Formalising Java RMI with Explicit Code Mobility

Alexander Ahern
aja@doc.ic.ac.uk

Nobuko Yoshida
yoshida@doc.ic.ac.uk

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

Abstract

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

Keywords

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 [40] (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 objects and services. However when rigorously analysing the dynamics of distributed programs, or when providing programmers with source-level control over code distribution [11], 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* [8] or *aggregation* [47, 46], 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 [32, 38, 19]. One highlight is the use of a *linear* type discipline [27, 18, 49] 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

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.

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 [11, 8, 47, 46].

As an application of our formalism, we show that the RMI *aggregation* optimisations proposed in [47, 46] 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.

- 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.
- 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.
- 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. Due to space limitations, the detailed definitions and proofs, as well as many examples, are left to the full technical reports [4, 3] available from: `http://www.doc.ic.ac.uk/~aja/dcbl.html`.

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 [14] but which is generally applicable to various forms of remote communication, is *aggregation* [8, 47, 46]. We explain this idea using a simple program.

```
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  }
```

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.

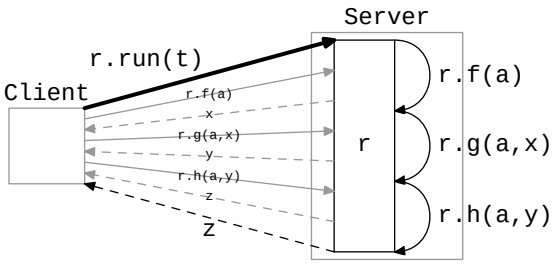
This optimisation methodology is implemented in the Veneer virtual Java Virtual Machine (vJVM) [47, 46], where sequences of adjacent calls to the same remote object are grouped together into an execution *plan* in bytecode format. This is then uploaded to and executed by the server, with the result of the computation being returned to the client. This simple idea—remote evaluation of code [39]—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 [47] 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 [14].

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 [47, 46], but requires explicit modelling if one wishes to discuss its properties or show that it preserves the original program semantics.

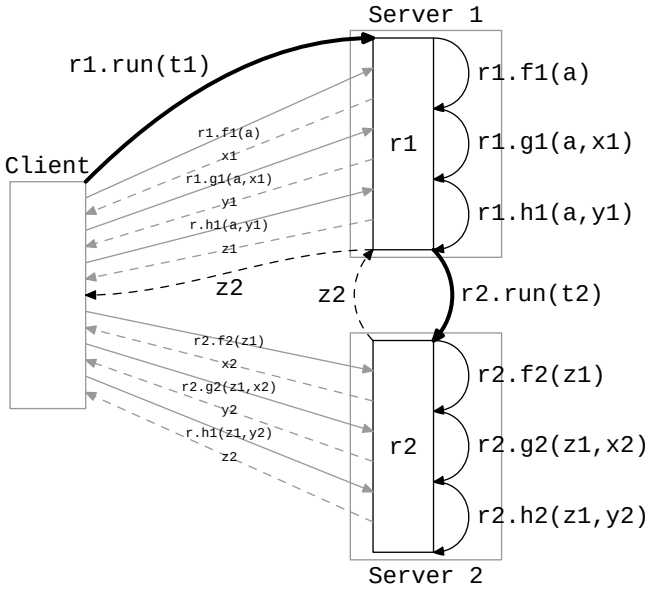
For this purpose we introduce two primitives, **`freeze`** and **`defrost`**. These mimic primitives found in well-known functional languages, for example the quotation and evaluation of code in Scheme, or the higher-order functions found in ML and Haskell. We illustrate these primitives using the optimised version of the code above.

```
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 }
```

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 expres-



(a) Aggregation



(b) Server forwarding

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.

Figure 2.1: Example optimisations

---

sion, 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.

In Figure 1(a) 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* [47, 46]. 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 following program.

```
int m2(RemoteObject r1,
       RemoteObject r2, int a) {
  int x1 = r1.f1(a);
  int y1 = r1.g1(a, x1);
  int z1 = r1.h1(a, y1);
  int x2 = r2.f2(z1);
  int y2 = r2.g2(z1,x2);
  int z2 = r2.h2(z1,y2);
  return z2;
}
```

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. In Figure 1(b) we give a diagram of the situation.

Optimised program 2. Server forwarding again uses code passing as an execution mechanism. We use closure passing for representing this optimised code, in which thunked code is *nested*, i.e. we are using higher-order code mobility.

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

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 following code:

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

Here we have two cases:

`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*.

`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*.

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 much 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 following code. 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`).

```
1  // Client
2  int mOpt3(RemoteObject r, MyObj a) {
3    ser<MyObj> b1 = serialize(a);
4    ser<MyObj> b2 = serialize(a);
5    ser<MyObj> b3 = serialize(a);
6    thunk<int> t = freeze {
7      int x = r.f(deserialize(b1));
8      int y = r.g(deserialize(b2), x);
9      int z = r.h(deserialize(b3), y);
10     z
11   };
12   return r.run(t);
13 }
```

In the above code, the declaration **`ser`**`<MyObj> b1` says that `b1` is a serialised representation of an object of class `MyObj`.

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 [29, 36]. 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 [32, 38, 19]. 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.

```
1 // Client
2 int mOpt4(RemoteObject r, MyObj a) {
3   ... // as in mOpt3
4   thunk<int> t = freeze[eager] {
5   ... // as in mOpt3
6 }
```

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 § 4.5 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.

```
1 // Client
2 thunk<int> t;
3 if (pingTime() > 1000) {// milliseconds
4   t = freeze[eager] {...};
5 } else {
6   t = freeze[lazy] {...};
7 }
```

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 [22] and MJ [7], 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 3.1. The first form is called *user syntax*, and corresponds to terms that can be written by a programmer as source code. The second form is called *runtime syntax*. It extends the user syntax with constructs that only appear during program execution, and these are distinguished in the figure by placing them in shaded regions. We briefly discuss each syntactic category below.

Types. T and U range over types for expressions and statements, which are explained in § 5. C, D, F range over class names. $\vec{f}$ denotes a vector of fields, and $\vec{T}\,\vec{f}$ is short-hand for a sequence of typed field declarations: $T_1 f_1; \ldots; T_n f_n$. We assume sequences contain no duplicate names, and apply similar abbreviations to other sequences with ε representing the empty sequence. $T \to U$ denotes an *arrow type*, which is assigned to frozen expressions that expect a parameter of type T and return a value of type U. We abbreviate the type of thunked frozen expressions as $\texttt{thunk}\langle U\rangle \stackrel{\text{def}}{=} \texttt{unit} \to U$. We associate the type $\texttt{ser}\langle U\rangle$ with frozen *values*. 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 § 7 for the formal proofs.

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

$$\begin{aligned} \texttt{serialize}(e) &\stackrel{\text{def}}{=} \texttt{freeze}[\texttt{lazy}]\{e\} \\ \texttt{deserialize}(e) &\stackrel{\text{def}}{=} \texttt{defrost}(e) \end{aligned}$$

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 the types of fields and 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. $\texttt{new}\, C^l(\vec{v})$, $\texttt{download}\,\vec{C}\,\texttt{from}\,l\,\texttt{in}\,e$

Syntax occurring only at runtime appears in shaded regions.

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

Figure 3.1: The syntax of the language DJ.

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^{-} 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 $\texttt{Error}$ 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 $\texttt{CT}$ 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. $\texttt{CT}$ 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 $\texttt{CT}$ 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 $\texttt{void}$ 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 [32]. It also includes $\texttt{Error}$ which denotes the result of class downloading and communication failure. The expression $\texttt{forked}\,e$ 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 $\texttt{CT}$. 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 $o.\texttt{wait}$ 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 $\texttt{CT}$, are a mapping from unlabelled class names to class definitions (metavariable L in Figure 3.1). Throughout the paper we write $\texttt{FCT}$ for the *foundation class table* that contains the common classes that every location in the network should possess, roughly corresponding to the $\texttt{java.*}$ classes.

New

$$\frac{\mathsf{fields}(C) = \vec{T}\,\vec{f} \qquad C \in \mathsf{dom}(\mathtt{CT})}{\mathtt{new}\ C^{\text{-}}(\vec{v}), \sigma, \mathtt{CT} \longrightarrow_l (\nu\, o)(o, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v}, 0, \emptyset)], \mathtt{CT})}$$

NewR

$$\frac{C \notin \mathsf{dom}(\mathtt{CT})}{\mathtt{new}\ C^{m}(\vec{v}), \sigma, \mathtt{CT} \longrightarrow_l \mathtt{download}\ C\ \mathtt{from}\ m\ \mathtt{in}\ \mathtt{new}\ C(\vec{v}), \sigma, \mathtt{CT}}$$

Figure 4.1: Rules for object creation

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

Figure 4.2: Evaluation contexts

Resolve

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

Download

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

DNothing

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

Err-ClassNotFound

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

Figure 4.3: Rules for class resolution and downloading

4. Operational Semantics

This section presents the formal operational semantics of DJ, extending the standard small step call-by-value reduction of [35, 7]. 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 [32], written $\equiv$. We define *multi-step* reductions as: $\twoheadrightarrow \stackrel{\text{def}}{=} (\longrightarrow \cup \equiv)^*$ and $\twoheadrightarrow_l \stackrel{\text{def}}{=} (\longrightarrow_l \cup \equiv)^*$. The reduction rules not introduced in this section are left to the the Appendix.

4.1 Local expressions

The rules for the sequential part of the language are standard [22, 7]. We list only the rules for object creation in Figure 4.1. 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 $\mathsf{fields}(C)$ examines the class signature and returns the field declarations for C. Tagged class creation takes place in **NewR**. This 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.

To reduce the number of computation rules, we make use of the evaluation contexts in Figure 4.2. 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 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 4.3 and approximately model the lazy downloading mechanism found in JDK 1.3 without verification [12]. 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 $\mathtt{download}\ \vec{C}\ \mathtt{from}\ l\ \mathtt{in}\ e$ 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. $\mathsf{fcl}(\mathtt{CT}_2(\vec{D}))$ denotes the union of free class names in class D_i. A class name C is *free* if it is the subject of an instantiation operation (written $\mathtt{new}\ C(\ldots)$), or if it appears as the class of an object in stores ($[o \mapsto (C, \ldots)] \in \sigma$). The function fcl is defined over expressions, threads, stores and class table entries.

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 is modular: we can model different class loading mechanisms by adjusting the reduction rules for downloading and resolution and the class graph algorithm introduced in Definition 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 § 4.5 (for the encoding, see § 3.1). Serialisation occurs in two instances. In the first, the expressions $\texttt{serialize}(e)$ and $\texttt{deserialize}(e)$ allow explicit flattening and re-inflation of objects by the programmer, whereas the second instance occurs when values must be transported across the network.

$\texttt{serialize}(e)$ and $\texttt{deserialize}(e)$ 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

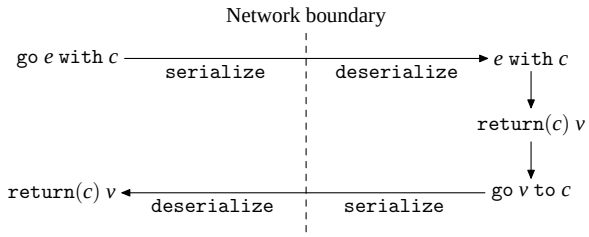
Unlike centralised formalisms, DJ describes *remote method invocation*. To accommodate RMI, the rules for method call take a novel form employing concepts from the π-calculus, representing the context of a call by a local linear channel. While this technique is well-known in the π-calculus [32], DJ may 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 serialisation, which is reflected as several extra reduction steps.

We summarise the general picture of a remote method invocation in Figure 4(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 4(b).

We start our illustration from local method calls. For a method call $o.\texttt{m}(v)$, if $o \in \mathsf{dom}(\sigma)$ then the rule **MethLocal** is applied. A new channel c is created to carry the return value of the method; the return point of the method call is replaced with the term $\texttt{await}\ c$ corresponding to a receiver waiting for the return value supplied on channel c. The method call itself is spawned in a new thread as $o.\texttt{m}(v)\ \texttt{with}\ c$ 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 $\mathsf{mbody}(\texttt{m}, C, \texttt{CT})$ looks 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 $[\texttt{return}(c)/\texttt{return}]$ 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 any local object identifiers in the vector of parameters $\vec{v}$. We note that frozen values are also transferred to the remote location without modification (like base values).



(a) Evaluation steps for a remote call

MethLocal

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

MethRemote

$$\frac{c \text{ fresh}, o \notin \mathsf{dom}(\sigma)}{\begin{array}{l} E[o.\texttt{m}(v)] \,|\, P, \sigma, \texttt{CT} \\ \longrightarrow_l (\nu\, c)(E[\texttt{await}\ c] \,|\, \texttt{go}\ o.\texttt{m}(\texttt{serialize}(v))\ \texttt{with}\ c \,|\, P, \sigma, \texttt{CT}) \end{array}}$$

MethInvoke

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

Await

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

SerReturn

$$\frac{c \notin \mathsf{fn}(P)}{l[\texttt{return}(c)\ v \,|\, P, \sigma, \texttt{CT}] \longrightarrow l[\texttt{go}\ \texttt{serialize}(v)\ \texttt{to}\ c \,|\, P, \sigma, \texttt{CT}]}$$

Leave

$$\frac{o \in \mathsf{dom}(\sigma_2)}{\begin{array}{l} l_1[\texttt{go}\ o.\texttt{m}(v)\ \texttt{with}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \\ \longrightarrow l_1[P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[o.\texttt{m}(\texttt{deserialize}(v))\ \texttt{with}\ c \,|\, P_2, \sigma_2, \texttt{CT}_2] \end{array}}$$

Return

$$\frac{c \in \mathsf{fn}(P_2)}{\begin{array}{l} l_1[\texttt{go}\ v\ \texttt{to}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \\ \longrightarrow l_1[P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[\texttt{return}(c)\ \texttt{deserialize}(v) \,|\, P_2, \sigma_2, \texttt{CT}_2] \end{array}}$$

Err-LostCall

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

Err-LostReturn

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

(b) Reduction rules

Figure 4.4: Remote method invocation

After serialisation, we are left with a thread of the form $\texttt{go}\ o.\texttt{m}(w)\ \texttt{with}\ c$ 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 parameters are 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. Like-

Freeze

$$\frac{\{\vec{y}\} = \mathsf{fv}(e) \setminus \{x\} \qquad \sigma_y = \bigcup \sigma(y_i) \qquad \sigma' = \mathsf{og}(\sigma, \mathsf{fn}(e) \cup \mathsf{fn}(\sigma_y)) \cup \sigma_y \qquad \{\vec{u}\} = \mathsf{dom}(\sigma') \qquad \mathtt{CT}' = \begin{cases} \mathsf{cg}(\mathtt{CT}, \mathsf{fcl}(e) \cup \mathsf{fcl}(\sigma')) & t = \mathtt{eager} \\ \mathsf{cg}(\mathtt{CT}, \vec{C}) & t = \vec{C} \\ \emptyset & t = \mathtt{lazy} \end{cases}}{\mathtt{freeze}[t](T\ x)\{e\}, \sigma, \mathtt{CT} \longrightarrow_l \lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma', \mathtt{CT}'), \sigma, \mathtt{CT}}$$

Defrost

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

LeaveSandbox

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

Figure 4.5: Rules for creating and executing frozen expressions

wise, in **Err-LostReturn**, the return value from a remote method call is lost. Both cases reduce to `Error`.

4.5 Direct code mobility

Frozen expressions offer a direct way to manipulate code and data. They permit the storing of unevaluated terms that can, for example, be shipped to remote locations for evaluation or merely saved for future use. As we have seen in § 3.1, our formulation of the primitives subsumes the serialisation operations found in Java that were explained in § 4.3.

As introduced in Figure 3.1, there are two operations associated with frozen values—for their creation and use—called *freezing* and *defrosting* respectively. Their rules are given in Figure 4.5.

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 Definition 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 and σ' depend 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 graph algorithm in Definition 2 to determine the classes that the expression (or the user-specified classes) depends upon.

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

The object graph is defined as the set of all mappings from 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.

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 [25] 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.

DEFINITION 2. The function $\mathsf{cg}(\mathtt{CT}, T)$ computes the class graph of type T in class table `CT` as follows:

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

A small example of the freezing process is as follows:

```
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};
```

After executing the above program at location l, we should obtain a frozen expression of the form:

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

where $\sigma_1 = [o_1 \mapsto (A, f : 5, g : o_2, 0, \emptyset)] \cdot [o_2 \mapsto (B, \varepsilon, 0, \emptyset)] \cdot [y \mapsto 6]$ and $\mathtt{CT}_1 = [B \mapsto \ldots]$

To defrost a frozen value $\lambda(T\ x).(\nu\, \vec{u})(l, e, \sigma_1, \mathtt{CT}_1)$ we use **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 `download` $\vec{F}$ `from` l `in sandbox` $\{e\}$. This accurately mimics the mechanism employed by the `RMIClassLoader` class used in 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 m, where it is important to notice that a variable y is already in scope: here the ν-operator will be used to avoid collision of bound variables. We abbreviate `download` to `dl` and `sandbox` to `sb` in the following:

$$
\begin{array}{l}
\quad \texttt{defrost}(5;\, t), [y \mapsto \texttt{true}], \texttt{CT} \\
\longrightarrow_m (\nu\, x, o_1, o_2, y_2)(\texttt{dl}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y_2 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_2) \\
\text{with } \sigma_2 = [y \mapsto \texttt{true}] \cdot [o_1 \mapsto (A, f : 5, g : o_2, 0, \emptyset)] \cdot \\
\qquad [o_2 \mapsto (B, \varepsilon, 0, \emptyset)] \cdot [y_1 \mapsto 6] \cdot [x \mapsto 5] \\
\text{and } \texttt{CT}_2 = \texttt{CT} \cdot [B \mapsto \ldots] \\
\longrightarrow_m (\nu\, x, o_1, o_2, y_1)(\texttt{resolve}\ A\ \texttt{from}\ l\ \texttt{in sb}\ \{x + y_1 + o_1.\texttt{f}\}, \sigma_2, \texttt{CT}_3) \\
\text{with } \texttt{CT}_3 = \texttt{CT}_2 \cdot [A \mapsto \ldots]
\end{array}
$$

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

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

In the final steps, we garbage-collect the store entries added by the frozen expression since they are now no longer required.

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 d(){return new D()}}
4 class D { }
```

- 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. d

4.6 Correctness of graph algorithms

In this subsection we show the correctness of the graph algorithms that are used in the proof of the results in § 6. The reader may safely skip this subsection if they wish.

Object graph algorithm. The predicate $\mathsf{reachable}(\sigma, o, o')$ holds 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 below, where $\sigma(o) = (C, \vec{f} : \vec{v}, \ldots)$:

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

With this predicate, $RCH(\sigma)$ which contains all reachable pairs of objects in a store σ, is defined below.

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

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 object graph*) holds if the computed graph preserves the reachability relation for all object identifiers in its object domain. Given $RCH(\sigma)$ and $RCH(\sigma_g)$, we define:

$$
\begin{array}{l}
\mathsf{ogcomp}(\sigma, \sigma_g) \text{ if for all } o \in \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma_g), \\
\qquad (o, o') \in RCH(\sigma) \iff (o, o') \in RCH(\sigma_g)
\end{array}
$$

This property ensures all links are correctly copied to the graph σ_g, and no new links are created. The algorithm used to compute the class graph can safely add extra objects into σ_g without violating this property iff those objects are *unreachable* from any other that should be in the graph.

Class graph algorithm. The correctness of the class graph algorithm relies upon the definition of the following predicate:

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

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:

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

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

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

LEMMA 1. (Correctness of algorithms)

1. $\sigma' = \mathsf{og}(\sigma, v)$ *implies* $\mathsf{ogcomp}(\sigma, \sigma')$.
2. $\mathsf{ctcomp}(\texttt{CT})$ *and* $\texttt{CT}' = \mathsf{cg}(\texttt{CT}, C)$ *imply* $\mathsf{ctcomp}(\texttt{CT}' \cup \texttt{FCT})$.

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 5.1.

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

$T ::= \texttt{bool} \mid \texttt{unit} \mid C \mid T \to U$ (Types)
$U ::= \texttt{void} \mid T$ (Returnable types)

Extended types not appearing in program text:

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

Figure 5.1: Syntax of types and environments

assume $<:$ causes no cycle as in [22, 7]), and is judged on the class signature. The arrow type is standard.

Two runtime types (which do not appear in programs) are newly introduced. *Return types* are ranged over by S are used to denote the type of value returned by a method invocation ($U\,\texttt{m}(C\ x)\{e\}$ 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 5.1, $\texttt{chanI}(U)$ 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.

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

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

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

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

DEFINITION 3. 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:

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

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 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. The judgement is lightweight or local in the sense that it does not require the current global or local class table $\texttt{CT}$ such as $\Gamma \vdash_{\texttt{CT}} e : U$; this approach is not suitable for DJ since during execution, new classes may be downloaded or discovered. The lightweight judgement is possible by the use of the class signatures and invariants as explained below.

5.2.1 Freeze and Defrost.

First we focus on the key typing rule for frozen expressions:

TV-Frozen
$$\frac{\Gamma, x:T, \vec{u}:\vec{T}' \vdash e : U \qquad \Gamma, \vec{u}:\vec{T}'; \emptyset \vdash \sigma : \texttt{ok} \qquad \vdash \texttt{CT} : \texttt{ok}}{\Gamma \vdash \lambda(T\ x).(\nu\,\vec{u})(l, e, \sigma, \texttt{CT}) : T \to U}$$

The rule for typing a frozen expression is given in **TV-Frozen**. In order for such a value to be well-typed we must ensure that the store σ and $\texttt{CT}$ 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 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 example, by the combination of the invariants from $\mathsf{Inv}(4)$ to $\mathsf{Inv}(14)$ in § 6.1.4, we know $\sigma(o_i) = v_i$ is closed so that we can ensure that the resulting frozen value is closed again.

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

We now show the typing rules for the freezing and defrosting operations:

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

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

5.2.2 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.

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

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

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

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.

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. Suppose we have the remote class which contains synchronised methods `set` and `get` in location 1 and two clients in locations 2 and 3.

```
// Client 1 in Location 2
// ... import reference to r via RMI registry
synchronized (r) {
  r.set(1);
  return r.get();
}
// Client 2 in Location 3
// ... import reference to r via RMI registry
synchronized (r) {
  r.set(2);
  return r.get();
}
```

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.

TE-Fld

$$\frac{\Gamma \vdash e : C \qquad e \neq \mathtt{this}, o \implies \mathsf{local}(C) \qquad \mathsf{fields}(C) = \vec{T}\,\vec{f}}{\Gamma \vdash e.\mathtt{f}_i : T_i}$$

TE-Sync

$$\frac{\Gamma \vdash e_1 : C \qquad e_1 \neq \mathtt{this}, o \implies \mathsf{local}(C) \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash \mathtt{sync}\ (e_1)\ \{e_2\} : S}$$

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 toy example:

```
// Client 1 in Location 2
// ... import reference to r via RMI registry
r.down();
r.set(1);
int v = r.get();
r.up();
return v;

// Client 2 in Location 3
// ... import reference to r via RMI registry
r.down();
r.set(2);
int v = r.get();
r.up();
return v;
```

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`.

5.3 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:

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

Key typing rules are given below. The most important rule for threads is **TT-Par**; we type a parallel compositions of threads if a composition of their respective channel environments preserves the linearity of channels. This is checked by $\Delta_1 \asymp \Delta_2$.

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

TT-Par

$$\frac{\Gamma;\Delta_i \vdash P_i : \mathtt{thread} \qquad \Delta_1 \asymp \Delta_2}{\Gamma;\Delta_1 \odot \Delta_2 \vdash P_1 \,|\, P_2 : \mathtt{thread}}$$

TN-Conf

$$\frac{\Gamma;\Delta \vdash F : \mathtt{conf}}{\Gamma;\Delta \vdash l[F] : \mathtt{net}}$$

TC-Conf

$$\frac{\Gamma;\Delta_1 \vdash P : \mathtt{thread} \qquad \Gamma;\Delta_2 \vdash \sigma : \mathtt{ok} \qquad \vdash \mathtt{CT} : \mathtt{ok} \qquad \mathtt{FCT} \subseteq \mathtt{CT} \qquad \Delta_1 \asymp \Delta_2}{\Gamma;\Delta_1 \odot \Delta_2 \vdash P, \sigma, \mathtt{CT} : \mathtt{conf}}$$

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 and initial networks

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

DEFINITION 4. 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 `Error` as subexpression, i.e. $\mathsf{Err} = \{N \mid N \equiv (\nu\,\vec{u})(l[E[\texttt{Error}] \mid P, \sigma, \texttt{CT}] \mid N')\}$. We say ψ is a *network invariant with an initial property* ψ_0 if $\psi = \{N \mid \exists N_0.(N_0 \models \psi_0,\ N_0 \longrightarrow\!\!\!\!\!\rightarrow N,\ N \not\models \mathsf{Err})\}$

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

DEFINITION 5. Given network $N \equiv (\nu\,\vec{u})(\prod_{0\le i<n} l_i[F_i])$ with $F_i = (P_i, \sigma_i, \texttt{CT}_i)$, and assuming $0 \le j < n$, $i \ne j$ where required, we define property $\mathsf{Inv}(r)$ as a set of networks which satisfy the condition r (with $1 \le r \le 16$) 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 [17].[1]

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

6.1.2 Locality.

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

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

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

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

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

$\mathsf{Inv}(9)$ Suppose

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

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

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 $\mathsf{Inv}(4)$ and $\mathsf{Inv}(5)$, we can derive $\mathsf{fv}(P_i) \cap \mathsf{fv}(P_j) = \emptyset$, which ensures that local variables are not shared between threads at different locations. In $\mathsf{Inv}(9)$ we ensure that non-remote operations like field access and thread synchronisation are not attempted on remote object references. This particular situation highlights the necessity of the invariants, since we cannot guarantee this property alone in the typing system as we discussed in § 5.2.

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.

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

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

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

6.1.4 Other invariants.

In the following, the predicate $\mathsf{insync}(o, E)$ is true if there exist E_1 and E_2 such that $E = E_1[\texttt{insync}\ o\ \{E_2[\,]\}]$. Intuitively this means that a thread has previously acquired the lock on object o, although it may have subsequently released it by calling $o.\texttt{wait}$.

We also use the following functions. Let $\sigma(o) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$. Then to read the re-entry count for the monitor on o we use $\mathsf{getLock}(\sigma, o) = n$. To obtain the queue of threads waiting on the monitor of o we use the function $\mathsf{blocked}(\sigma, o) = \{\vec{c}\}$.

Closedness

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

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

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

Locks

$\mathsf{Inv}(15)$ $P_i \equiv E[\texttt{ready}\ o\ n] \mid Q_i \implies \mathsf{insync}(o, E) \wedge n > 0$

$\mathsf{Inv}(16)$ $P_i \equiv E[\texttt{waiting}(c)\ n] \mid Q_i$
$\implies \exists! o.c \in \mathsf{blocked}(\sigma_i, o) \wedge \mathsf{insync}(o, E) \wedge n > 0$

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

The *lock* invariants ensure the correct behaviour of the locking primitives at runtime. $\mathsf{Inv}(15)$ ensures that a thread that is ready to reacquire a lock will set that lock's count to a non-zero number. $\mathsf{Inv}(16)$ ensures that a thread does not wait for a non-existent lock.

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 states these conditions formally.

DEFINITION 6. 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:
  $\mathsf{fcl}(P_i) \subseteq \mathsf{dom}(\mathtt{CT}_i)$,
  $C \in \mathsf{fcl}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_i) \implies \mathsf{comp}(C,\mathtt{CT}_i)$ and
  $\sigma_i(o) = (C,\dots) \implies \mathsf{comp}(C,\mathtt{CT}_i)$.
- We also strengthen the locality invariant $\mathsf{Inv}(9)$ by replacing E by the arbitrary context.

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

The second extra requirement states that all initial class tables are complete w.r.t. classes in the program and stores. Note that during runs of programs, the initial properties may *not* be satisfied since classes can be downloaded lazily.

6.2 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 subcases:

(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* $1 \le r \le 16$ *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* $1 \le r \le 16$. *Then* $N_m \longrightarrow N_{m+1}$ *implies* $N_{m+1} \models \mathsf{Inv}(r)$ *for each* $1 \le r \le 16$ *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 non-trivial; it requires key additional invariants for runtime expressions related to dynamic class downloading. Then assuming **Step 1** holds, the proof of **Step 2** proceeds by induction on the derivation of reduction with a case analysis on the final typing rule applied. Lemma 1 plays a key role. [3] presents all proofs as well as the use of invariants.

THEOREM 1. **(Subject reduction)**

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

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 1. **(Network invariant)** $\wedge_{1\le r\le 16}\,\mathsf{Inv}(r)$ *is a network invariant with the initial network properties* Init *defined in Definition 6.*

Finally by each property such as availability and linearity, we can derive the following advanced progress and linearity properties.

DEFINITION 7. (Progress invariants)
Given network $N \equiv (\nu\,\vec{u})(\prod_{0\le i<n} l_i[P_i,\sigma_i,\mathtt{CT}_i])$, and assuming $0 \le k < n$, we define property $\mathsf{Prog}(r)$ as a set which satisfy the following conditions.

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

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

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

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

$\mathsf{Prog}(5)$ $P_i \equiv E[o.\mathtt{f}_i = v]\,|\,Q_i \implies [o \mapsto (C,\dots)] \in \sigma_i \wedge \mathsf{fields}(C) = \vec{T}\,\vec{f}$
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.

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

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

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

For the case of synchronisation, see [3].

THEOREM 2. **(Progress, Locality and Linearity)** *$\wedge_{1\le r\le 10}$ $\mathsf{Prog}(r)$ is a network invariant with the initial network properties* Init *defined in Definition 6.*

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

7. Justification for optimisations

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* [24, 37] to justify the correctness of these rules. Let us write $N \stackrel{\circ}{\longrightarrow} N'$ for a transformation rule of the optimisation from N to N'. Once we check $N \stackrel{\circ}{\longrightarrow} N'$ is type-preserving and satisfies the following noninterference property, then N and N' are immediately observationally equal, hence the transformation is semantics-preserving.

> if $N \longrightarrow N_1$ and $N \stackrel{\circ}{\longrightarrow} N_2$, then $N_1 \equiv N_2$ or there exists N' such that $N_1 \stackrel{\circ}{\longrightarrow} N'$ and $N_2 \longrightarrow N'$.

For tractable reasoning, we introduce syntactic transformation rules which satisfy the noninterference property. These equational laws, that come from those of the linear types of mobile processes [27, 49], allow us to check the optimisations purely syntactically.

7.1 Observational congruence

We define an observational congruence over the typed language and runtime by applying the equational theory of process algebra [21]. Hereafter we assume all networks are typed and started executing from the initial condition Init.

A relation $\mathscr{R}$ over networks is *typed* when $\Gamma_1;\Delta_1 \vdash N_1 \mathscr{R} \Gamma_2;\Delta_2 \vdash N_2$ implies $\Gamma_1 = \Gamma_2$ and $\Delta_1 = \Delta_2$. We write $\Gamma;\Delta \vdash N_1 \mathscr{R} N_2$ (or $N_1 \mathscr{R} N_2$ if $\Gamma;\Delta$ is obvious from the context) when $\Gamma;\Delta \vdash N_1$ and $\Gamma;\Delta \vdash N_2$ are related by a typed relation $\mathscr{R}$. A *typed congruence* is a typed relation $\mathscr{R}$ which is an equivalence closed under all typed contexts and the structure rules, i.e. $\equiv \subseteq \mathscr{R}$.

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 [21]. For an observation, we take the output ("`go`") to channel c.

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

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

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

Now we define the observational congruence.

DEFINITION 8. A typed congruence $\mathscr{R}$ 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 exclude 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 rules

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

DEFINITION 9. Let us assume $\stackrel{\circ}{\longrightarrow}$ is a typed relation closed under name restriction, parallel composition and the structure rules. We say $\stackrel{\circ}{\longrightarrow}$ satisfies a *noninterference property*, i.e. if $N \longrightarrow N_1$ and $N \stackrel{\circ}{\longrightarrow} N_2$, then $N_1 \equiv N_2$ or there exists N' such that $N_1 \stackrel{\circ}{\longrightarrow} N'$ and $N_2 \longrightarrow N'$.

LEMMA 2. *Suppose $\stackrel{\circ}{\longrightarrow}$ satisfies a noninterference property and $\stackrel{\circ}{\longrightarrow}$ respects the observational predicate. Then $N_1 \stackrel{\circ}{\longrightarrow} N_2 \not\models \mathsf{Err}$ implies $N_1 \cong N_2$.*

Code that can move safely. The transformation rules should reduce the number of communications and class downloads 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 § 5.2.2 and § 6.1.2).

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

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

Transformation Rules. We list the key transformation rules. Assume the right hand side is typed under $\Gamma;\Delta$. We omit surrounding context where it is unnecessary.

Linearity

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

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

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

Class

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

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

Closed

(cr) $(\nu\,x)(E[x] \,|\, P,[x \mapsto v]\cdot\sigma) \mapsto (E[v],\sigma)$ when $x \notin \mathsf{fv}(P) \cup \mathsf{fv}(E)$

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

where in (fr), $\mathsf{dom}(\sigma') \cap (\mathsf{fv}(E) \cup \mathsf{fn}(P) \cup \mathsf{fv}(E) \cup \mathsf{fn}(E)) = \emptyset$, $(\mathsf{fn}(\sigma') \cup \mathsf{fv}(\sigma')) \subseteq \mathsf{dom}(\sigma')$ and σ' and $\mathtt{CT}'$ are given following **Freeze**. These rules mean that the timing of reading a value or of freezing an expression is unimportant, provided it shares no information with other parties. Note "$\nu\,\vec{u}$" in (fr) ensures u_i is not shared.

Method Invocation

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

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

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

We leave other rules to [3]. The transformation rule $N \mapsto N'$ is defined as a binary relation generated by the above rules and closed under parallel composition, name restriction and structure rules.

THEOREM 3.

1. **(noninterference)** $\mapsto$ *satisfies a noninterference property and respects the observational predicate under a network invariant.*
2. **(type preservation)** *Assume* $\Gamma; \Delta \vdash N : \mathtt{net}$ *and* $N \not\models \mathsf{Err}$. *Then* $N \mapsto N'$ *implies* $\Gamma; \Delta \vdash N' : \mathtt{net}$.
3. **(semantic preservation)** $N \mapsto N'$ *implies* $N \cong N'$.

PROOF. (1) and (2) are mechanical. (3) uses (1) and (2) together with Lemma 2. □

Note that by Definition 4, Theorem 3 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 1.

1. $\mathtt{freeze}[t](T\ x)\{e\} \cong \mathtt{freeze}[t'](T\ x)\{e\}$.
2. *There is a fully abstract embedding* $[\![N]\!]$ *of networks* N *that contain methods* $\mathtt{m}(\vec{e})$ *and frozen expressions* $\mathtt{freeze}[t](\vec{T}\ \vec{x})\{e\}$ *with multiple parameters into networks with methods and frozen expressions with only single parameters.*

PROOF. (1) Use Lemma 2 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 $o.\mathtt{m}(\vec{v})$ becomes $o.\mathtt{m}(\mathtt{new}\ C(\vec{v}))$ for some C. We then prove that $N \cong [\![N]\!]$. See [3] for the detailed proofs. □

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) to the original program 1 (RMI1). 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 $o.\mathtt{mOpt1}(r, n)\ \mathtt{with}\ c$, (Opt1) becomes:

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

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

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

The last line is identical to (RMI1) after the method invocation by $o.\mathtt{m1}(r, n)\ \mathtt{with}\ c$. Note that `defrost` and `sandbox` do not affect other parties, so that the reduction at $(\star)$ 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) is also straightforward by repeating the same routine twice.

We show (RMI3) is equivalent with (Opt3) under the assumption there is no call-back.[2] Then the body of (Opt3) is equivalent to $\mathtt{return}\ r.\mathtt{run}(\mathtt{freeze}(e[\vec{e}'/\vec{b}]; z))$ and $e'_i = \mathtt{deserialize}(v_i)$ where $v_i = \lambda(\mathtt{unit}\ x).(\nu\, \vec{u})(l, a, \sigma_i)$ is a serialised value at line i in (Opt3) ($3 \le i \le 5$). Then we apply a similar transformation with the above to derive (RMI3). Hence (RMI3) is equivalent to (Opt3).

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

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

Finally by Proposition 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 1 is not applicable. See [3] for the full proofs. To summarise, we have:

THEOREM 4. **(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 [29, 36]. Our formulation of class downloading is modular, so it is adaptable to model other linking strategies [12, 13], see § 4.2. We set the class invariant $\mathsf{Inv}(3)$ in Definition 5. 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 [28] 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

[2] The equation 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. However, since most RMI programs do not use call-backs, we do not investigate them.

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 [10] 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 [31] 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 [33] formalised a typing system for a core Obliq calculus and studied different kinds of object aliasing. Briais and Nestmann [9] then strengthened the safety result in [31] 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 [10] 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 [25] 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 [15] extend the object calculus [2] 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 [23] treats an extension of [15] 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 [23] treats transactional serialisation as in [10]), which are among the key elements for analysis of RMI and code mobility in Java.

Scope and runtime formalisms for Java. Zhao et al [51] 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 5 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 [16] and Yu et al [50].

The JavaSeal [44] 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 [34] 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 [1] 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 [41] 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 class-loading we address.

Kamin et al [26] 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 [52] 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 [8, 47, 46], 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 [47, 46] 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 [47, 46], 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 [25, 10], 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 [22, 7], exceptions [5] and parametric polymorphism [22]; 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 [30, 8]. 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 [45]. 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 [48].

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* [20, 43] 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 [40]) 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 [42]. 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 put marshaling cost in rule **Freeze** 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 [6] to compare program performance.

Acknowledgements. We deeply thank Paul Kelly for his discussions about the Veneer vJVM and the RMI optimisations in [47, 46]. 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

[1] Acute home page. `www.cl.cam.ac.uk/users/pes20/acute`.

[2] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] A. Ahern and N. Yoshida. Full version of this paper. `www.doc.ic.ac.uk/~aja/dcbl.html`.

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

[5] D. Ancona, G. Lagorio, and E. Zucca. Simplifying types in a calculus for Java exceptions. Technical report, DISI - Univ. di Genova, 2002.

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

[7] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Univ. of Cambridge Computer Laboratory, April 2003.

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

[9] S. Briais and U. Nestmann. Mobile objects "must" move safely. In *FMOODS'02*, pages 129–146. Kluwer, 2002.

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

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

[12] S. Drossopoulou and S. Eisenbach. Manifestations of Dynamic Linking. In *USE 2002*, Málaga, Spain, June 2002.

[13] S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible Models for Dynamic Linking. In P. Degano, editor, *ESOP'03*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

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

[15] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. Technical Report 457, Univ. of Cambridge Computer Laboratory, February 1999.

[16] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *POPL'01*, pages 248–260. ACM Press, 2001.

[17] T. Greanier. Discover the secrets of the Java Serialization API. `java.sun.com/developer/technicalArticles/Programming/serialization/`.

[18] K. Honda. Composing processes. In *POPL'96*, pages 344–357, 1996.

[19] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, volume 512, pages 133–147, 1991.

[20] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

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

[22] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. and Sys.*, 23(3):396–450, 2001.

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

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

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

[26] S. Kamin, L. Clausen, and A. Jarvis. Jumbo:run-time code generation for java and its applications. In *CGO'03*. IEEE, 2003.

[27] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

[28] C. Krintz, B. Calder, and U. Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *OOPSLA'99*, pages 276–291. ACM Press, 1999.

[29] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA'98*, pages 36–44. ACM Press, 1998.

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

[31] M. Merro, J. Kleist, and U. Nestmann. Mobile objects as mobile processes. *Inf. & Comp.*, 177(2):195–241, 2002.

[32] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Inf. & Comp.*, 100(1), 1992.

[33] U. Nestmann, H. Hüttel, J. Kleist, and M. Merro. Aliasing models for mobile objects. *Inf. & Comp.*, 177(2):195–241, 2002.

[34] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *POPL'93*, pages 99–112. ACM Press, 1993.

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

[36] Z. Qian, A. Goldberg, and A. Coglio. A formal specification of Java class loading. In *OOPSLA'00*, pages 325–336. ACM Press, 2000.

[37] J. C. Reynolds. Syntactic control of interference. In *POPL'78*, pages 39–46. ACM Press, 1978.

[38] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, Univ. of Edinburgh, 1992.

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

[40] Sun Microsystems Inc. Java Remote Method Invocation (RMI). `java.sun.com/products/jdk/rmi/`.

[41] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *PEPM'97*, pages 203–217. ACM Press, 1997.

[42] F. Tejani. Implementation of a distributed mobile Java. Master's thesis, Imperial College London, 2005.

[43] V. T. Vasconcelos, A. Ravara, and S. Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

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

[45] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. In *SOSP'97*, pages 116–128. ACM Press, 1997.

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

[47] K. Yeung and P. Kelly. Optimizing Java RMI programs by communication restructuring. In *Middleware'03*, volume 2672 of *LNCS*, pages 324–343. Springer-Verlag, 2003.

[48] N. Yoshida. Channel dependency types for higher-order mobile processes. In *POPL'04*, pages 147–160. ACM Press, 2004. Full version available at `www.doc.ic.ac.uk/~yoshida`.

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

[50] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *POPL'04*, pages 39–51. ACM Press, 2004.

[51] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS'04*, 2004.

[52] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *GPCE'04*, volume 3286 of *LNCS*, pages 1–19. Springer-Verlag, 2004.

Appendix

This appendix contains the reduction rules for DJ that were not introduced in the main body of the paper. Several auxiliary definitions are omitted due to space limitations, but these can be found in the long version of this paper [3].

Field lookup

$$\mathsf{fields}(Object) = \bullet \qquad \frac{\mathsf{CSig}(C) = \texttt{extends}\ D\ \ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : C_i \to U_i\} \quad \mathsf{fields}(D) = \vec{T}'\vec{f}'}{\mathsf{fields}(C) = \vec{T}'\vec{f}', \vec{T}\vec{f}}$$

Method type lookup

$$\frac{\mathsf{CSig}(C) = \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : C_i \to U_i\}}{\mathsf{mtype}(\texttt{m}_\texttt{i}, C) = C_i' \to U_i'}$$

$$\frac{\mathsf{CSig}(C) = \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : C_i \to U_i\} \quad m \notin \{\vec{m}\}}{\mathsf{mtype}(\texttt{m}, C) = \mathsf{mtype}(\texttt{m}, D)}$$

Method body lookup

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

[Expression]

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

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

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

Fld
$$\frac{\sigma(o) = (C, \vec{f} : \vec{v}, n, \vec{c})}{o.\texttt{f}_\texttt{i}, \sigma, \texttt{CT} \longrightarrow_l v_i, \sigma, \texttt{CT}}$$

Seq
$$\frac{e_1, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(v, \sigma', \texttt{CT}')}{e_1; e_2, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(e_2, \sigma', \texttt{CT}')}\ \ \vec{u} \notin \mathsf{fn}(e_2) \cup \mathsf{fv}(e_2)$$

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

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

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

Cong
$$\frac{e, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(e', \sigma', \texttt{CT}')}{E[e], \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(E[e'], \sigma', \texttt{CT}')}\ \ \vec{u} \notin \mathsf{fn}(E) \cup \mathsf{fv}(E)$$

The predicate $\mathsf{insync}(o, E)$ is true if there exist E_1 and E_2 such that $E = E_1[\texttt{insync}\ o\ \{E_2[\ ]\}]$. We also use the following functions. Let $\sigma(o) = (C, \vec{f} : \vec{v}, n, \{\vec{c}\})$.

$$\text{read/update the counter} \begin{cases} \mathsf{setLock}(\sigma, o, n') & = \sigma[o \mapsto (C, \vec{f} : \vec{v}, n', \{\vec{c}\})] \\ \mathsf{getLock}(\sigma, o) & = n \end{cases}$$

$$\text{read/update the queue} \begin{cases} \mathsf{blocked}(\sigma, o) & = \{\vec{c}\} \\ \mathsf{block}(\sigma, o, c) & = \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \cup \{c\})] \\ \mathsf{unblock}(\sigma, o, \vec{c}') & = \sigma[o \mapsto (C, \vec{f} : \vec{v}, n, \{\vec{c}\} \setminus \{\vec{c}'\})] \end{cases}$$

[Synchronisation]

Fork
$E[\texttt{fork}(e)] \,|\, Q, \sigma, \texttt{CT} \longrightarrow_l E[\varepsilon] \,|\, \texttt{forked}\ e \,|\, Q, \sigma, \texttt{CT}$

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

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

Wait
$$\frac{\mathsf{insync}(o, E) \quad \mathsf{getLock}(\sigma, o) = n \quad \mathsf{setLock}(\sigma, o, 0) = \sigma'' \quad \mathsf{block}(\sigma'', o, c) = \sigma'}{E[o.\texttt{wait}] \,|\, Q, \sigma, \texttt{CT} \longrightarrow_l (\nu\,c)(E[\texttt{waiting}(c)\ n] \,|\, Q, \sigma', \texttt{CT})}$$

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

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

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

Ready
$$\frac{\mathsf{getLock}(\sigma, o) = 0 \quad \mathsf{setLock}(\sigma, o, n) = \sigma'}{\texttt{ready}\ o\ n, \sigma, \texttt{CT} \longrightarrow_l \varepsilon, \sigma', \texttt{CT}}$$

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

[Errors]

Err-Null
$\texttt{null.f}, \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$
$\texttt{null.f} = v, \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$
$\texttt{null.m}(v), \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$

Err-Monitor
$$\frac{\neg\mathsf{insync}(o, E)}{\begin{array}{c} E[o.\texttt{notify}], \sigma, \texttt{CT} \longrightarrow_l E[\texttt{Error}], \sigma, \texttt{CT} \\ E[o.\texttt{notifyAll}], \sigma, \texttt{CT} \longrightarrow_l E[\texttt{Error}], \sigma, \texttt{CT} \\ E[o.\texttt{wait}], \sigma, \texttt{CT} \longrightarrow_l E[\texttt{Error}], \sigma, \texttt{CT} \end{array}}$$

[Threads]

RC-Par
$$\frac{P_1, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1', \sigma', \texttt{CT}')}{P_1 \,|\, P_2, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1' \,|\, P_2, \sigma', \texttt{CT}')}\ \ \vec{u} \notin \mathsf{fn}(P_2) \cup \mathsf{fv}(P_2)$$

RC-Str
$$\frac{F \equiv F_0 \longrightarrow_l F_0' \equiv F'}{F \longrightarrow_l F'}$$

RC-Res
$$\frac{(\nu\,\vec{u})(P, \sigma, \texttt{CT}) \longrightarrow_l (\nu\,\vec{u}')(P', \sigma', \texttt{CT}')}{(\nu\,u\vec{u})(P, \sigma, \texttt{CT}) \longrightarrow_l (\nu\,u\vec{u}')(P', \sigma', \texttt{CT}')}$$

[Network]

RN-Conf
$$\frac{F \longrightarrow_l F'}{l[F] \longrightarrow l[F']}$$

RN-Par
$$\frac{N \longrightarrow N'}{N \,|\, N_0 \longrightarrow N' \,|\, N_0}$$

RN-Res
$$\frac{N \longrightarrow N'}{(\nu\,u)N \longrightarrow (\nu\,u)N'}$$

RN-Str
$$\frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$