}$$

By application of Lemma D.2 (1) we then have that $\Gamma, x : D \vdash \sigma \cdot [x \mapsto v] : \mathtt{ok}$. To complete the case we then apply TC-Conf followed by TC-ResId giving

$$\Gamma; \Delta_1 \odot \Delta_2 \vdash (\nu\, x)(e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}], \sigma \cdot [x \mapsto v]) : \mathtt{conf}$$

as required.

Case Await. Assume $E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v \longrightarrow_l E[v]$ and $\Gamma; \Delta_1 \vdash E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v : \mathtt{thread}$. To type this, we applied TT-Par with premises:

$$\Gamma; \Delta_{11} \vdash E[\mathtt{await}\ c] : \mathtt{thread} \qquad \Gamma; \Delta_{12} \vdash \mathtt{return}(c)\ v : \mathtt{thread} \qquad \Delta_{11} \asymp \Delta_{12} \tag{a}$$

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

$$\begin{array}{ll} \Gamma; \Delta'_{11} \vdash E[\ ]^U : \mathtt{thread} & \text{with } \Delta_{11} = \Delta'_{11}, c : \mathtt{chanI}(U) \\ \Gamma \vdash \mathtt{return}\ v : \mathtt{ret}(U') & \text{with } U' <: U \end{array} \tag{b}$$

To derive (b), we applied TE-Return with the premise that $\Gamma \vdash v : U'$. By Lemma 6.5(3) we obtain $\Gamma; \Delta'_{11} \vdash E[v] : \texttt{thread}$. To complete the case we apply Lemma D.1 (weakening) to the environment Δ'_{11} to give $\Gamma; \Delta_1 \vdash E[v] : \texttt{thread}$.

Case Fork. Similar to the above case, using Lemma 6.5(3).

Case ThreadDeath. Trivial.

Case 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 6.5(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 holds, 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 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 6.5(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

$$\mathsf{insync}(o, E) \qquad \mathsf{getLock}(\sigma, o) = n \qquad \mathsf{setLock}(\sigma, o, 0) = \sigma'' \qquad \mathsf{block}(\sigma'', o, c) = \sigma'$$

By Lemma 6.4, 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 5.1, $\Delta_{11}, c : \texttt{chanI}(\texttt{void}) \asymp \Delta_{12}$. 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 D.2 (9), $\Gamma; \Delta_2, c : \texttt{chan0}(\texttt{void}) \vdash \sigma' : \texttt{ok}$. By Definition 5.1 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 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

$$\Gamma; \Delta_{11} \odot \Delta_{12} \vdash E[o.\texttt{notify}] \mid E_1[\texttt{waiting}(c)\ n] : \texttt{thread} \qquad \Delta_1 = \Delta_{11} \odot \Delta_{12} \quad \text{(a)}$$
$$\Gamma; \Delta_2 \vdash \sigma : \texttt{ok} \quad \text{(b)}$$

To derive (a) we applied TT-Par with premises

$$\text{(i) } \Gamma; \Delta_{11} \vdash E[o.\texttt{notify}] : \texttt{thread} \qquad \text{(ii) } \Gamma; \Delta_{12} \vdash E_1[\texttt{waiting}(c)\ n] : \texttt{thread} \quad \text{(c)}$$

To derive (c-i) there were two possible rules 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 6.5(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:

$$\Gamma; \Delta'_{12} \vdash E_1[\,]^{\texttt{void}} : \texttt{thread}, \quad c \notin \mathsf{dom}(\Delta'_{12}) \quad n > 0 \quad \Delta_{12} = \Delta'_{12}, c : \texttt{chanI}(\texttt{void}) \quad \text{(d)}$$

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 that $c \in \mathsf{blocked}(\sigma, o)$, 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 D.1 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 D.2 (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 NotifyAll. Similar to the case for Notify.

Case NotifyNone. Straightforward.

Case READY. Assume this should be in expressions $\texttt{ready}\ o\ n, \sigma \longrightarrow_l \epsilon, \sigma'$ with $\Gamma \vdash \texttt{ready}\ o\ n : \texttt{void}$ and $\Gamma; \Delta_2 \vdash \sigma : \texttt{ok}$. Trivially, $\Gamma \vdash \epsilon : \texttt{ok}$, therefore it remains 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 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 the premise of LEAVECRITICAL we have that $\mathsf{getLock}(\sigma, o) = n$ and $\mathsf{setLock}(\sigma, o, n-1) = \sigma'$, and by the assumption of σ, we have that $n \geq 0$. By Lemma 6.4 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 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

$$\Gamma; \Delta_{11} \vdash P_1 : \texttt{thread},\ \Gamma; \Delta_{12} \vdash P_2 : \texttt{thread},\ \Delta_{11} \asymp \Delta_{12}\ \Delta_{11} \odot \Delta_{12} = \Delta_1 \qquad \text{(a)}$$

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 D.4, $\Delta_{11} \asymp \Delta_2$. 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 virtue of 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 RC-STR. Straightforward by Lemma 6.2.

Case 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.

(2)

Case 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 6.1 (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 DOWNLOAD. Assume $l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] \longrightarrow l_1[E[\texttt{resolve}\ \vec{D}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1 \cup \texttt{CT}'] \mid l_2[P_2, \sigma_2, \texttt{CT}_2]$ 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:

$$\Gamma; \Delta_1 \vdash l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1] : \texttt{net} \quad \Delta_1 \asymp \Delta_2 \qquad \text{(a)}$$
$$\Gamma; \Delta_2 \vdash l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$$

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 $\vdash \vec{C} : \texttt{tp}$. Then by examination of the premises of reduction rule DOWNLOAD, we have that $\vec{D} \subseteq \vec{C}$ and so trivially $\vdash \vec{D} : \texttt{tp}$. Thus we can apply TE-CLASSLOAD again to derive that $\Gamma \vdash \texttt{resolve}\ \vec{D}\ \texttt{from}\ l_2\ \texttt{in}\ e : U$ as required. Now it remains 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:

$$\Gamma; \Delta \vdash l_1[E[\texttt{resolve}\ \vec{D}\ \texttt{from}\ l_2\ \texttt{in}\ e] \mid P, \sigma_1, \texttt{CT}_1 \cup \texttt{CT}'] \mid l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$$

Case 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 it remains to show that the channel environments can be safely composed, however this is straightforward by Lemma D.4 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 RETURN. Similar to case LEAVE.

Case SERRETURN. Assume $l[\texttt{return}(c)\ v \mid P, \sigma, \texttt{CT}] \longrightarrow l[\texttt{go}\ \texttt{serialize}(v)\ \texttt{to}\ c \mid P, \sigma, \texttt{CT}]$ 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

$$\Gamma; \Delta_{11} \vdash \texttt{return}(c)\ v : \texttt{thread} \qquad \Delta = \Delta_1 \odot \Delta_2 \qquad \Delta_1 = \Delta_{11} \odot \Delta_{12} \qquad \text{(a)}$$
$$\Gamma; \Delta_{12} \vdash P : \texttt{thread} \qquad \Delta_{11} \asymp \Delta_{12}$$

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 RN-Par. Straightforward by the inductive hypothesis.

Case 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 RN-Str. Straightforward using Lemma 6.2.

G. PROOFS OF PROPOSITION 7.1 (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.

$$[\![\texttt{CT} \cdot [C \mapsto L]]\!] \stackrel{\text{def}}{=} [\![\texttt{CT}]\!] \cdot [C \mapsto L'] \cup \texttt{CT}' \text{ where } (L', \texttt{CT}') = [\![L]\!]$$

$$[\![\texttt{CSig} \cdot C : \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T_i} \to U_i\}]\!] \stackrel{\text{def}}{=} [\![\texttt{CSig}]\!] \cdot C : \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : C_{mi} \to U_i\} \cdot C_{mi} : \vec{T_i}\vec{\texttt{f}_i}$$

$$[\![\texttt{class}\ C\ \texttt{extends}\ D\,\{\vec{T}\vec{f};\ K\,\vec{M}\}]\!] \stackrel{\text{def}}{=} (\texttt{class}\ C\ \texttt{extends}\ D\,\{\vec{T}\vec{f};\ K\,\vec{M}'\}, \texttt{CT}) \text{ where } (\vec{M}', \texttt{CT}) = [\![\vec{M}]\!]_{\texttt{this}:C}$$

$$[\![U\, m\, (\vec{T}\vec{x})\{e\}]\!]_{\texttt{this}:C} \stackrel{\text{def}}{=} (U\, m\, (C_m\ z)\{e'\}, [C_m \mapsto \texttt{class}\ C_m\{\vec{T}\vec{f};\ K\}] \cup \texttt{CT}) \quad \text{where } (e', \texttt{CT}) = [\![e]\!]^{\vec{x},z}_{\texttt{this}:C,\vec{x}:\vec{T}}$$

$$[\![\lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT})]\!] \stackrel{\text{def}}{=} \lambda(T\ x).(\nu\,\vec{u})(l, e', \sigma', \texttt{CT} \cup \texttt{CT}') \quad \text{where } (e', \texttt{CT}_0) = [\![e]\!]_{\vec{x}:\vec{T},\vec{u}:\vec{T'}}, (\sigma', \texttt{CT}_1) = [\![\sigma]\!]_{\vec{x}:\vec{T},\vec{u}:\vec{T'}}, \texttt{CT}' = \texttt{CT}_0 \cup \texttt{CT}_1$$

The encoding of other values are identical. Next we define the main rules for the expressions. Others are just homomorphic like $e_0; e_1$ below. We let $(e'_i, \texttt{CT}_i) = [\![e_i]\!]^{\vec{x},z}_{\Gamma}$ for $i \geq 0$ in the following.

$$[\![y]\!]^{x_1,\ldots,x_n,z}_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} (z.\texttt{f}_\texttt{i}, \emptyset) & \text{if } y = x_i \\ (y, \emptyset) & \text{otherwise} \end{cases} \qquad [\![\texttt{this}]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (\texttt{this}, \emptyset)$$

$$[\![y := e_0]\!]^{x_1,\ldots,x_n,z}_{\Gamma} \stackrel{\text{def}}{=} \begin{cases} (z.\texttt{f}_\texttt{i} := e'_0, \texttt{CT}_0) & \text{if } y = x_i \\ (y := e'_0, \texttt{CT}_0) & \text{otherwise} \end{cases} \qquad [\![e_0; e_1]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (e'_0; e'_1, \bigcup \texttt{CT}_i)$$

$$[\![e_0.m(e_1, \ldots, e_n)]\!]^{\vec{x},z}_{\Gamma} \stackrel{\text{def}}{=} (e'_0.m(\texttt{new}\ C_m(e'_1, \ldots, e'_n)), \bigcup \texttt{CT}_i \cup [C_m \mapsto \texttt{class}\ C_m\{\vec{T}\vec{f};\ K\}]) \quad \text{where } \Gamma \vdash e_0 : C$$

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 $[\![\sigma]\!]_\Gamma=(\sigma',\mathtt{CT})$. Let $\mathcal{R}=\{((\nu\,\vec{u})(N\mid M),(\nu\,\vec{u})([\![N]\!]\mid M))\}$. Then we prove $\mathcal{R}$ respects the action predicate and reduction-closed up to $\mapsto$, i.e.

- if $(\nu\,\vec{u})(N\mid M)\longrightarrow N_1$, then there exists N_2 s.t. $(\nu\,\vec{u})([\![N]\!]_\Gamma\mid M)\mapsto^*\longrightarrow\mapsto^* N_2$ with $N_1\equiv(\nu\,\vec{u}')(N'\mid M')$ and $N_2\equiv(\nu\,\vec{u}')([\![N']\!]_\Gamma\mid M')$ for some $\vec{u}'$, N' and M'.
- (a) if $(\nu\,\vec{u})([\![N]\!]_\Gamma\mid M)\longrightarrow N_2$, then there exists N_1 s.t. $(\nu\,\vec{u})(N\mid M)\longrightarrow N_1$ with $N_1\equiv(\nu\,\vec{u}')(N'\mid M')$ and $N_2\mapsto^*(\nu\,\vec{u}')([\![N']\!]_\Gamma\mid M')$ for some $\vec{u}'$, N' and M'; or
  (b) if $(\nu\,\vec{u})([\![N]\!]_\Gamma\mid M)\mapsto N_2$, then there exists N_1 s.t. $(\nu\,\vec{u})(N\mid M)\longrightarrow N_1$ with $N_1\equiv(\nu\,\vec{u}')(N'\mid M')$ and $N_2\longrightarrow\mapsto^*(\nu\,\vec{u}')([\![N']\!]_\Gamma\mid M')$ for some $\vec{u}'$, N' and M'

The key cases are variable-read $[\![z]\!]^{x_1,\ldots,x_n,z}_\Gamma$ and variable-write $[\![z:=v]\!]^{x_1,\ldots,x_n,z}_\Gamma$, for which we can directly use (cr) in the transformation rule in § 7.2. Since $\mapsto\subseteq\cong$, we have $N\cong[\![N]\!]_\Gamma$, as desired. □

H. PROOFS OF THEOREM 7.2 (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 value at line i in (Opt3) ($3\le i\le 5$). 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 7.1 (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:

$$P \stackrel{\text{def}}{=} \texttt{return}(c)\ r.\texttt{run}(\texttt{freeze}\{r.\texttt{g}(\texttt{deserialize}(v_2), r.\texttt{f}(\texttt{deserialize}(v_1)))\})$$

where $v_i = \lambda(\texttt{unit}\ x).(\nu\,\vec{u})(l, a, \sigma_i)$ is a serialised value at b_i in (Opt3) above. Note that if the method invocation `r.f` does not terminate in `m3`, then the both programs `m3` and `mOpt3` diverge, thus they are trivially equivalent. Hence we assume the case the method invocation `r.f` terminates.

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.

$$\begin{array}{lll}
 & l[\texttt{return}(c)\ r.\texttt{g}(a, r.\texttt{f}(a))] \,|\, m[\mathbf{0}] & \\
\mapsto & l[\texttt{return}(c)\ r.\texttt{g}(a, \texttt{await}\ c_1) \,|\, \texttt{go}\ r.\texttt{f}(\texttt{serialize}(a))\ \texttt{with}\ c_1] \,|\, m[\mathbf{0}] & \text{(ni)} \\
\longrightarrow & l[\texttt{return}(c)\ r.\texttt{g}(a, \texttt{await}\ c_1) \,|\, \texttt{go}\ r.\texttt{f}(v_1)\ \texttt{with}\ c_1] \,|\, m[\mathbf{0}] & (s1) \\
\mapsto^+ & l[\texttt{return}(c)\ r.\texttt{g}(a, \texttt{await}\ c_1)] \,|\, m[r.\texttt{f}(\texttt{deserialize}(v_1))\ \texttt{with}\ c_1] & \text{(ni)} \\
\mapsto^+ & l[...] \,|\, m[(\nu\,\vec{u})(r.\texttt{f}(a)\ \texttt{with}\ c_1, \sigma_1)] & \text{(ni)} \\
\longrightarrow\!\!\!\!\rightarrow & l[...] \,|\, m[\texttt{return}(c_1)\ n] & (\ddagger) \\
\mapsto^+ & l[\texttt{return}(c)\ r.\texttt{g}(a, n)] \,|\, m[\mathbf{0}] & \text{(rm,l2,ni)} \\
\mapsto\longrightarrow\mapsto^+ & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, m[r.\texttt{g}(\texttt{deserialize}(v_2), n)\ \texttt{with}\ c'] & \text{(ni,s2,ni)} \\
\mapsto^+ & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, (\nu\,\vec{u})(m[\texttt{return}(c')\ r.\texttt{g}(a, n), \sigma_2]) & \text{(ni)}
\end{array}$$

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.

$$\begin{array}{lll}
 & l[P] \,|\, m[\mathbf{0}] & \\
\mapsto & l[\texttt{return}(c)\ r.\texttt{run}(\lambda().(r.\texttt{g}(\texttt{deserialize}(v_2), r.\texttt{f}(\texttt{deserialize}(v_1)))))] \,|\, m[\mathbf{0}] & \\
\mapsto^+ & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, (\nu\,\vec{u})(m[\texttt{return}(c')\ r.\texttt{g}(\texttt{deserialize}(v_2), r.\texttt{f}(a)), \sigma_1]) & \\
\longrightarrow\!\!\!\!\rightarrow & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, m[\texttt{return}(c')\ r.\texttt{g}(\texttt{deserialize}(v_2), n)] & (\dagger) \\
\mapsto^+ & l[\texttt{return}(c)\ \texttt{await}\ c'] \,|\, (\nu\,\vec{u})(m[\texttt{return}(c')\ r.\texttt{g}(a, n), \sigma_2]) &
\end{array}$$

In the above, we know that we can get n at ($\dagger$) if and only if n is obtained at ($\ddagger$). Also the serialisation (s1,s2) in the original code can be derived with the assumption such that v_1 and v_2 are serialised values in Line 2 and Line 3 in the optimised code. Since $\mapsto \subseteq \cong$, (RMI3) is equivalent with (Opt3).