Formal Analysis of a Distributed Object-Oriented Language and Runtime

Alexander Ahern and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. Distributed language features form an important part of modern object-oriented programming. In spite of their prominence in today's computing environments, the formal semantics of distributed primitives for object-oriented languages have not been well-understood, in contrast to their sequential part. This makes it difficult to perform rigorous analysis of their behaviour and develop formally founded safety methodologies. As a first step to rectify this situation, we present an operational semantics and typing system for a Java-like core language with primitives for distribution. The language captures the crucial but often hidden concerns involved in distributed objects, including object serialisation, dynamic class downloading and remote method invocation. We propose several invariant properties that describe important correctness conditions for distributed runtime behaviour. These invariants also play a fundamental rôle in establishing type safety, and help bound the design space for extensions to the language. The semantics of the language are constructed modularly, allowing straightforward extension, and this is exploited by adding primitives for direct code distribution to the language: thunk passing. Typing rules for the new primitives are developed using the invariants as an analysis tool, with type soundness ensuring that their inclusion does not violate safety guarantees.

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 interface for coordination. Distributing an application in this way confers many advantages to the programmer, such as resource sharing, load balancing, and fault tolerance [8]. Remote procedure call helps simplify such engineering practice by attempting to offer a seamless integration of network resource access and local procedure calls.

The Java programming language is a popular choice for developing such systems, with a highly dynamic and customisable class loading mechanism at its heart. Applets are a widespread example of code mobility derived from the use of custom class loaders—when a user visits a page containing a Java applet, the virtual machine running inside their web browser can automatically download and link the required classes without user intervention. Obtaining classes automatically is fundamental to Java Remote Method Invocation [26] (RMI). This is widely adopted for the Java platform and offers the programmer a straightforward mechanism for accessing shared remote resources. It fully exploits the customisable class loading system of the underlying language to

allow code to propagate around the network. 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.

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, for the very reason of maintaining seamless integration with the original language. For example, not all objects in RMI are passed by reference (unlike local invocation): only those of classes implementing the `Remote` marker interface are. All other objects are passed by value, which invokes the serialisation mechanism of the language. Similarly, if an object of a class finds its way to a particular location that does not have the byte-code for that class, it uses a customised class loading mechanism to attempt to obtain the class from the network. Moreover, verifying that the received class is safe to use may require the downloading of many others (such as the direct superclass, classes mentioned in method bodies and so on). Typically the programmer is only aware of these actions when something fails.

The presence of these hidden features at runtime makes the behaviour of a program using idioms for distributed computation much more complex and less understandable than a sequential one. Due to this complexity, the need to identify a precise semantics for distributed primitives is arguably even greater. By having a good semantic account for distribution, we may benefit from many of the useful outcomes of similar formal studies in the context of sequential language features. For example, we can build a basis for reasoning and verification of mobile code (e.g. applets, agents); we can use the resulting theoretical framework for the analysis of new language constructs and engineering methodologies; and we can discuss optimisations based on a clear semantic understanding. Further, and perhaps most importantly, a good semantic account of the features makes the life of the systems designer and implementor easier by providing a reference framework for implementation.

This paper aims to give such a formal semantics for a core Java-like language with basic programming primitives for distribution, extending existing core languages [18, 4]. A central challenge to modelling distributed features is to give a representation of runtime behaviour that reflects the "hidden" elements discussed above. This includes class downloading and serialisation; both are features that contribute to the observable behaviour of a program. Another key activity, matching return points for a remote method call, is non-trivial because of unavoidable interleaving of operations in a system with multiple locations, even if the underlying programmer-level language does not explicitly mention them. We show that a succinct representation of these runtime behaviours is indeed possible, partly using techniques from the π [28] and Higher-Order π-calculi [32]. One of the highlights is the use of a *linear* type discipline [21, 15] to ensure correct intermediate state of RMI and its returns. The semantics avoids becoming tied to a particular implementation of RMI by abstracting away concerns such as the methods for resource binding or the use of stubs and skeletons (this is sensible since implementation strategies change over time: for example, the need for skeletons was removed in homogeneous, all-Java, systems when JDK version 1.1 moved to version 1.2). The semantics is flexible and modular: we distill the key features of class downloading and serialisation separately, so that important design choices (for example eager/lazy

class downloading) can be easily reflected in the semantics. Our hope is that the formal semantics we present here will offer a clear high-level understanding of the behaviour of distributed extensions to Java-like languages, as well as contributing a starting point for the further development and application of formal semantics to such languages.

Applying the proposed formal semantics, we first establish the most basic safety property for any typed formalism: type soundness. Establishing type safety is highly non-trivial, making use of many *invariant properties* which runtime configurations should satisfy over time. For instance, one simple example of an invariant says that if an expression contains a reference to an instance of a local class, then that object should be stored in the same physical location as the code in question. The invariants capture basic safety criteria for distributed states that we believe sensible implementations should conform to. They can be used as a 'sanity check' during the design of typing rules: if an invariant is not guaranteed by reduction then it tends to point to a concrete issue in typing rules as well as a possible remedy. The use of invariants in proving safety guarantees goes beyond type safety: as a simple example, we show progress properties can also be derived from combinations of invariants.

As another use of the proposed formal machinery, we show how our framework can incorporate a new primitive for distributed object-oriented programming, *thunk passing*, which passes fragments of code as a value in communication for later execution. While well-known in languages like Scheme, Lisp and MetaML [1], and while having significant usability in distributed object-oriented programming, we do not know a consistent, type-safe language-level incorporation of this primitive for distributed Java-like languages. Indeed, a typing rule for the primitive strong enough to ensure type safety is far from obvious because of its interaction with the existing distributed primitives. To find a sound typing rule, we use invariants as a tool for analysis (for example, thunk passing leaks references to local objects over the network under a simple minded typing rule: the violation directly suggests how the rule can be strengthened). The resulting typing rule guarantees preservation of fundamental properties such as type safety and progress in the extended language. We also experiment with the incorporation of distributed failure to demonstrate extensibility of our framework.

We summarise our major technical contributions below.

- Introduction of a core calculus for a class based Java-like typed object-oriented programming language with basic primitives for distribution, including dynamic class downloading and serialisation. Its formal semantics centres on the representation of runtime distributed configurations, uses techniques from process calculi, and treats different design choices modularly, such as eager/lazy class downloading.
- A new technique to systematically prove type safety in distributed formalisms using network invariants. The invariants represent healthiness conditions of distributed states. Not only are they essential for proving type safety but also they are a useful analytical tool for developing consistent typing rules.
- An integration of thunk-passing primitives into distributed object-oriented programming, and establishment that their integration with RMI and class downloading mechanisms preserve type safety and progress. The typing rules for these primitives are developed through the analysis of their interplay with invariants.

In the remainder, Section 2 informally introduces the language we study in the present work. Section 3 introduces its syntax. Section 4 lists the auxiliary definitions. Sections 5 defines the operational semantics. Section 6 lists examples of the operational semantics; two of them give a detailed analysis of behaviour of the programs listed in Section 2. Section 7 defines the typing system. Section 8 proves the basic properties of the typing system. Section 9 develops the invariants and progress properties. Section 10 proves the subject reduction using the invariants. Section 11 discusses related work. Section 12 concludes the paper with further topics. Appendix lists all rules for the operational semantics and the typing system for reference.

2 Program Examples

This section illustrates the basic ideas of the small distributed Java-like language we study in the present paper, using concrete programming examples. Consider the following class definition:

```
1 class Server {int doTask(Task t){ return t.compute(); } }
```

A server can be created by instantiating `Server`. The idea is simple—a site with powerful computing resources can accept, via RMI calls, `Tasks`, which specify the work the compute server should perform on behalf of the supplier. Its code is as follows:

```
1 class Task { int compute(){ return 0; } }
```

In our simple setting we assume tasks return integers, but one could easily say that a special `TaskResult` class might be created for this purpose. Upon receipt of an instance of the `Task` class (or a subclass of it), the server calls the `compute()` method on it. The client to such a server would look something like the following:

```
1  class Client {
2   Server s;
3   Client(Server s) { this.s = s; }
4   int gcd(int a, int b) { return s.doTask(new GcdTask(a,b)); }
5  }
6  class GcdTask extends Task {
7   int a,b;
8   GcdTask(int a, int b) { this.a = a; this.b = b; }
9   int compute() {
10    int r = 0;
11    while (true) {
12     if (this.b == 0) {
13      return this.a;
14     } else {
15      r = this.a % this.b;
16      this.a = this.b;
17      this.b = r;
18 }}}}
```

Here, the client wishes the server to compute the greatest common divisor of two integers using an iterative version of Euclid's algorithm. To do so, the programmer creates a subclass `GcdTask` with the correct body overriding the `compute()` method of class `Task`. The client requests work be done by the server by making the remote method call `s.doTask(t)` and awaiting the result.

The behaviour of the above program, although conceptually simple, contains significant elements which cannot be captured in purely sequential (or non-distributed) formalisms. Firstly, the server site may not have a copy of the class `GcdTask`: in which case, it is standard practice in RMI that the site will request a copy of the class binary (byte code) from the client, invoking remote communication. In contrast, non-distributed formalisms assume all classes are available locally. A second observation is that, when sending the instance of the `GcdTask` to a server, if that class does not implement the `Remote` interface, then the sender site invokes the serialisation mechanism. Again, a non-distributed formalism does not have to consider passing objects by value, hence ignore this aspect. Semantically, however, the distinction between remote references and local references is essential in RMI (in Java and other similar languages), so this distinction and associated behaviour should be modelled. Finally, in the above example, it is natural to expect that many clients could be handled by a single server, so the formalism should support concurrency. Formal semantics of distributed objects must inevitably include runtime behaviour, such as a remote invocation in transit, a return message in transit, and a request for a class to be downloaded and a class being delivered. Unlike in sequential formalisms, representation of these intermediate states, or runtime, is a fundamental part of the formal semantics. We shall later present a simple way to represent them borrowing ideas from a (typed) HOπ-calculus.

The class downloading mechanisms in RMI realise a natural method for transferring code over network. The design space for code passing is however not limited to them. Another construct which complements class downloading may realise code mobility more explicitly, in the form known as *thunk passing* [1]. Later sections will examine the introduction of this primitive from a formal viewpoint. Here let us illustrate this primitive and its significance using the previous example of a task server. As mentioned, if the server does not have the class `GcdTask`, it must download it. Not only does this add an extra burden of making sure that any class the server might need to perform its job is available in a directory accessible via the network; it also leads to low efficiency and high rate of failures. Suppose there is a deep inheritance hierarchy above `GcdTask`. Then the server, after obtaining the class, must fetch all of its superclasses as needed. This can require several trips across the network, increasing the risk of failure and adding latency. Further, the "compute server" is tightly coupled with the notion of a `Task`—any work that a client wishes to do must be cast in this framework. These issues arise because class downloading does not allow direct, fine-grained control of code passing by programming.

Primitives for the creation and execution of a thunk, which we write $\texttt{freeze}[t](e)$ and $\texttt{defrost}(e)$, solve this issue. First we give the code for the server:

```
1 class ThunkServer {
2   int compute(thunk(int) t) { return defrost(t); }
3 }
```

As one can see, it makes no mention of any class—its only stipulation is that the thunk in question, when *defrosted* (read: evaluated), returns something of type `int`. That is not to say that the body of the thunk cannot use any other classes, but the server developer is not tied to any particular representation of the work to be done. Next we show the code for the modified client:

```
1  class ThunkClient {
2   ThunkServer s;
3   ThunkClient(ThunkServer s) { this.s = s; }
4   int gcd(int a, int b) {
5    thunk(int) g =
6     freeze[t](
7      int x = a;
8      int y = b;
9      int r = 0;
10     while (true) {
11      if (y == 0) {
12       return x;
13      } else {
14       r = y % x;
15       x = y;
16       y = r;
17      }});
18   return s.compute(g);
19 }}
```

Here, the client makes use of the $\texttt{freeze}[t](e)$ expression of the language. Instead of sending a class embodying the code for Euclid's algorithm, the client first creates a frozen representation of that algorithm, sending this instead. The only minor difference between the body of the `compute()` method of `GcdTask` and the thunk is at lines 7 and 8. These two lines simply copy the formal method parameters `a` and `b` into the body of the thunk, so that at the executing site it computes the correct result. Then, the client supplies the server with the thunk `g` which the server can defrost and run.

Given this primitive, code passing becomes fully controllable at the user-level: the server is no longer tied to a particular convention for the shape of tasks, while the client does not need to create a new class for each individual task. Most importantly, it offers a natural mechanism for a client to explicitly specify how a piece of code is passed to a server, using distinct tag t for thunking which we shall explain in Section 3. The cost of expressiveness is subtlety in its typing, which we shall explore in Section 7.

Finally, good formal semantics should precisely and concisely capture semantic differences between significant design choices. One such choice arises in the way class downloading is executed. In *eager* class downloading, the code for all associated classes will be sent immediately once and for all. In *lazy* class downloading (which the current standard implementation employs), each associated class will be fetched one by one as it becomes necessary at a remote site. This distinction has significant consequences in failure semantics as well as in efficiency. It is thus desirable that the formal semantics can cleanly capture these two ideas, just as the formal semantics of sequential languages can cleanly represent both call-by-name and call-by-value calling conventions.

3 Language

This section presents the formal syntax of the language, which we call DJ. We consider a configuration consisting of multiple hosts (virtual machines) for a single high-level class-based language. Inter-host communication is by remote method invocation. Parameters to remote methods may include base values, references to remote objects, and most interestingly, references to objects local to the caller. During a remote invocation, the parameters are implicitly marshaled and unmarshaled by the sending and receiving sites. In brief, marshaling is the process of transforming parameters from their "in memory" representation into one suitable for transmission over the network. Unmarshaling is the dual to this. We allow the explicit marshaling and unmarhsaling of data and arbitrary programs carrying local code (classes). Related to all these features is the provision for automatic, dynamic class downloading from the remote site. The language demonstrates the key aspects of RMI and code distribution in Java and the CLR [19, 7], making its operational semantics non-trivial. In spite of the simple syntax extension for distribution, its operational semantics are largely governed by runtime behaviour hidden from the programmer. Hence the language DJ is formalised in two kinds of syntax—that which is used for writing programs at each local site, which we call *user syntax*, and that which occurs only at runtime as intermediate forms, which we call *runtime syntax*.

3.1 User syntax

We first introduce the user syntax in Fig. 3.1. The syntax is an extension of FJ [18] and MJ [4] augmented with basic primitives for distribution, including those for thunks discussed in Section 2.

The metavariables T and U range over expression and statement types of the language. T represents expression types: booleans (these are the only base values considered), class names (ranged over by C, D, E, F), thunked expressions of type U and serialised objects of type C. The metavariable U ranges over the same types as T but is augmented with the special type `void` with the usual empty meaning.

Class declarations are ranged over by L, constructors by K and method declarations by M. f ranges over field names, m ranges over method names and x is used to denote both local variables and formal parameters. $\vec{f}$ denotes a vector of fields, and $\vec{T}\vec{f}$ is short-hand for a sequence of typed field declarations: $T_1 f_1; \dots ; T_n f_n$. We apply similar abbreviations to other sequences. We assume sequences contain no duplicate names.

The declaration $\texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f};\, K\vec{M}\}$ introduces a class named C with a direct superclass D. It has fields $\vec{f}$ with types $\vec{T}$, a constructor K and several methods written $\vec{M}$. In the Java language, fields may be redeclared in a subclass, with the new definition shadowing that found in the superclass. To simplify, we assume that any fields declared in C will have different names to any fields declared in superclasses, an approach also adopted in FJ. However, we do allow the overriding of methods with the same names by a subclass and the addition of new methods.

Constructors are written in the form $C\,(\vec{T}\vec{f})\{\texttt{super}(\vec{f});\texttt{this}.\vec{\texttt{f}} := \vec{f}\}$. This initialises a new instance of class C by first initialising the fields of the superclasses via a call to `super`. The fields declared in C itself are then initialised via a sequence of

$T ::= \texttt{bool} \mid C \mid \texttt{thunk}(U) \mid \texttt{ser}(C)$	Types
$U ::= \texttt{void} \mid T$	
$L ::= \texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}$	Classes
$K ::= C\ (\vec{T}\vec{f})\{\texttt{super}(\vec{f}); \texttt{this}.\vec{\texttt{f}} := \vec{f}\}$	Constructors
$M ::= U\,\texttt{m}(\vec{T}\vec{x})\{e\}$	Methods
$e ::= v \mid x \mid \texttt{this} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e.\texttt{f} \mid e;e \mid T\,x = e$	Expressions
$\mid pe \mid \texttt{return}\ e \mid \texttt{return} \mid \texttt{serialize}(e) \mid \texttt{deserialize}(e)$	
$\mid \texttt{freeze}[t](e) \mid \texttt{defrost}(e)$	
$pe ::= x := e \mid e.\texttt{f} := e \mid \texttt{new}\ C(\vec{e}) \mid e.\texttt{m}(\vec{e})$	Promotable
$v ::= \texttt{true} \mid \texttt{false} \mid \texttt{null}$	Values
$t ::= \texttt{eager} \mid \texttt{lazy}$	Tags
$\mathsf{CSig} ::= \emptyset \mid \mathsf{CSig} \cdot C : \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T}_i \to U_i\}$	Class Signatures

Fig. 3.1. User syntax

the form $\texttt{this}.\vec{\texttt{f}} := \vec{f}$. There must be precisely the correct number of parameters in the constructor declaration as there are fields in the class and all superclasses. This ensures correct initialisation.

Methods are declared as $U\,\texttt{m}\,(\vec{T}\vec{x})\{e\}$. This denotes a method called m that returns a value of type U. It takes parameters $\vec{x}$ with types $\vec{T}$. The body of the method is represented by expression e. Values ranged over by v have standard meaning, as do expressions, e, except for the following important new primitives.

Expressions in the language are denoted by e. The syntax of expressions is standard except the two pairs of distributed primitives. It consists of values, variables (ranged over $x, y, ..$), branchings, field accesses, sequencing and local variable declarations. We also have a special class of promotable expressions denoted pe [4, 9]; computation of any expression results in a value, but we allow promotable expressions to be used where their return value is not needed, such as in sequences of expressions.

We now introduce the two pairs of distributed primitives. The first pair is for serialisation. $\texttt{serialize}(e)$ takes the value computed by the expression e and serialises it. This produces a new value that is suitable for transfer over the network (in Java, serialisation writes to a byte stream [25]: while the original form is treatable, the chosen syntax leads to a simpler presentation of typing rules). $\texttt{deserialize}(e)$ takes the serialised value computed by expression e and converts it back into a structured form.

The other pair of primitives for distribution are for creating thunks. $\texttt{freeze}[t](e)$ takes the expression e and, without evaluating it, produces a frozen representation of its code. The expression is not evaluated, but is stored for later use as a value. The tag t is a flag to control the amount of class information sent along with e by the user. If s/he specifies `eager`, then the code is automatically frozen together with all classes that *may* be used. If `lazy` is specified, it is the responsibility of the receiving virtual machine to

obtain missing classes. Dual to freezing, the action $\texttt{defrost}(e)$ expects the evaluation of expression e to produce a piece of frozen code, which will then be executed.

v ranges over values, which consists with boolean constants and `null`.

Finally, a class signature CSig is a mapping from class names to their interface types (or signatures). We assume CSig is given globally (this does not lose generality since uniqueness of each class is maintained through its digital signature in standard implementations), 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 (the formal definition appears in Definition 4.7). 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.

For simplicity, we omit casting [18, 4], exceptions [3], synchronisation [12] and multiple inheritance; adaptations with these features are straightforward.

3.2 Runtime syntax

The runtime syntax in Fig. 3.2 extends the user syntax and represents a distributed state of multiple sites communicating with each other, including remote operations in transit.

$e ::= \cdots \mid \texttt{new}\ C^l(\vec{v}) \mid \texttt{download}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e \mid \texttt{resolve}\ \vec{C}\ \texttt{from}\ l\ \texttt{in}\ e$	Expressions
$\quad \mid \quad \texttt{await}\ c \mid \texttt{Error}$	
$v ::= \cdots \mid o \mid \ulcorner e\ \texttt{with}\ \texttt{CT}\ \texttt{from}\ l \urcorner \mid \lambda \vec{o}.(\vec{v}, \sigma, l) \mid \varepsilon \mid \vec{v}$	Values
$u ::= x \mid n$	Identifiers
$n ::= o \mid c$	Names
$P ::= \mathbf{0} \mid P_1 \mid P_2 \mid (\nu\, u)P$	Threads
$\quad \mid \quad \texttt{go}\ e\ \texttt{with}\ c \mid e\ \texttt{with}\ c \mid \texttt{return}(c)\ e \mid \texttt{go}\ e\ \texttt{to}\ c \mid \texttt{Error}$	
$F ::= (\nu\, \vec{u})(P, \sigma, \texttt{CT})$	Configurations
$N ::= \mathbf{0} \mid l[F] \mid N_1 \mid N_2 \mid (\nu\, u)N$	Networks
$\sigma ::= \emptyset \mid \sigma \cdot [x \mapsto v] \mid \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})]$	Stores
$\texttt{CT} ::= \emptyset \mid \texttt{CT} \cdot [C \mapsto L]$	Class Tables

Fig. 3.2. Runtime Syntax

The syntax uses *location names* $l, m, \ldots$ which can be thought of as IP addresses in a network. *Tagged class names*, written C^l, indicate that the body of class C can be obtained from location l. Typically when code is shipped around the network, class names will be decorated with these labels so that consumers of expressions can safely

request resources they need for execution [26, §3.4]. We write C^{-} when the treatment of class name C is independent of whether it is tagged or not.

The first three extended expressions are used to define the machinery for class downloading. The tagged class creation, "new $C^{l}(\vec{v})$", is going to download the class C from l before executing new operator. "download $\vec{C}$ from l in e" downloads classes $\vec{C}$ from location l before executing e; "resolve $\vec{C}$ from l in e" checks superclasses of $\vec{C}$ at the location l. These facilitate automatic class downloading and intuitively adopt the task of the RMIClassLoader class [26, §5.6].

"await c" is used for RMI, which is explained with threads. The distinguished expression Error is the result of a computation that de-references a null pointer.

The fourth line extends values. Object identifiers $o,\ldots$ denote references to instances of classes as well as the destination of an RMI call. We shall frequently write "o-id" for brevity. Channels $c,\ldots$ 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*, ranged over n.

Identifiers, u, range over names n and variables x.

The second extended value is a *thunked expression* (or *thunk* for brevity), a "frozen" piece of code that can be passed between methods as a value. Later, it can be "defrosted" at which point it is executed to compute a value. $\ulcorner e$ with CT from $l \urcorner$ denotes an expression e frozen with class table CT from l. 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.

The third extended value $\lambda\vec{o}.(\vec{v},\sigma,l)$ is a serialised array (or *blob* for brevity), which is a flattened data representation suitable for transfer over the network. In $\lambda\vec{o}.(\vec{v},\sigma,l)$, $\vec{v}$ contains the values for transfer. Each $v_i \in \vec{v}$ is treated differently according to its nature. If v_i is a base value or the identifier of a remote object then it is included without special treatment. However for each $o_i \in \vec{v}$, where o_i is an identifier of a local object then this object must be flattened for transfer. This creates the object graph σ, containing all the local objects transitively referenced by each o_i. The final component of a blob is the location name l, indicating where it was created. The fourth extended value is the empty value ε used as a placeholder for a return value by methods declared as void.

Threads, ranged over by P,Q,R, comprise several expressions that can be evaluated in parallel $P\,|\,Q$ and also the new name operator $(\nu\, u)P$ for restricting identifier u (which should not be confused with new object creation). $\mathbf{0}$ denotes an empty thread. This notation comes from the π-calculus [28]. It also includes Error which denotes the result of communication failure. The last four primitives are essential to represent the RMI mechanism. c is the channel created at runtime when we spawn a thread for RMI. "await c" is a placeholder waiting for the result of RMI at c. "go e with c" and are "e with c" are messages to access remote methods carrying "return(c) e" is the result of the method invocation; and "go e to c" is a message going back to "await c" in the remote location. The detailed explanation of these primitives is given together with operational semantics in § 5.6.

We represent an instance of a virtual machine by a *configuration*, ranged over by metavariable F. A configuration is written $(\nu\,\vec{u})(P,\sigma,\mathtt{CT})$ and consists of some threads, P, a store σ containing local variables and objects and a class table written CT. We

surround the configuration by a possibly empty vector of restricted names, $\nu\vec{u}$ which limits the scope of local names and variables.

Networks, written N, comprise zero or more configurations executing in parallel. $\mathbf{0}$ denotes the empty network. $l[F]$ denotes a configuration F executing at location l. $N_1|N_2$ and $(\nu\vec{u})N$ are understood as in threads. The scope of restricted identifiers can be opened across different located configurations using the structural equivalence rules explained later to permit remote method invocation.

Finally *stores*, ranged over by σ, σ'..., consist of a mapping from variable names to values, written $[x \mapsto v]$, or from object identifiers to store objects, written $[o \mapsto (C, \vec{f} : \vec{v})]$ indicating that identifier o maps to an object of class C with a vector of fields with values $\vec{f} : \vec{v}$. Class Tables, $\mathtt{CT}$, are a mapping from undecorated class names to class definitions (metavariable L in Fig. 3.1).

4 Auxiliary definitions

The operational semantics, typing rules and structural equivalences of DJ depend on several auxiliary definitions. These are explained in this section. The most important definitions are the object graph in Definition 4.8 and the class graph in Definition 4.12, which are used to formulate the operational semantics for serialisation and code-mobility respectively. The reader can skip this section and come back when necessary.

Definition 4.1 (Domains). The functions $\mathsf{dom_v}$ and $\mathsf{dom_o}$ compute the domain of variable mappings and object mappings of a term respectively. They are inductively defined over networks N, configurations F and stores σ, and are given in Fig. 4.1. We adopt the convention $\mathsf{dom}(\sigma) = \mathsf{dom_v}(\sigma) \cup \mathsf{dom_o}(\sigma)$. Similarly for F and N.

The store at a location can be split into two parts—cells containing the values of variables and cells containing objects. The domains of a given store σ are denoted $\mathsf{dom_v}(\sigma)$ and $\mathsf{dom_o}(\sigma)$ for the variable and object parts respectively. The domains of a configuration are defined as the domains of the store part of that configuration, minus any names restricted by it, while that of a network are defined as the sum of the domains of all the configurations making up the network, minus any names restricted at the network level.

Definition 4.2 (Lookup functions). In Fig. 4.2, we provide several functions for determining the types of fields and methods for a particular class, and for retrieving the code that forms the body of methods. The distinguished class $Object$ contains no fields or methods, and so forms the base for recursive definitions.

Fields. The fields of a class C, written $\mathsf{fields}(C)$ yields a sequence $\vec{T}\vec{f}$ where f_i is the name of a field and T_i is the type of that field according to the class signature for C. We write $\bullet$ to represent the empty sequence of fields.

Method interface. The type of a particular method m in class C according to the class signature is given by the function $\mathsf{mtype}(\mathtt{m}, C)$, and is denoted $\vec{T} \rightarrow U$ to indicate that the method takes a sequence of parameters of type $\vec{T}$ and returns a value of type U. This corresponds to the interface of the method.

	$\mathsf{dom_v}$	$\mathsf{dom_o}$
Configurations		
$(\nu\,\vec{u})(P,\sigma,\mathtt{CT})$	$\mathsf{dom_v}(\sigma)\setminus\mathsf{fv}(\vec{u})$	$\mathsf{dom_o}(\sigma)\setminus\mathsf{fn}(\vec{u})$
Networks		
$\mathbf{0}$	$\emptyset$	$\emptyset$
$l[F]$	$\mathsf{dom_v}(F)$	$\mathsf{dom_o}(F)$
$N_1 \mid N_2$	$\mathsf{dom_v}(N_1)\cup\mathsf{dom_v}(N_2)$	$\mathsf{dom_o}(N_1)\cup\mathsf{dom_o}(N_2)$
$(\nu\,u)N$	$\mathsf{dom_v}(N)\setminus\mathsf{fv}(u)$	$\mathsf{dom_o}(N)\setminus\mathsf{fn}(u)$
Stores		
$\emptyset$	$\emptyset$	$\emptyset$
$\sigma\cdot[x\mapsto v]$	$\{x\}\cup\mathsf{dom_v}(\sigma)$	$\mathsf{dom_o}(\sigma)$
$\sigma\cdot[o\mapsto(C,\vec{f}:\vec{v})]$	$\mathsf{dom_v}(\sigma)$	$\{o\}\cup\mathsf{dom_o}(\sigma)$

Fig. 4.1. Domains

Method body. Unlike lookup of fields and method types which rely on the lightweight CSig, the code supplying the body of a method is obtained from the local class table $\mathtt{CT}$. The function call $\mathsf{mbody}(\mathtt{m},C,\mathtt{CT})$ returns a pair $(\vec{x},e)$ where $\vec{x}$ denotes the formal parameters of the method and e is the body of the method.

Valid method overriding. As in the Java language, any overriding method in a subclass must have the exact same signature as declared for that method by the superclass. If $\mathsf{override}(\mathtt{m},D,\vec{T}'\to U')$ is defined then this means thats the new signature for method $\mathtt{m}$ in class D is such a correct overriding.

Definition 4.3 (Free class names). A class name C is defined as *free* if it is the subject of an instantiation operation (written $\mathtt{new}\ C(\ldots)$). The set of free class names for a given term is given by the function fcl which is defined over expressions, threads and class table entries. The free class names of a value v is defined as $\mathsf{fcl}(v)=\emptyset$. The free class names of an expression are defined recursively as the union of the free class names of all sub-expressions, with the exception that:

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

For threads, the definition is equally obvious:

$$\mathsf{fcl}(\mathbf{0})=\emptyset\qquad\mathsf{fcl}(P_1\mid P_2)=\bigcup\mathsf{fcl}(P_i)\qquad\mathsf{fcl}((\nu\,u)P)=\mathsf{fcl}(P)$$

Finally, for class table entries we retrieve the free class names appearing in the bodies of methods:

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

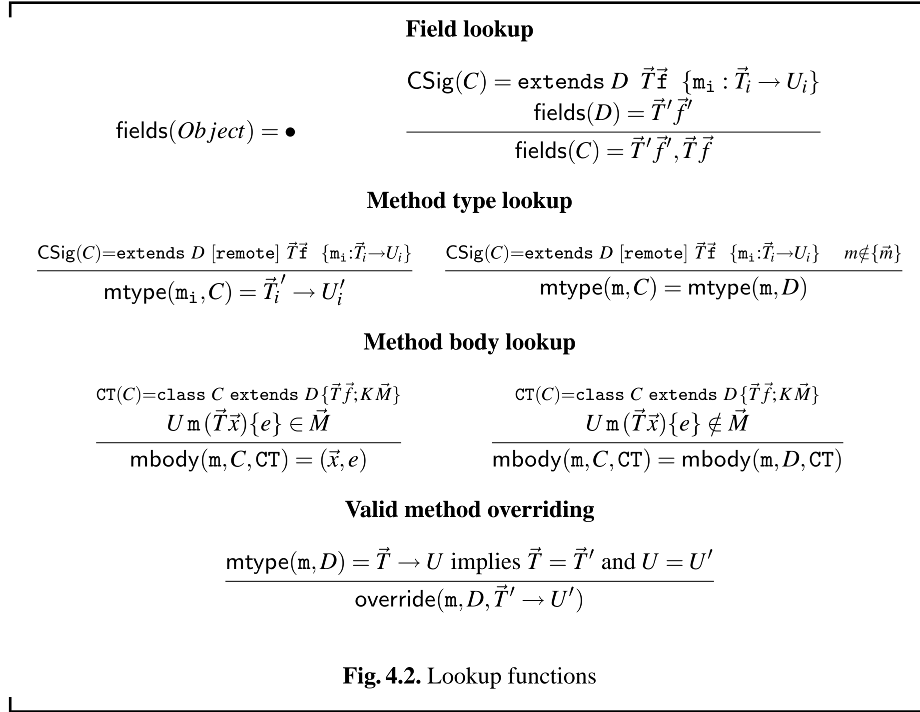


Fig. 4.2. Lookup functions

Definition 4.4 (Free variables and names). The functions for determining free variables fv and free names fn are defined in Fig. 4.3. The function $\mathsf{fnv}(N)$ is defined as $\mathsf{fv}(N) \cup \mathsf{fn}(N)$. Variables, represented by x, range over local variables in threads and variables mentioned in both the domain *and* the co-domain of store mappings.

Names are more complex—a name can be either an object identifier (usually written o) or a channel name (written c). Object identifiers can appear in both threads and stores (in both the domain and co-domain), whereas channels are only permitted at the thread level.

The following definition is used to define the rule for creating thunks.

Definition 4.5 (Free assigned variables). The function $\mathsf{fav}(e)$ returns the free assigned variables in expression e. Its definition is identical to that of $\mathsf{fv}(e)$ with the exception that:

$$\mathsf{fav}(x) = \mathsf{fav}(\mathtt{this}) = \emptyset$$

For example, $\mathsf{fav}(x := y) = \{x\}$, while $\mathsf{fv}(x := y) = \{x, y\}$. Other cases are omitted for brevity as they can easily be constructed from Fig. 4.3.

Definition 4.6 (Location names). The location names function $\mathsf{loc}(N)$ is defined over inductively over the structure of network N and returns, as a set, the names of the loca-

	fv	fn
Values		
$\texttt{true}, \texttt{false}, \texttt{null}, \varepsilon, \texttt{Error}$	$\emptyset$	$\emptyset$
o	$\emptyset$	$\{o\}$
$\ulcorner e \texttt{ with } \texttt{CT} \texttt{ from } l \urcorner$	$\mathsf{fv}(e) \cup \mathsf{fv}(\texttt{CT})$	$\mathsf{fn}(e) \cup \mathsf{fn}(\texttt{CT})$
$\lambda \vec{o}.(\vec{v}, \sigma, l)$	$\mathsf{fv}(\vec{v}) \cup \mathsf{fv}(\sigma)$	$(\mathsf{fn}(\vec{v}) \cup \mathsf{fn}(\sigma)) \setminus \{\vec{o}\}$
$\vec{v}$	$\bigcup \mathsf{fv}(v_i)$	$\bigcup \mathsf{fn}(v_i)$
Expressions		
x	$\{x\}$	$\emptyset$
$\texttt{this}$	$\{\texttt{this}\}$	$\emptyset$
$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2$	$\mathsf{fv}(e) \cup \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)$	$\mathsf{fn}(e) \cup \mathsf{fn}(e_1) \cup \mathsf{fn}(e_2)$
$e.\texttt{f}$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$e; e'$	$\mathsf{fv}(e) \cup \mathsf{fv}(e')$	$\mathsf{fn}(e) \cup \mathsf{fn}(e')$
$T\ x = e\ ;\ e'$	$\mathsf{fv}(e) \cup (\mathsf{fv}(e') \setminus \{x\})$	$\mathsf{fn}(e) \cup \mathsf{fn}(e')$
$x := e$	$\{x\} \cup \mathsf{fv}(e)$	$\mathsf{fn}(e)$
$e.\texttt{f} := e'$	$\mathsf{fv}(e) \cup \mathsf{fv}(e')$	$\mathsf{fn}(e) \cup \mathsf{fn}(e')$
$\texttt{new } C\text{-}(\vec{e})$	$\bigcup \mathsf{fv}(e_i)$	$\bigcup \mathsf{fn}(e_i)$
$e.\texttt{m}(\vec{e})$	$\bigcup \mathsf{fv}(e_i) \cup \mathsf{fv}(e)$	$\bigcup \mathsf{fn}(e_i) \cup \mathsf{fn}(e)$
$\texttt{return } e$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{return}$	$\emptyset$	$\emptyset$
$\texttt{serialize}(e)$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{deserialize}(e)$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{freeze}[t](e)$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{defrost}(e)$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{return}(c)\ e$	$\mathsf{fv}(e)$	$\{c\} \cup \mathsf{fn}(e)$
$\texttt{await } c$	$\emptyset$	$\{c\}$
$[\texttt{go}]\ e \texttt{ with } c$	$\mathsf{fv}(e)$	$\mathsf{fn}(e) \cup \{c\}$
$\texttt{go } e \texttt{ to } c$	$\mathsf{fv}(e)$	$\mathsf{fn}(e) \cup \{c\}$
$\texttt{download } C \texttt{ from } l \texttt{ in } e$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
$\texttt{resolve } C \texttt{ from } l \texttt{ in } e$	$\mathsf{fv}(e)$	$\mathsf{fn}(e)$
Configurations		
$(\nu\, n)F$	$\mathsf{fv}(F)$	$\mathsf{fn}(F) \setminus \{n\}$
$(\nu\, x)F$	$\mathsf{fv}(F) \setminus \{x\}$	$\mathsf{fn}(F)$
$(P, \sigma, \texttt{CT})$	$\mathsf{fv}(P) \cup \mathsf{fv}(\sigma) \cup \mathsf{fv}(\texttt{CT})$	$\mathsf{fn}(P) \cup \mathsf{fn}(\sigma) \cup \mathsf{fn}(\texttt{CT})$
Threads		
$\mathbf{0}$	$\emptyset$	$\emptyset$
$P_1 \mid P_2$	$\mathsf{fv}(P_1) \cup \mathsf{fv}(P_2)$	$\mathsf{fn}(P_1) \cup \mathsf{fn}(P_2)$
$(\nu\, n)P$	$\mathsf{fv}(P)$	$\mathsf{fn}(P) \setminus \{n\}$
$(\nu\, x)P$	$\mathsf{fv}(P) \setminus \{x\}$	$\mathsf{fn}(P)$
Networks		
$\mathbf{0}$	$\emptyset$	$\emptyset$
$l[F]$	$\mathsf{fv}(F)$	$\mathsf{fn}(F)$
$N_1 \mid N_2$	$\mathsf{fv}(N_1) \cup \mathsf{fv}(N_2)$	$\mathsf{fn}(N_1) \cup \mathsf{fn}(N_2)$
$(\nu\, u)N$	$\mathsf{fv}(N) \setminus \mathsf{fv}(u)$	$\mathsf{fn}(N) \setminus \mathsf{fn}(u)$
Stores		
$\emptyset$	$\emptyset$	$\emptyset$
$\sigma \cdot [x \mapsto v]$	$\{x\} \cup \mathsf{fv}(v) \cup \mathsf{fv}(\sigma)$	$\mathsf{fn}(v) \cup \mathsf{fn}(\sigma)$
$\sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})]$	$\mathsf{fv}(\vec{v}) \cup \mathsf{fv}(\sigma)$	$\{o\} \cup \mathsf{fn}(\vec{v}) \cup \mathsf{fn}(\sigma)$
Class Tables		
$\emptyset$	$\emptyset$	$\emptyset$
$U\, \texttt{m}(\vec{T}\vec{x})\{e\}$	$\mathsf{fv}(e) \setminus \{\vec{x}\}$	$\mathsf{fn}(e)$
$\texttt{class } C \texttt{ extends } D\{\vec{T}\vec{f}; K\vec{M}\}$	$\bigcup \mathsf{fv}(M_i)$	$\bigcup \mathsf{fn}(M_i)$
$\texttt{CT} \cdot [C \mapsto L]$	$\mathsf{fv}(L) \cup \mathsf{fv}(\texttt{CT})$	$\mathsf{fn}(L) \cup \mathsf{fn}(\texttt{CT})$

Fig. 4.3. Free variables and names

tions comprising a given network N. It is defined as follows:

$$\mathsf{loc}(\mathbf{0}) = \emptyset \qquad \mathsf{loc}(l[F]) = \{l\}$$
$$\mathsf{loc}(N_1 \,|\, N_2) = \mathsf{loc}(N_1) \cup \mathsf{loc}(N_2) \qquad \mathsf{loc}((\nu\, u)N) = \mathsf{loc}(N)$$

Definition 4.7 (Local and remote classes). The formal definition of remote and local classes is as follows: a type U is said to be local class if the $\mathsf{local}(U)$ predicate is true.

$$\mathsf{local}(U) = \begin{cases} \text{true} & \text{if } \mathsf{CSig}(U) = \texttt{extends}\ D\ \ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T}_i \rightarrow U_i\} \\ \text{false} & \text{otherwise} \end{cases}$$

If the $\mathsf{remote}(U)$ predicate is true, that type is said to be remote class.

$$\mathsf{remote}(U) = \begin{cases} \text{true} & \text{if } \mathsf{CSig}(U) = \texttt{extends}\ D\ \texttt{remote}\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T}_i \rightarrow U_i\} \\ \text{false} & \text{otherwise} \end{cases}$$

If neither predicate is true, the type is not a class.

The following definition is used for formulating the serialised object identifier.

Definition 4.8 (Object graph). The function $\mathsf{og}(\sigma, v)$ computes the object graph of value v in store σ. This 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. If the value v refers to a remote object, or a base value such as a boolean, then the object graph is empty. The algorithm is defined as follows:

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

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

It is easy to show that, given σ and v, the algorithm always terminates.

Example 4.9 (Object graph computation). Suppose: $\sigma = \sigma' \cdot [a \mapsto (A, f' : b, g : c)] \cdot [b \mapsto (B, f' : d, g : e)] \cdot [c \mapsto (C, f' : f, g : a)] \cdot [d \mapsto (D, f' : f, g : a)] \cdot [e \mapsto (E, f' : f)] \cdot [f \mapsto (F, \varepsilon)]$ where $\mathsf{dom}(\sigma') \cap \{a, b, c, d, e, f, g\} = \emptyset$. We assume D is remote and others are

local. We compute $\mathsf{og}(\sigma, a)$ as follows:

$$
\begin{aligned}
\mathsf{og}(\sigma, a) &= [a \mapsto (A, f' : b, g : c)] \cup \sigma_1 \\
&\cup\ \mathsf{og}(\sigma \setminus (\{a\} \cup \mathsf{dom_o}(\sigma_1)), c) \\
&\quad \text{where } \sigma_1 = \mathsf{og}(\sigma \setminus \{a\}, b) \\
\mathsf{og}(\sigma \setminus \{a\}, b) &= [b \mapsto (B, f' : d, g : e)] \cup \sigma_2 \\
&\cup\ \mathsf{og}(\sigma \setminus (\{a, b\} \cup \mathsf{dom_o}(\sigma_2)), e) \\
&\quad \text{where } \sigma_2 = \mathsf{og}(\sigma \setminus \{a, b\}, d) \\
\mathsf{og}(\sigma \setminus \{a, b\}, d) &= \emptyset \text{ as } \mathsf{remote}(D) \\
\mathsf{og}(\sigma \setminus \{a, b\}, e) &= [e \mapsto (E, f' : f)] \cup \mathsf{og}(\sigma \setminus \{a, b, e\}, f) \\
\mathsf{og}(\sigma \setminus \{a, b, e\}, f) &= [f \mapsto (F, \varepsilon)] \\
\mathsf{og}(\sigma \setminus (\{a\} \cup \mathsf{dom_o}(\sigma_1)), c) &= [c \mapsto (C, f' : f, g : a)] \cup \sigma_3 \\
&\cup\ \mathsf{og}(\sigma \setminus (\{a, b, e, f, c\} \cup \mathsf{dom_o}(\sigma_3)), a) \\
&\quad \text{where } \sigma_3 = \mathsf{og}(\sigma \setminus \{a, b, e, f, c\}, f) \\
\mathsf{og}(\sigma \setminus \{a, b, e, f, c\}, f) &= \emptyset \\
\mathsf{og}(\sigma \setminus (\{a, b, e, f, c\} \cup \mathsf{dom_o}(\sigma_3)), a) &= \emptyset
\end{aligned}
$$

Therefore we obtain:

$$
\begin{aligned}
\mathsf{og}(\sigma, a) = {} & [a \mapsto (A, f' : b, g : c)] \cdot [b \mapsto (B, f' : d, g : e)] \cdot [e \mapsto (E, f' : f)] \\
& \cdot [f \mapsto (F, \varepsilon)] \cdot [c \mapsto (C, f' : f, g : a)]
\end{aligned}
$$

Figure 4.4 shows the calculation of the above example where the arrow denotes a pointer to the store which contains the local class, while the dotted arrow shows that to the store which contains the remote class. The region denoted by the dashed lines represents the stores which are collected at each step.

We introduce the preliminary notion of *reachability*.

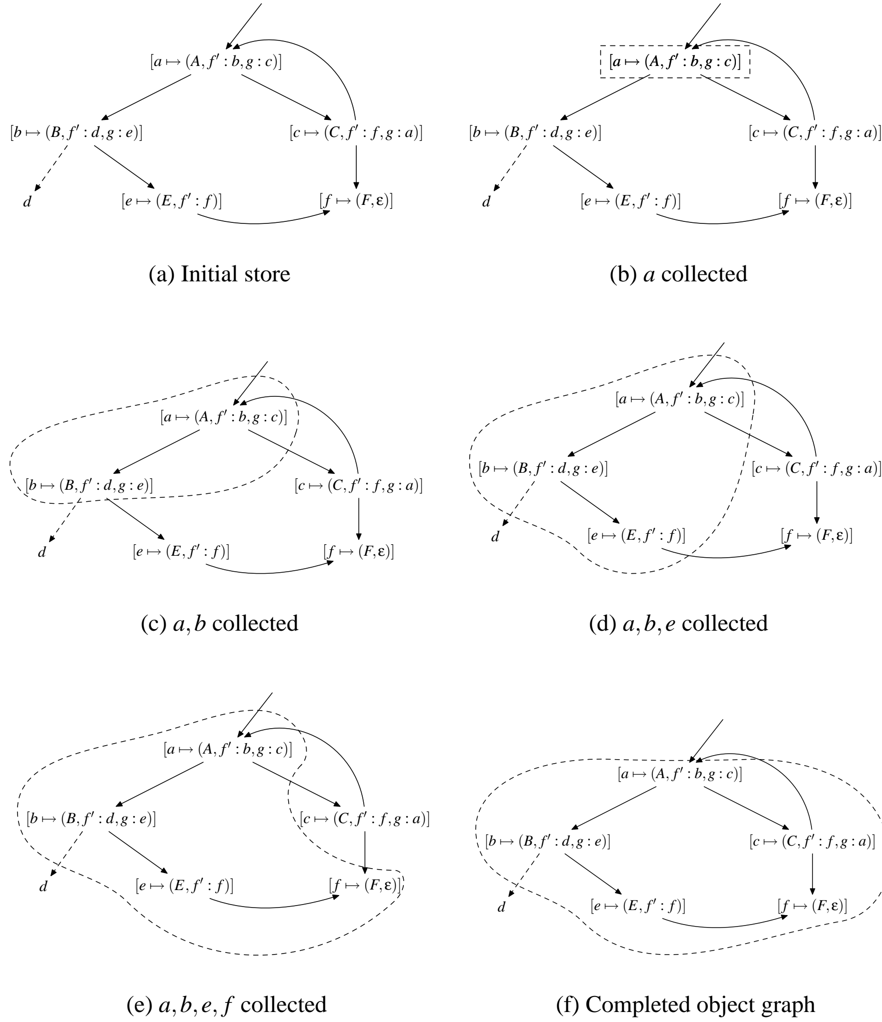
Definition 4.10 (Object graph reachability). The predicate $\mathsf{reachable}(\sigma, o, o')$ holds if there exists a path in store σ from the object with identifier o to the object with identifier o'. This can be an immediate link (when o' is stored in a field of o), or it can be via the fields of one or more intermediaries. This is defined below:

$$
\begin{aligned}
\mathsf{reachable}(\sigma, o, o') &\iff (o' \in \mathsf{fn}(\vec{v}) \vee \exists o'' \in \mathsf{fn}(\vec{v}).\mathsf{reachable}(\sigma, o'', o')) \\
&\quad \text{where } \sigma(o) = (C, \vec{f} : \vec{v})
\end{aligned}
$$

This predicate may be used to construct the relation RCH_σ containing all reachable pairs of objects in a store σ:

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

Definition 4.10 is important. Our object graph algorithm must, to be correct, preserve the tree structure of the store when copying objects. In other words, it must preserve this reachability relation. To determine correctness of the algorithm in Definition 4.8, we introduce the notion of a *complete object graph* in Definition 4.11.



(a) Initial store (b) a collected

(c) a,b collected (d) a,b,e collected

(e) a,b,e,f collected (f) Completed object graph

Fig. 4.4. Object graph example

Definition 4.11 (Complete object graph). For a store σ and an object graph σ_g computed from that store, the predicate $\mathsf{og_comp}(\sigma,\sigma_g)$ holds if computed graph preserves the reachability relation for all object identifiers in its object domain. Given RCH_σ and RCH_{σ_g} as in Definition 4.10:

$$\mathsf{og_comp}(\sigma,\sigma_g) \text{ if } \forall o \in \mathsf{dom_o}(\sigma) \cap \mathsf{dom_o}(\sigma_g).(o,o') \in RCH_\sigma \iff (o,o') \in RCH_{\sigma_g}$$

This property ensures all links are correctly copied to the graph σ_g, and no new links are created. An interesting point is that 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. Intuitively, safety is preserved since the recipient of such an object graph will merely add an unreachable element to its store (which could be immediately garbage collected). Of course, such behaviour may be inefficient because garbage data may be transmitted across the network.

The next definition is used to calculate a class table which is frozen when we create a thunked expression.

Definition 4.12 (Class graph). The function $\mathtt{cg}(\mathtt{CT}, T)$ computes the class graph of type T in class table $\mathtt{CT}$. It is defined as the set of all classes referenced transitively from the type T in $\mathtt{CT}$. A class name is referenced if it appears in any of the method bodies defined in a class, or if it is defined as the direct superclass. The algorithm is defined as follows:

$$
\begin{aligned}
\mathtt{cg}(\mathtt{CT}, \mathtt{bool}) &= \mathtt{cg}(\mathtt{CT}, \mathtt{void}) = \emptyset \\
\mathtt{cg}(\mathtt{CT}, \vec{M}) &= \bigcup \mathtt{cg}(\mathtt{CT}, M_i) \\
\mathtt{cg}(\mathtt{CT}, U\,\mathtt{m}(\vec{T}\vec{x})\{e\}) &= \mathtt{cg}(\mathtt{CT}, e) \\
\mathtt{cg}(\mathtt{CT}, C) &= \begin{cases} \emptyset & \text{if } C \notin \mathsf{dom}(\mathtt{CT}) \vee C \in \mathsf{dom}(\mathtt{FCT}) \\ \mathtt{cg}(\mathtt{CT}, \mathtt{CT}(C)) & \text{otherwise} \end{cases} \\
\mathtt{cg}(\mathtt{CT}, C^l) &= \emptyset \\
\mathtt{cg}(\mathtt{CT}, \vec{C}) &= \bigcup \mathtt{cg}(\mathtt{CT}, C_i) \\
\mathtt{cg}(\mathtt{CT}, e) &= \mathtt{cg}(\mathtt{CT}, \mathsf{fcl}(e))
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{cg}(\mathtt{CT}, \mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}) = {} & \mathtt{cg}(\mathtt{CT} \setminus C, D) \cup \mathtt{cg}(\mathtt{CT} \setminus C, \vec{M}) \\
& \cup [C \mapsto \mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}]
\end{aligned}
$$

Definition 4.13 (Complete class table). We say a class table $\mathtt{CT}$ is *complete with respect to* class C. if the following predicate holds:

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

We say a class table $\mathtt{CT}$ is *complete* if the following predicate holds:

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

Intuitively a class table $\mathtt{CT}$ is said to be complete if for every class $C \in \mathsf{dom}(\mathtt{CT})$, every superclass of C is also available in $\mathtt{CT}$. Completeness is essential for proper instantiation. Note that $\mathtt{cg}(\mathtt{CT}, C)$ generates the transitive closure of the superclasses of C. Lemma 8.4 shall show that if $\mathtt{CT}$ is typable, $\mathtt{cg}(\mathtt{CT}, C)$ returns a subset of $\mathtt{CT}$ which is complete together with $\mathtt{FCT}$.

Definition 4.14 (Substitutions). In the calculus there are three different kinds of substitution: those pertaining to the update of the store by assignments, those applied to the method call mechanism (such as substituting the actual receiver for the distinguished expression `this`), and those related to class labelling. We show each in turn.

For a store object $(C, \vec{f} : \vec{v})$, the substitution $[f \mapsto v]$ is defined as follows iff $f \in \vec{f}$:

$$(C, \vec{f}_1, f, \vec{f}_2 : \vec{v}_1, v, \vec{v}_2)[f \mapsto v'] = (C, \vec{f}_1, f, \vec{f}_2 : \vec{v}_1, v', \vec{v}_2)$$

Substitution of objects in the store is defined as follows:

$$\begin{aligned}
\emptyset[o \mapsto (C', \dots)] &= \emptyset \\
\sigma \cdot [x \mapsto v][o \mapsto (C', \dots)] &= \sigma[o \mapsto (C', \dots)] \cdot [x \mapsto v] \\
\sigma \cdot [o' \mapsto (C, \dots)][o \mapsto (C', \dots)] &= \sigma[o \mapsto (C', \dots)] \cdot [o' \mapsto (C, \dots)] \\
\sigma \cdot [o \mapsto (C, \dots)][o \mapsto (C', \dots)] &= \sigma \cdot [o \mapsto (C', \dots)]
\end{aligned}$$

For non-object store entries, substitutions are defined as follows:

$$\begin{aligned}
\emptyset[x \mapsto v] &= \emptyset \\
\sigma \cdot [x \mapsto v][x \mapsto v'] &= \sigma \cdot [x \mapsto v'] \\
\sigma \cdot [y \mapsto v][x \mapsto v'] &= \sigma[x \mapsto v'] \cdot [y \mapsto v] \\
\sigma \cdot [o \mapsto (C, \dots)][x \mapsto v'] &= \sigma[x \mapsto v'] \cdot [o \mapsto (C, \dots)]
\end{aligned}$$

We now consider the substitutions used in method invocation. For *receiver* substitution $e[o/\texttt{this}]$ is defined recursively over all sub-expressions of e. The only interesting base case is as follows:

$$\texttt{this}[o/\texttt{this}] = o$$

Similarly, *return* substitution $e[\texttt{return}(c)/\texttt{return}]$ is defined recursively over the sub-expressions of e with:

$$(\texttt{return}\ e)[\texttt{return}(c)/\texttt{return}] = \texttt{return}(c)\ (e[\texttt{return}(c)/\texttt{return}])$$

For example,

$$\begin{aligned}
&(\texttt{if}\ e\ \texttt{then}\ (e_1;\texttt{return}\ e_2)\ \texttt{else}\ (\texttt{return}\ e_3))[\texttt{return}(c)/\texttt{return}] \\
&\qquad \stackrel{\text{def}}{=} \texttt{if}\ e\ \texttt{then}\ (e_1;\texttt{return}(c)\ e_2)\ \texttt{else}\ (\texttt{return}(c)\ e_3)
\end{aligned}$$

Class labelling substitutions $[C^l/C]$ are defined as follows:

$$\begin{aligned}
\texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}[F^l/F] &= \texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f}; K\ (\vec{M}[F^l/F])\} \\
U\,\texttt{m}(\vec{T}\vec{x})\{e\}[F^l/F] &= U\ \texttt{m}(\vec{T}\vec{x})\{e[F^l/F]\}
\end{aligned}$$

As with the other substitutions, $e[F^l/F]$ is defined recursively over all sub-expressions of e. The only interesting case is:

$$\texttt{new}\ F(\vec{e})[F^l/F] = \texttt{new}\ F^l(\vec{e}') \quad \text{where } e'_i = e_i[F^l/F]$$

Configurations

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

Threads

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

Networks

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

Fig. 5.1. Structural equivalence

5 Operational Semantics

This section presents the operational semantics of DJ. It follows a standard small step call-by-value semantics [30, 4]. A large step semantics is not suitable due to our consideration of concurrent execution and possible interference between reductions. The structure of this section follows. We first introduce the structure rules in § 5.1; then we define the reduction rules for the standard expressions in § 5.4, serialisation/deserialisation in § 5.5, local/remote method invocation in § 5.6, freezing/defrosting in § 5.7, class downloading in § 5.8, and errors in § 5.10. Rules in all categories are mutually related. For reference, Appendix A lists all reduction rules.

5.1 Structural equivalences

This subsection defines the structural equivalences for DJ. They are defined for threads, networks and configurations in Fig. 5.1. This equivalence relation, which comes from π-calculus, handles scope of identifiers and parallel composition of threads and networks naturally by regarding two programs as identical. Formally, $\equiv$ is an equivalence relation which includes α-conversion and is generated by the equations in Fig. 5.1.

The last two rules for configurations define garbage collection of useless store entries, while the last three rules for threads are used to erase runtime value ε of the void type. Others rules, including scope opening, are inherited from those of the π-calculus [28], and so are standard.

5.2 Reduction

Reduction in DJ is expressed by two relations. The first is defined over configurations executing within an individual location, given by the binary relation $F \longrightarrow_l F'$ where l is the name of the location containing F. The second relation is global, defined over networks, and written $N \longrightarrow N'$. This relation includes the key distributed rules of DJ: (1) remote method invocation and (2) class downloading between locations.

We also define *multi-step* reduction as the union of the structural equivalence relation with the transitive closure of the $\longrightarrow$ relation as $\twoheadrightarrow \stackrel{\text{def}}{=} (\longrightarrow \cup \equiv)^*$ and $\twoheadrightarrow_l \stackrel{\text{def}}{=} (\longrightarrow_l \cup \equiv)^*$.

5.3 Evaluation contexts

To reduce the number of computation rules, we make use of the evaluation contexts in Fig. 5.2.

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. Evaluation order of terms in the language is determined by the construction of these contexts.

$$
\begin{aligned}
E ::=\ & [\,] \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid E.\texttt{f} \mid E;e \mid T\ x = E \mid x := E \mid E.\texttt{f} := e \mid o.\texttt{f} := E \\
\mid\ & \texttt{new}\ C(\vec{v},E,\vec{e}) \mid E.\texttt{m}(\vec{e}) \mid o.\texttt{m}(\vec{v},E,\vec{e}) \mid \texttt{defrost}(E) \mid \texttt{return}(c)\ E \\
\mid\ & \texttt{go}\ E\ \texttt{with}\ c \mid E\ \texttt{with}\ c \mid \texttt{go}\ E\ \texttt{to}\ c
\end{aligned}
$$

Fig. 5.2. Evaluation contexts

5.4 Standard expressions

In Fig. 5.3 we outline the basic reduction rules for the language. These rules form the sequential part of the language, and do not mention concurrency or channel-based communication. Most rules are standard [18, 4, 9], except the following three points:

fresh identifier creation When allocating new space on the store by **RC-Dec** and **RC-New**, we explicitly restrict identifiers. This operation represents "the freshness" or "uniqueness" of the address of the new entries. This facility is important for a natural formulation of the distributed reduction relations as well as invariants related to the locality of identifiers (cf. Section 9).

tagged class creation The special allocation rule, **RC-NewR**, is applied whenever execution attempts to instantiate an object of a tagged class. Instead of immediately allocating a new object, this rule first attempts to download the actual body of the class from the labelled location (the reduction rules for class downloading are discussed in §. 5.8).

use of reduction context When expressions are evaluated in contexts, they may create new identifiers (by **RC-Dec** and **RC-New**, for example). The scope of these new identifiers must be opened over the whole of the context, as shown:

$$(E[\texttt{new}\ C(\vec{v})], \sigma, \texttt{CT}) \longrightarrow_l (\nu\, o)(E[o], \sigma, \texttt{CT})$$

This allows newly created identifiers to be successfully propagated to where they must be used. For example in the case above, if $E \equiv [\,].\texttt{m}()$, then without automatic scope opening the method call $o.\texttt{m}()$ could never be evaluated. The restriction of **RC-Seq** is similarly explained.

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

RC-Cond
$$\texttt{if true then}\ e_1\ \texttt{else}\ e_2, \sigma, \texttt{CT} \longrightarrow_l e_1, \sigma, \texttt{CT}$$
$$\texttt{if false then}\ e_1\ \texttt{else}\ e_2, \sigma, \texttt{CT} \longrightarrow_l e_2, \sigma, \texttt{CT}$$

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

RC-Seq
$$\frac{e_1, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(v, \sigma', \texttt{CT}')}{e_1; e_2, \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{u})(e_2, \sigma', \texttt{CT}')} \quad \vec{u} \notin \mathsf{fnv}(e_2)$$

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

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

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

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

RC-NewR
$$\texttt{new}\ C^m(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l \texttt{download}\ C\ \texttt{from}\ m\ \texttt{in new}\ C(\vec{v}), \sigma, \texttt{CT} \quad C \notin \mathsf{dom}(\texttt{CT})$$

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

Fig. 5.3. Rules for local expressions

5.5 Serialisation and deserialisation

One of the contributions of DJ is a precise formalisation of the semantics of serialisation. This subsection outlines our interpretation of the Java *serialisation* API in the

Serialize

$$\frac{\sigma' = \bigcup \mathsf{og}(\sigma, v_i) \qquad \{\vec{o}\} = \mathsf{dom_o}(\sigma')}{\texttt{serialize}(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l \lambda\vec{o}.(\vec{v}, \sigma', l), \sigma, \texttt{CT}}$$

Deserialize

$$\frac{\{\vec{F}\} = \{C \mid \sigma'(o_i) = (C, \ldots)\} \qquad \vec{o} \notin \mathsf{dom}(\sigma)}{\texttt{deserialize}(\lambda\vec{o}.(\vec{v}, \sigma', m)), \sigma, \texttt{CT} \longrightarrow_l (\nu\,\vec{o})(\texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in}\ \vec{v}, \sigma \cup \sigma', \texttt{CT})}$$

Fig. 5.4. Rules for serialisation and deserialisation

form of operational semantics. Serialisation occurs in two situations. In the first situation, the expressions $\texttt{serialize}(e)$ and $\texttt{deserialize}(e)$ allow explicit flattening and re-inflation of objects by the programmer.

The second instance occurs automatically when values must be transported across the network. Instances of local classes are incapable of remote method invocation, and so we cannot pass them by reference as parameters or as return values to remote method invocations. Should this occur, the remote party would receive the identifier of an unreachable object. Avoiding this problem involves sending local objects to and from remote methods *by value*, i.e. in serialised form. Therefore $\texttt{serialize}(e)$ and $\texttt{deserialize}(e)$ must appear automatically as runtime expressions, to serialise parameters and return values of remote method invocations.

The reduction rules for (de)serialisation appear in Fig. 5.4. For primitive values such as integers, producing a serialised representation is straightforward as they have no significant internal structure. However objects may refer to others in the store, and so care must be taken when serialising them.

For *serialisation* we apply **Serialize**. For remote references or primitives, we assume these values are already of suitable form and so serialisation makes no change. However if a local object reference o is passed to a remote method as part of the parameters $\vec{v}$, it must be sent with all its dependent objects. This means taking a copy of every *local* object referenced by o either directly or transitively. For this purpose we apply the object graph computation algorithm given in Definition 4.8 to obtain $\sigma' = \bigcup \mathsf{og}(\sigma, v_i)$. Once the graph σ' has been constructed, we are able to build a blob of the form $\lambda\vec{o}.(\vec{v}, \sigma', l)$. All parameters must be serialised at the same time, since referential integrity must be maintained. For instance, in the following code (cf. [37]): `x.f = y; r.m(x,y);`, the remote method r is called with two local object parameters x, y. If we serialise each parameter individually then the relationship between the two values (via the field f) is lost. In this situation two copies of y are created at the remote site, violating referential integrity.

The *deserialisation* operation is the dual to serialisation. We deserialise a blob by applying **Deserialize**. Remote object identifiers and base values are invariant under this operation, but local objects contained in the blob must be treated with care. For each $o \in \vec{o}$ we create a new local identifier, renaming all occurrences of the reference within the object graph to this new identifier, preserving the graph structure. After completing

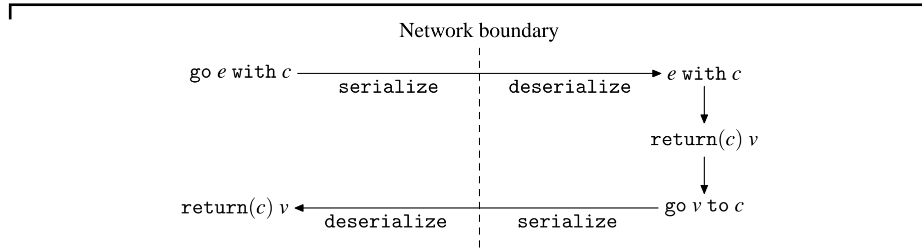


Fig. 5.5. Remote method invocation

this process, we append the resulting data to the local store ("$\sigma \cup \sigma'$"). The only complication arises when σ' contains instances of classes that are not in the class table of the deserialising location. These must be retrieved from the serialising location m by making a call to download vector $\vec{F}$. This accurately mimics the mechanism employed by the `RMIClassLoader` class used in Java RMI. When sending marshaled objects, RMI implementations annotate the data stream for classes with a codebase URL. This is a pointer to a remote directory that the `RMIClassLoader` can refer to download classes that are not available at the current location.

5.6 Method invocation

A key omission from Fig. 5.3 is the rules for *method invocation*. Unlike sequential formalisms, DJ describes *remote method invocation*. To accommodate RMI, the rules for method call take a novel form employing concepts from the π-calculus, representing the context of a call by a local linear channel. While this technique is well-known in the π-calculus [27], 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. The general picture of a remote method invocation is summarised in Fig. 5.5, which starts from dispatch of a remote method and ends with delivery of its return value. The corresponding formal rules are given in Fig. 5.6.

We first start from local method calls. For a method call $o.\mathtt{m}(\vec{v})$, if $o \in \mathsf{dom}_{\mathsf{o}}(\sigma)$ (where σ is the local heap) then the rule **RC-MethLocal** is applied. This indicates a local method invocation. There are three operations defined in this rule:

1. a new channel c is created to carry the return value of the method;
2. the return point of the method call is replaced with the term `await` c which can be thought of as a receiver waiting for some value to be supplied on channel c;
3. the method call itself is spawned in a new thread and rewritten to $o.\mathtt{m}(\vec{v})$ `with` c which should be read as "the method call $o.\mathtt{m}(\vec{v})$ will return its value to channel c".

RC-MethLocal
$E[o.\mathtt{m}(\vec{v})] \,|\, P, \sigma, \mathtt{CT} \longrightarrow_l (\nu c)(E[\mathtt{await}\ c] \,|\, o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, P, \sigma, \mathtt{CT}) \quad c \text{ fresh}, o \in \mathsf{dom_o}(\sigma)$

RC-MethRemote
$E[o.\mathtt{m}(\vec{v})] \,|\, P, \sigma, \mathtt{CT} \longrightarrow_l (\nu c)(E[\mathtt{await}\ c] \,|\, \mathsf{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c \,|\, P, \sigma, \mathtt{CT})$
$c \text{ fresh}, o \notin \mathsf{dom_o}(\sigma)$

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

RC-Await
$E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v, \sigma, \mathtt{CT} \longrightarrow_l E[v], \sigma, \mathtt{CT}$

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

RN-Leave
$l_1[\mathsf{go}\ o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, P_1, \sigma_1, \mathtt{CT}_1] \,|\, l_2[P_2, \sigma_2, \mathtt{CT}_2]$
$\longrightarrow l_1[P_1, \sigma_1, \mathtt{CT}_1] \,|\, l_2[o.\mathtt{m}(\mathtt{deserialize}(\vec{v}))\ \mathtt{with}\ c \,|\, P_2, \sigma_2, \mathtt{CT}_2] \qquad o \in \mathsf{dom_o}(\sigma_2)$

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

Fig. 5.6. Rules for method invocation

The next stage of execution is the application of **RC-MethInvoke**, which is recognisable as in the style of a traditional method invocation rule. Both remote and local method invocations ultimately apply this rule; it contains the common parts required for method invocation, such as assigning store space for formal parameters, setting the receiver and gathering the return value. There are three operations defined in this rule:

1. the receiver parameter is substituted $[o/\mathtt{this}]$ at invocation time;
2. new store entries for the formal parameters $\vec{x}$ are created and initialised to the supplied parameters $\vec{v}$ as similar with **RC-Dec** and **RC-New**.
3. the value returned by the method must be sent along channel c, which is realised by the substitution $[\mathtt{return}(c)/\mathtt{return}]$.

The most interesting aspect of method call in DJ is the use of channels as described in the final step above. This leads to the definition of the method return rule, **RC-Await**, which is also common to remote and local calls. This communicates the return value to its caller.

For the call shown above, if it is the case that $o \notin \mathsf{dom_o}(\sigma)$ then we have a *remote method invocation*. In this case the rule **RC-MethRemote** is applied. This rule shares some similarities with **RC-MethLocal**: it creates a new channel and return point, and it spawns a new thread to carry the method's code. However it differs because we can

no longer assume that any object identifiers in the parameter vector $\vec{v}$ can be passed by reference. For example:

$$o.\mathtt{m}(o')$$

Here, if the object identifier o' does not refer to an instance of a remote class then it will be essentially meaningless to a remote server. To cope with this, Java RMI makes use of the serialisation API, and in DJ we employ a similar mechanism. In a nutshell, when the name of a local object is passed to a remote method as a parameter the entire object graph it represents must be copied. This means any other local objects referred as fields must be taken along with the original object to ensure that the remote method can access any information it needs. This has been discussed in § 5.5. We note that the thunked values are also transferred to the remote location without modification (like base values).

After this serialisation has taken place, we are left with a thread of the form $\mathtt{go}\ o.\mathtt{m}(\vec{w})\ \mathtt{with}\ c$ where $\vec{w}$ is the serialised representation of the original parameters $\vec{v}$. At this point, the network level rule **RN-Leave** triggers the migration of the method calling thread to the location that holds the receiving object in its local store. After transfer over the network, any parameters that were previously serialised must be deserialised and then **RC-MethInvoke** is applied.

The return value of a remote method must be serialised using **RN-SerReturn**, after which it crosses the network by application of **RN-Return**. After returning to the calling site, it is again deserialised using the same mechanism. Fig. 5.5 summarises how values are returned from methods by channel communication.

5.7 Direct code mobility

Before moving to class downloading, we first illustrate the semantics of thunk creation/resolution. Thunks offer a direct way to manipulate code (classes).

As noted, there are two operations associated with thunks, one for the creation of thunks and the other for their later use, called *freezing* and *defrosting*. Their rules are given in Fig. 5.7. A notable point is a use of the invariants to design the reduction rule for freezing, as we illustrate below.

RC-Freeze

$$\frac{\{\vec{x}\} = \mathsf{fv}(e) \quad v_i = \sigma(x_i) \qquad \mathtt{CT}' = \begin{cases} \mathsf{cg}(\mathtt{CT}, \mathsf{fcl}(e)) & t = \mathtt{eager} \\ \emptyset & t = \mathtt{lazy} \end{cases}}{\mathtt{freeze}[t](e), \sigma, \mathtt{CT} \longrightarrow_l \ulcorner e[\vec{v}/\vec{x}]\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ l \urcorner, \sigma, \mathtt{CT}}$$

RC-Defrost

$$\frac{\{\vec{C}\} = \mathsf{fcl}(e) \setminus \mathsf{dom}(\mathtt{CT}')}{\mathtt{defrost}(\ulcorner e\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ m \urcorner), \sigma, \mathtt{CT} \longrightarrow_l e[\vec{C^m}/\vec{C}], \sigma, \mathtt{CT} \cup \mathtt{CT}'}$$

Fig. 5.7. Rules for creating and executing thunked expressions

Freezing defined by **RC-Freeze** takes two modes, `lazy` or `eager`, and its operation is divided into two steps.

1. we instantiate values $\vec{v}$ stored in the local memory σ to the free local variables $\mathsf{fv}(e)$ occurring in e creating $e[\vec{v}/\vec{x}]$.
2. when the mode is `eager`, we calculate the class table using the class graph function $\mathtt{cg}(\mathtt{CT}, \mathsf{fcl}(e))$ defined in Definition 4.12, and attach together with e.

The initial step (1) above is the same for the both modes. We take special care of local variable references made in e. For example, it would be dangerous to freeze an expression such as $o.\mathtt{m}(x)$ because at the eventual point of use, variable x will no longer be accessible. To avoid this, while still allowing some use of local variables in frozen expressions, the freeze operation takes the step 1 above. In the above example, if we suppose that at the point of evaluation of $\mathtt{freeze}[t](o.\mathtt{m}(x))$, variable x was mapped to a value `true` in the store, then the freezing operation can safely replace x with its actual value yielding $\ulcorner o.\mathtt{m}(\mathtt{true})\ \mathtt{with}\ \mathtt{CT}\ \mathtt{from}\ l\urcorner$ which does not contain the obvious error. Assuming the invariant which states that all values stored are closed (which will be stated as $\mathsf{Inv}(5)$ in Definition 9.2 later), we can ensure the newly created thunk is closed again.

Concerning the mode of freezing, this merely determines the amount of information sent in the class table fragment that accompanies the expression in the thunk. In `lazy` mode, this is the empty class table (hence $\ulcorner e\ \mathtt{with}\ \emptyset\ \mathtt{from}\ l\urcorner$)—lazy means that it is the client's responsibility to download the classes if it needs them.

In the `eager` mode, the freeze operation calculates the class dependency graph required for the expression to evaluate using the class graph function defined in Definition 4.12. The eager approach is conservative, computing all the classes that *may* be used (including all superclasses) rather than those that necessarily *are*. For example, the following expression:

$$\mathtt{if\ false\ then\ new}\ C(v)\ \mathtt{else\ new}\ D(v) \quad \text{with } C <: E \text{ and } D <: F$$

actually needs only class D and (maybe) its superclass F to execute safely, but the class graph function includes $\{C, D, E, F\}$.

The rule **RC-Defrost** is used to defrost a thunk $\ulcorner e\ \mathtt{with}\ \mathtt{CT}\ \mathtt{from}\ l\urcorner$. The first step is to augment the class table of the current location with that provided by the thunk. All classes mentioned in e are tagged with their originating location: $\mathtt{new}\ C(\vec{v})$ becomes $\mathtt{new}\ C^l(\vec{v})$. During execution of the newly defrosted thunk, if an expression such as the above $\mathtt{new}\ C^l(\vec{v})$ is encountered then **RC-NewR** is applied as explained earlier. It is possible that C may not have been downloaded to the execution location (since it is up to the party who froze the thunk on the contents of `CT`). The alternative rule for new object creation downloads these class table entries only if necessary, giving lazy semantics. Note that in the eager case, all required classes are attached together as $\mathtt{CT}'$, hence this substitution is unnecessary (i.e. $\vec{C} = \emptyset$). On the other hand, in the lazy case, $\mathtt{CT}' = \emptyset$ always. Note also that a thunk would be shipped to the remote location as the argument of RMI, and be defrost in that location.

RC-Resolve

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

RN-Download

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

RC-DownloadNothing

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

Fig. 5.8. Rules for class downloading

Remark 5.1. Instead of having only two modes, we can generalise the freeze operator to ship the user-defined classes as follows.

RC-FreezeC

$$\frac{\{\vec{x}\} = \mathsf{fv}(e) \quad v_i = \sigma(x_i) \qquad \mathtt{CT}' = \mathtt{cg}(\mathtt{CT},\vec{C})}{\mathtt{freeze}[\vec{C}](e),\sigma,\mathtt{CT} \longrightarrow_l \ulcorner e[\vec{v}/\vec{x}]\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ l\urcorner,\sigma,\mathtt{CT}}$$

This requires no change to the rule for defrosting. To uniformly prove the correctness of all three choices, we use the invariant properties introduced in Section 9. Specifically we use an invariant to guarantee the completeness of class table CT.

5.8 Class downloading

The formalisation of class downloading is one of the key contributions of DJ. Class mobility is very important in Java RMI systems, since it reduces the 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.

The rules for class downloading in DJ are given in Fig. 5.8. The *download* expression is responsible for the transfer of class table entries from a remote site. The semantics of the operation are given in **RN-Download**, informally $\mathtt{download}\ \vec{C}\ \mathtt{from}\ l\ \mathtt{in}\ e$ attempts to download the vector of classes $\vec{C}$ from location l for use in expression e. Resolution defined by **RC-Resolve** is the process of examining classes for unmet dependencies and scheduling the download of missing classes. The two rules work together to iteratively resolve all class dependencies for a given object. Once all dependencies have been met, normal execution continues after **RC-DownloadNothing**.

For a request to download classes $\vec{C}$ by `download` $\vec{C}$ `from` l_2 `in` e there are three actions:

1. $\vec{D}$ is calculated. It comprises the classes in $\vec{C}$ less any already available in the local class table (preventing duplicate downloads);
2. we then compute vector $\vec{F}$. This comprises the free class names contained in all method bodies of classes in $\vec{D}$;
3. finally, class table $\mathtt{CT}'$ is copied to the local class table. $\mathtt{CT}'$ comprises the bodies corresponding to the class names in $\vec{D}$, with any occurrence of a member of $\vec{F}$ tagged with the name of the remote site (l_2 in this case).

These rules represent the lazy downloading mechanism, as is standard in Java (they approximately model the strategy in JDK 1.3 without verification [10]).

Remark 5.2. Eager class downloading, which is effective in a high bandwidth environment, is easily defined by changing $\mathtt{CT}_2(\vec{D})[\vec{F}^{l_2}/\vec{F}]$ in the premise of **RN-Download** into $\mathtt{cg}(\mathtt{CT}_2,\vec{D})$. The correctness is easily proved by the completeness of class table $\mathtt{CT}$ (in particular, this case is a special case where a chain of download-resolve reductions does not exist, cf. Lemma 9.4(3,4)).

5.9 Threads and networks

In Fig. 5.9 we give the standard reduction rules for executing threads and networks in parallel, closed under restricted names and also the execution of configurations within locations. **RN-Conf** promotes the local reduction $\longrightarrow_l$ to the network level $\longrightarrow$. These contextual rules are standard, see [28].

RC-Par

$$\frac{P_1,\sigma,\mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1',\sigma',\mathtt{CT}')}{P_1\,|\,P_2,\sigma,\mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(P_1'\,|\,P_2,\sigma',\mathtt{CT}')}\quad \vec{u}\notin \mathsf{fnv}(P_2)$$

RC-Str

$$\frac{F \equiv F_0 \longrightarrow_l F_0' \equiv F'}{F \longrightarrow_l F'}$$

RC-Res

$$\frac{(\nu\,\vec{u})(P,\sigma,\mathtt{CT}) \longrightarrow_l (\nu\,\vec{u}')(P',\sigma',\mathtt{CT}')}{(\nu\,u\vec{u})(P,\sigma,\mathtt{CT}) \longrightarrow_l (\nu\,u\vec{u}')(P',\sigma',\mathtt{CT}')}$$

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

$$\frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

Fig. 5.9. Rules for network and thread

5.10 Errors

The final subsection defines two distinct classes of error: those arising from a programmer mistake and those arising from some unforeseen system failure. In these rules, `Error` represents the state of execution reached when a program goes awry in some way.

The key programming error that may arise is attempting to dereference a null pointer. This gives rise to the `NullPointerException` in Java. The reduction rules concerning these errors are given in Fig. 5.10. **Err-NullFld** and **Err-NullFldAss** characterise the error when field access is attempted on a null reference, while **Err-NullMeth** does when a method invocation is attempted on a null reference. It is important to note that in this situation the parameters of the method invocation are fully computed (even though it is possible to determine a `null` receiver before then). These formulations are consistent with the current Java specification.

Err-NullFld
$\mathtt{null.f}, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$

Err-NullFldAss
$\mathtt{null.f} := v, \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$

Err-NullMeth
$\mathtt{null.m}(\vec{v}), \sigma, \mathtt{CT} \longrightarrow_l \mathtt{Error}, \sigma, \mathtt{CT}$

Fig. 5.10. Programmer errors

Err-Download

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

Err-MLossLeave

$$l_1[\mathtt{go}\ o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, P_1, \sigma_1, \mathtt{CT}_1] \,|\, l_2[P_2, \sigma_2, \mathtt{CT}_2] \longrightarrow l_1[\mathtt{Error} \,|\, P_1, \sigma_1, \mathtt{CT}_1] \,|\, l_2[P_2, \sigma_2, \mathtt{CT}_2] \quad o \in \mathsf{dom_o}(\sigma_2)$$

Err-MLossReturn

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

Fig. 5.11. Network failure errors

The second class of error is a natural consequence of a distributed system - network failure. Fig. 5.11 outlines three situations when such a failure might exhibit an effect

on a network. The rule **Err-Download** this characterises the corruption of class table data as it is transmitted over the network. In the case of **Err-MLossLeave**, the network has become partitioned such that a remote method call attempting to reach location l_2 cannot. This results in the calling code reducing to `Error`. Likewise, in the case of **Err-MLossReturn**, the return value from a remote method call cannot return to the original caller located at l_2. It too reduces to `Error`.

6 Examples of operational semantics

This section shows examples of operational semantics focussing on RMI with code mobility and serialisation. The first subsection presents eager and lazy code mobility; the third one demonstrates code mobility across remote sites; and the final one shows serialisation. The first example in § 6.1 corresponds to the second program in Section 2, while the last example in § 6.3 to the first one in in Section 2. We slightly modify the programs in Section 2 to demonstrate a non-trivial interplay between RMI, code mobility, class downloading and serialisation.

6.1 Code mobility

Eager code mobility We write *cl* for the mobile phone ("client") site, and *sv* for the server. Each site maintains a class table of well-known classes and the specialist classes in Listing 6.1 and Listing 6.2 respectively. For simplicity we assume that the library function `mod` is available universally. The computation in question is the simple calculation of the greatest common divisor of two numbers. In the client code the class *Task* contains the body of Euclid's algorithm. As can be seen from Listing 6.1, *Client* makes a thunk of the expression `new Task().gcd(a,b)`, and, depending on the choice of tag t, may enclose a copy of the *Task* class so that execution can begin immediately. In this case, the defrosting operation at the remote location *sv* will append the definition of class *Task* to the class table of *sv*. It then instantiates *Task* and calls the method *gcd*. The parameters a and b are replaced with their actual values by *cl* at the time of freezing. Assume $\sigma_{sv} = [s \mapsto (Server, \dots)]$, and that the client code at location *cl* has a reference to the remote object s. We write the network as follows:

$$(\nu\, s)(cl[\texttt{return}(c)\ \texttt{new}\ Client(s).\texttt{gcd}(1071, 1029), \sigma_{cl}, \texttt{CT}_{cl}]\,|\,sv[\mathbf{0}, \sigma_{sv}, \texttt{CT}_{sv}])$$

We consider location *cl* initially in isolation, until the remote method call $s.\texttt{cpu}(\dots)$.

$$\begin{array}{ll} & \texttt{return}(c)\ \texttt{new}\ Client(s).\texttt{gcd}(1071, 1029), \sigma_{cl}, \texttt{CT}_{cl} \\ \longrightarrow_{cl} & (\nu\, o)(\texttt{return}(c)\ o.\texttt{gcd}(1071, 1029), \sigma'_{cl}, \texttt{CT}_{cl}) \quad (\sigma'_{cl} = \sigma_{cl} \cdot [o \mapsto (Client, s : s)]) \\ \longrightarrow_{cl} & (\nu\, od)(\texttt{return}(c)\ \texttt{await}\ d\,|\,o.\texttt{gcd}(1071, 1029)\ \texttt{with}\ d, \sigma'_{cl}, \texttt{CT}_{cl}) \end{array}$$

Now let $\sigma''_{cl} = \sigma'_{cl} \cdot [a, b \mapsto 1071, 1029]$ and $e \equiv \texttt{new}\ Task().\texttt{gcd}(a, b)$. Then the last line reduces to:

$$(\nu\, odab)(\texttt{return}(c)\ \texttt{await}\ d\,|\,\texttt{thunk}(int)\ g = \texttt{freeze}[t](e)\ ;\ \dots, \sigma''_{cl}, \texttt{CT}_{cl})$$

```
1  /** Local class */
2  class Client extends Object {
3    Server s;
4    Client(Server s) { this.s = s; }
5    int gcd(int a, int b) {
6      thunk(int) g = freeze[t](new Task().gcd(a,b));
7      return s.cpu(g);
8    }
9  }
10 /** Local class */
11 class Task extends Object {
12   int gcd(int a, int b) {
13     if (b == 0) {
14       return a;
15     } else {
16       return this.gcd(b,mod(a,b));
17     }
18   }
19 }
```

Listing 6.1. Class table $\mathtt{CT}_{cl}$

Let $t = \mathtt{eager}$, therefore $\mathtt{CT}' = [Task \mapsto \ldots]$ and $e' \equiv \mathtt{new}\ Task().\mathtt{gcd}(1071, 1029)$. Then $\mathtt{freeze}[t](e)$ reduces to $\ulcorner e'\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ cl \urcorner$. Hence it reduces to

$$
\begin{aligned}
& (\nu\, odab)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{thunk}(int)\ g = \ulcorner e'\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ cl \urcorner\,;\ \ldots, \sigma''_{cl}, \mathtt{CT}_{cl}) \\
\equiv\ & (\nu\, d)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{thunk}(int)\ g = \ulcorner e'\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ cl \urcorner\,;\ \ldots, \sigma_{cl}, \mathtt{CT}_{cl}) \quad (6.0.1)
\end{aligned}
$$

Let $\sigma'''_{cl} = \sigma_{cl} \cdot [g \mapsto v]$ where $v = \ulcorner e'\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ cl \urcorner$. Then (6.0.1) reduces to:

$$
\begin{aligned}
& (\nu\, dg)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{return}(d)\ o.\mathtt{s.cpu}(g), \sigma'''_{cl}, \mathtt{CT}_{cl}) \\
\longrightarrow_{cl}\ & (\nu\, dg)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{return}(d)\ s.\mathtt{cpu}(g), \sigma'''_{cl}, \mathtt{CT}_{cl}) \\
\longrightarrow_{cl}\ & (\nu\, dg)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{return}(d)\ s.\mathtt{cpu}(v), \sigma'''_{cl}, \mathtt{CT}_{cl}) \\
\equiv\ & (\nu\, d)(\mathtt{return}(c)\ \mathtt{await}\ d \,|\, \mathtt{return}(d)\ s.\mathtt{cpu}(v), \sigma_{cl}, \mathtt{CT}_{cl}) \quad (6.0.2)
\end{aligned}
$$

```
1  /** Remote class */
2  class Server {
3    int cpu(thunk(int) t) {
4      return defrost(t);
5    }
6  }
```

Listing 6.2. Class table $\mathtt{CT}_{sv}$

Since $s \notin \mathsf{dom}_{\mathsf{o}}(\sigma_{cl})$ we have a remote method call. Hence from (6.0.2), we have:

$$\begin{aligned}&(\nu\, df)(\texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f \,|\, \texttt{go}\ s.\texttt{cpu}(\texttt{serialize}(v))\ \texttt{with}\ f, \sigma_{cl}, \texttt{CT}_{cl})\\ \longrightarrow_{cl}\ &(\nu\, df)(\texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f \,|\, \texttt{go}\ s.\texttt{cpu}(v)\ \texttt{with}\ f, \sigma_{cl}, \texttt{CT}_{cl})\end{aligned} \tag{6.0.3}$$

At (6.0.3) the method is ready to move to location sv by rule **RN-Leave**. Before this can happen, the scope of the restricted names must be opened over the network using $\equiv$. After simplifying the network can be written:

$$\begin{aligned}(\nu\, csdf)(cl[\texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f\ &|\, \texttt{go}\ s.\texttt{cpu}(v)\ \texttt{with}\ f, \sigma_{cl}, \texttt{CT}_{cl}]\\ &|\, sv[\mathbf{0}, \sigma_{sv}, \texttt{CT}_{sv}])\end{aligned}$$

After movement this becomes:

$$\begin{aligned}(\nu\, csdf)(cl[\texttt{return}(c)\ \texttt{await}\ d\ &|\, \texttt{return}(d)\ \texttt{await}\ f, \sigma_{cl}, \texttt{CT}_{cl}]\\ &|\, sv[s.\texttt{cpu}(\texttt{deserialize}(v))\ \texttt{with}\ f, \sigma_{sv}, \texttt{CT}_{sv}])\end{aligned}$$

Now consideration turns to location sv in isolation. Then from the previous line, we have:

$$\begin{aligned}&s.\texttt{cpu}(v)\ \texttt{with}\ f, \sigma_{sv}, \texttt{CT}_{sv}\\ \longrightarrow_{sv}\ &(\nu\, t)(\texttt{return}(f)\ \texttt{defrost}(t), \sigma'_{sv}, \texttt{CT}_{sv}) \quad (\sigma'_{sv} = \sigma_{sv} \cdot [t \mapsto v])\\ \longrightarrow_{sv}\ &(\nu\, t)(\texttt{return}(f)\ \texttt{defrost}(v), \sigma'_{sv}, \texttt{CT}_{sv})\\ \equiv\ &\texttt{return}(f)\ \texttt{defrost}(v), \sigma_{sv}, \texttt{CT}_{sv}\\ \longrightarrow_{sv}\ &\texttt{return}(f)\ \texttt{new}\ \mathit{Task}().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'\\ \longrightarrow_{sv}\ &(\nu\, o)(\texttt{return}(f)\ o.\texttt{gcd}(1071, 1029), \sigma''_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}') \quad \sigma''_{sv} = \sigma_{sv} \cdot [o' \mapsto (\mathit{Task}, \varepsilon)]\end{aligned} \tag{6.0.4}$$

$$\longrightarrow_{sv}\ (\nu\, oh)(\texttt{return}(f)\ \texttt{await}\ h \,|\, o.\texttt{gcd}(1071, 1029)\ \texttt{with}\ h, \sigma''_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}') \tag{6.0.5}$$

Assuming the calculation goes well, then the greatest common divisor of 1071 and 1029 is 21. Each recursive call generates a new pair of entries on the stack for the parameters a and b. We denote these by the vector $\vec{u}$. It is interesting to note that each "stack frame" in the recursive call corresponds to a thread of the form $\texttt{return}(x)\ \texttt{await}\ y$ for some channels x and y. We resume execution at the point where the final stack frame is removed and the result is returned. We write σ'''_{sv} to represent the store after execution. Then from (6.0.5), we have:

$$\begin{aligned}&(\nu\, oh\vec{u})(\texttt{return}(f)\ \texttt{await}\ h \,|\, \texttt{return}(h)\ 21, \sigma'''_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}')\\ \equiv\ &(\nu\, h)(\texttt{return}(f)\ \texttt{await}\ h \,|\, \texttt{return}(h)\ 21, \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}')\\ \longrightarrow_{sv}\ &(\nu\, h)(\texttt{return}(f)\ 21, \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}')\\ \equiv\ &\texttt{return}(f)\ 21, \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'\\ \longrightarrow_{sv}\ &\texttt{go}\ \texttt{serialize}(21)\ \texttt{to}\ f, \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'\\ \longrightarrow_{sv}\ &\texttt{go}\ 21\ \texttt{to}\ f, \sigma_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'\end{aligned} \tag{6.0.6}$$

Opening the scope of the restricted names, and applying garbage collection to $\vec{u}, t, o$, rule **RN-Return** is used to give the following network (written in normal form):

$$(\nu\, csdf)(cl[\texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f, \sigma_{cl}, \texttt{CT}_{cl}] \,|\, sv[\texttt{go}\ 21\ \texttt{to}\ f, \sigma'''_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'])$$

After movement this becomes:

$$\begin{aligned}(\nu\, csdf)(cl[\ & \texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f \\ & |\, \texttt{return}(f)\ \texttt{deserialize}(21), \sigma_{cl}, \texttt{CT}_{cl}] \\ & |\, sv[\mathbf{0}, \sigma'''_{sv}, \texttt{CT}_{sv} \cup \texttt{CT}'])\end{aligned}$$

Again concentrating on location cl:

$$\begin{aligned} & \texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f \,|\, \texttt{return}(f)\ \texttt{deserialize}(21), \sigma_{cl}, \texttt{CT}_{cl} \\ \longrightarrow_{cl}\ & \texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ \texttt{await}\ f \,|\, \texttt{return}(f)\ 21, \sigma_{cl}, \texttt{CT}_{cl} \\ \longrightarrow_{cl}\ & \texttt{return}(c)\ \texttt{await}\ d \,|\, \texttt{return}(d)\ 21, \sigma_{cl}, \texttt{CT}_{cl} \\ \longrightarrow_{cl}\ & \texttt{return}(c)\ 21, \sigma_{cl}, \texttt{CT}_{cl} \end{aligned} \tag{6.0.7}$$

Finally, we assume there is a channel c on which another process is waiting for input. This is used to model the return value of the entire program.

Lazy code mobility At (6.0.1), we chose to assume that the tag t was set to `eager` mode. In this mode, every class potentially required to evaluate the thunk at a remote site is bundled with it. In this case, the only class needed was *Task*. However, suppose we had chosen $t = \texttt{lazy}$, then we would have that $\texttt{CT}' = \emptyset$.

To illustrate lazy downloading, we shall assume that $Task \notin \mathsf{dom}(\texttt{CT}_{sv})$. So to execute the code e', location sv must obtain the classes it needs. At the defrost step, we have:

$$\begin{aligned} & \texttt{return}(f)\ \texttt{defrost}(v), \sigma_{sv}, \texttt{CT}_{sv} \\ \longrightarrow_{sv}\ & \texttt{return}(f)\ \texttt{new}\ Task^{cl}().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \end{aligned}$$

Since $Task \notin \mathsf{dom}(\texttt{CT}_{sv})$, the class name *Task* has been tagged with the location that created the thunk as specified by **RC-Defrost**. Then instead of normal instantiation by **RC-New**, we must download *Task* from cl using **RC-NewR**. Hence the previous line reduces to:

$$\texttt{return}(f)\ \texttt{download}\ Task\ \texttt{from}\ cl\ \texttt{in}\ \texttt{new}\ Task().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \tag{6.0.8}$$

We now apply **RN-Download** to retrieve the definition of *Task*:

$$\begin{aligned} & \texttt{return}(f)\ \texttt{resolve}\ Task\ \texttt{from}\ cl\ \texttt{in}\ \texttt{new}\ Task().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \cdot [Task \mapsto \ldots] \\ \longrightarrow_{sv}\ & \texttt{return}(f)\ \texttt{download}\ Object\ \texttt{from}\ cl\ \texttt{in}\ \texttt{new}\ Task().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \cdot [Task \mapsto \ldots] \end{aligned} \tag{6.0.9}$$

$Object \in \mathsf{dom}(\texttt{CT}_{sv})$ because it is a foundation class, so **RC-DownloadNothing** is applied:

$$\texttt{return}(f)\ \texttt{new}\ Task().\texttt{gcd}(1071, 1029), \sigma_{sv}, \texttt{CT}_{sv} \cdot [Task \mapsto \ldots] \tag{6.0.10}$$

Now all necessary classes are downloaded, execution can resume as before from (6.0.4).

6.2 Nested code mobility

This example shows the use of thunked process to compute a value incrementally in several stages, moving among three different sites. Consider the class table in Listing 6.3. Suppose there are two remote object identifiers (o and p), both instances of class B, in scope of the program:

```
new A().compute(2,3,o,p);
```

After execution, this method should return the value 24. The actual computation approach makes use of the ability to arbitrarily nest $\texttt{freeze}[t](e)$ expressions to perform the calculation in two steps at two different locations.

```
1  class A {
2    int compute(int a, int b, B o, B p) {
3      return o.execute(freeze[eager](
4      int x = a + 2;
5      int y = b + 3;
6      return p.execute(freeze[eager](x * y))
7      ));
8    }
9  }
10 class B {
11   int execute(thunk(int) t) {
12     return defrost(t);
13   }
14 }
```

Listing 6.3. Partial computation class table

After the initial $\texttt{freeze}[\texttt{eager}](e)$ operation, we obtain a thunk $\ulcorner e' \texttt{ with } \emptyset \texttt{ from } _\urcorner$ where e' denotes:

```
int x = 2 + 2; int y = 3 + 3; return p.execute(freeze(x * y));.
```

The remote procedure call `o.execute(...);` triggers $\texttt{defrost}(e')$ at the location of object identifier o. This means that location begins executing the code e'. After performing the simple calculations in the declarations of variables x and y, o must create a new thunk which is sent to p. The thunk becomes $\ulcorner 4*6 \texttt{ with } \emptyset \texttt{ from } _\urcorner$ where we write $_$ for the location as it is not important. The defrost at p then performs the trivial computation of the result, 24, which is returned to o which in turn returns it to the original caller.

6.3 Remote method invocation with object serialisation and class downloading

This subsection gives the outline of the reduction steps for the first example in Section 2. We demonstrate how our operational semantics precisely model RMI with object serialisation, which is associated with class downloading. We first recall the remote classes at the server side. To avoid confusion, we append “Trad” to class names in Section 2 (Trad means “traditional”).

```
/** Server */
class TradServer {
  int doTask(Task t) {
    return t.compute();
  }
}
```

The following class is shared by the server and client.

```
/** Server and client */
class Task {
  int compute() { return 0; }
}
```

To show the effect of inheritance on class downloading, we slightly modify the programs in Section 2 as follows.

```
/** Server class */
class TradClient {
  TradServer s;
  TradClient(TradServer s) { this.s = s; }
  int gcd(int a, int b) {
    Task t = new GcdTask(a,b);
    return s.doTask(t);
  }
}
/** Client classes */
class GcdTask extends TaskMod{
  int a;
  int b;
  GcdTask(int a, int b) { this.a = a; this.b = b; }
  int compute() {
  // the same as the example in Section 2
  }
}
class TaskMod extends Task{
  int compute() { return new Mod().compute();}
}
class Mod extends Object{
  int compute() { return 2;}
}
```

In the client side, (1) there exists an inheritance such that $GcdTask <: TaskMod <: Task$; and (2) *TaskMod* has a reference to class *Mod* in method `compute()`. We also assume *Task* is shared between the server and client, but other classes are local to the client, hence do not initially exist in the server's class table. In summary, we have:

$$\sigma_{sv} = [s \mapsto (TradServer, ..)] \qquad \mathsf{dom}(\mathtt{CT}_{sv}) = \{TradServer, Task\}$$
$$\sigma_{cl} = \emptyset \qquad \mathsf{dom}(\mathtt{CT}_{cl}) = \{GcdTask, TaskMod, Mod, Task\}$$

As in the previous example in § 6.1, we consider the following initial network:

$$(\nu\, s)(cl[\mathtt{return}(c)\ \mathtt{new}\ \mathit{TradClient}(s).\mathtt{gcd}(1071,1029),\emptyset,\mathtt{CT}_{cl}]\,|\,sv[\mathbf{0},\sigma_{sv},\mathtt{CT}_{sv}])$$

The intermediate reduction steps are essentially similar to the reductions in § 6.1 until (6.0.3). We restart from the following configuration at the client side, focussing on the serialisation and deserialisation of the local object with identifier o.

$$\mathtt{go}\ s.\mathtt{doTask}(\mathtt{serialize}(o))\ \mathtt{with}\ f,...,\sigma_{cl}\cdot[o\mapsto(\mathit{GcdTask},a:1071,b:1029)],\mathtt{CT}_{cl}$$

Let $\sigma=[o\mapsto(\mathit{GcdTask},a:1071,b:1029)]$. Since $\mathit{GcdTask}$ is local, o method must be serialised for network transfer. Using the procedure outlined in Definition 4.8 (and illustrated in Example 4.9), serialisation takes a copy of the store σ. Hence we have:

$$\mathtt{go}\ s.\mathtt{doTask}(\lambda o.(o,\sigma,cl))\ \mathtt{with}\ f,...,\sigma_{cl}\cdot[o\mapsto(\mathit{GcdTask},a:1071,b:1029)],\mathtt{CT}_{cl}$$

Now go moves to the server, and at the same, the blob is deserialised, invoking class download.

$$\begin{array}{ll} & sv[s.\mathtt{doTask}(\mathtt{deserialize}(\lambda o.(o,\sigma,cl)))\ \mathtt{with}\ f,\sigma_{sv},\mathtt{CT}_{sv}] \\ \longrightarrow_{sv} & sv[(\nu\, o)(s.\mathtt{doTask}(\mathtt{download}\ \mathit{GcdTask}\ \mathtt{from}\ cl\ \mathtt{in}\ o),\sigma_{sv}\cdot\sigma,\mathtt{CT}_{sv})] \end{array} \quad (6.0.11)$$

Note that the name restriction operator guarantees the freshness of the o-id o in σ. Similarly to (6.0.8) in § 6.1, downloading calls a series of iteration between `resolve` and `download` as follows.

$$\begin{array}{ll} & s.\mathtt{doTask}(\mathtt{resolve}\ \mathit{GcdTask}\ \mathtt{from}\ cl\ \mathtt{in}\ o),\sigma_{sv}\cdot\sigma,\mathtt{CT}_{sv}\cdot[\mathit{GcdTask}\mapsto ..] \\ \longrightarrow_{sv} & s.\mathtt{doTask}(\mathtt{download}\ \mathit{TaskMod}\ \mathtt{from}\ cl\ \mathtt{in}\ o),\sigma_{sv}\cdot\sigma,\mathtt{CT}_{sv}\cdot[\mathit{GcdTask}\mapsto ..] \\ \twoheadrightarrow_{sv} & s.\mathtt{doTask}(o),\sigma_{sv}\cdot\sigma,\mathtt{CT}_{sv}\cdot[\mathit{GcdTask}\mapsto ..]\cdot[\mathit{TaskMod}\mapsto L[\mathit{Mod}^{cl}/\mathit{Mod}]] \end{array}$$

where L is the body of class *TaskMod*. We note that (1) all superclasses of *GcdTask* are downloaded before executing $s.\mathtt{doTask}(o)$; (2) *Mod* does *not* have to be downloaded, but it is replaced by Mod^{cl} to allow further lazy downloading from cl. For example, if `new TaskMod().compute()` is performed at the server side, then we can lazily download the class *Mod* using **RC-NewR** from cl. The rest of this reduction is similar to the local method invocation in § 6.1.

7 Typing system

This section presents the typing system for DJ. There are three technical key points for typing.

Linearity of channels Linear channel types guarantee determinacy of the destination of RMI and a return point.

Class signature its use offers the lightweight type checking for RMI and preserves consistency for serialisation and code freezing.

Freezing and thunks a special care is required for freezing and thunking expressions which contain free variables and object ids.

$T ::= \texttt{bool} \mid C \mid \texttt{thunk}(U) \mid \texttt{ser}(\vec{U})$	Types
$U ::= \texttt{void} \mid T$	Returnable types
$S ::= U \mid \texttt{ret}(U)$	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

Fig. 7.1. Syntax of types and environments

7.1 Syntax of types and environments

First we introduce the syntax of types and environments in Fig. 7.1.
T represents expression types: booleans, class names, thunked expressions of type U and serialised objects of type C. The metavariable U ranges over the same types as T but is augmented with the special type `void` with the usual empty meaning. We write $C <: D$ when class C is a subtype of class D. Our notion of subtyping is mostly standard (we assume $<:$ causes no cycle as in [18, 4]), and is judged on the class signature.

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 ($U\,\texttt{m}(\vec{T}\vec{x})\{e\}$ is well-typed if e has the type $\texttt{ret}(U)$). The metavariable τ ranges over types for the 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.

7.2 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 Fig. 7.1, $\texttt{chanI}(U)$ 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 $\texttt{return}(c)\ e$ or [go] e `with/to` c). Note that since channels only appear at runtime, channels do not carry channels.

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.

$$l_1[E_1[\texttt{await}\ c], \sigma_1, \texttt{CT}_1] \mid l_2[E_2[\texttt{await}\ c], \sigma_2, \texttt{CT}_2] \mid l_3[\texttt{go}\ v\ \texttt{to}\ c, \sigma_3, \texttt{CT}_3] \quad (7.0.12)$$

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

$$l_1[\texttt{return}(c)\ e_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[\texttt{return}(c)\ e_2, \sigma_2, \texttt{CT}_2] \tag{7.0.13}$$

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

Definition 7.1 (Channel environment composition). 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 $\texttt{chan}$, hence it becomes uncomposable because $\texttt{chan} \not\asymp \tau$ for any τ. Intuitively if P is typed by environment Δ_1 and Q by Δ_2, and if $\Delta_1 \asymp \Delta_2$, then we can compose P and Q as $P \,|\, Q$ safely, preserving channel linearity. Hence (7.0.12) is untypable because of $\texttt{chanI}(U) \not\asymp \texttt{chanI}(U)$ at c. (7.0.13) is too by $\texttt{chanO}(U) \not\asymp \texttt{chanO}(U)$ at c.

7.3 Subtyping

The notion of subtyping in DJ is mostly standard. The judgements are shown in Fig. 7.2.

ST-Refl
$$T <: T$$

ST-Trans
$$\frac{C <: D \qquad D <: E}{C <: E}$$

ST-Expr
$$\frac{U' <: U}{\texttt{thunk}(U') <: \texttt{thunk}(U) \quad \texttt{ret}(U') <: \texttt{ret}(U)}$$

ST-Vec
$$\frac{U_i' <: U_i \qquad 0 \leq i \lneq n}{\vec{U}' <: \vec{U}}$$

ST-Ser
$$\frac{\vec{U}' <: \vec{U}}{\texttt{ser}(\vec{U}') <: \texttt{ser}(\vec{U})}$$

ST-Class
$$\frac{\mathsf{CSig}(C) = \texttt{extends}\ D \ldots}{C <: D}$$

Fig. 7.2. Subtyping

The rule **ST-Refl** is a reflexive subtyping judgement—any type is a trivial subtype of itself. **ST-Trans** ensures that the subtyping relation is transitive with respect to classes—if class C is a subtype of class D and class D is a subtype of class E, then C is said to be a subtype of E. We assume there is no atomic subtyping. In **ST-Expr**, expression subtyping is introduced. For example, $\texttt{thunk}(U') <: \texttt{thunk}(U)$ ensures that a thunk

that computes a more precise value can be safely used in place of a thunk of a coarser type. Likewise, the judgement $\mathtt{ret}(U') <: \mathtt{ret}(U)$ is essential for assigning types to method bodies for the same reason. We make a similar argument for **ST-Ser**.

Importantly, rule **ST-Class** is used to judge subtypes between classes. It is entirely based upon the class signature CSig, rather than any particular class table entries. This allows judgements in the type system to be more lightweight. For example, if we had opted to judge subtypes based on the knowledge held only in class tables this would have required typing judgements of the form $\Gamma \vdash_{\mathtt{CT}} e : U$, with the current class table $\mathtt{CT}$ augmented each time new classes were downloaded or discovered.

7.4 Well-formedness

Well-formedness is defined for types, environments, stores and class tables. There are six kinds of judgement, and all are interrelated. Below we assume α ranges over either $\tau, S, U, \vec{U}$ or $\mathtt{extends}\ D\ [\mathtt{remote}]\ \vec{T}\vec{\mathtt{f}}\ \ \{\mathtt{m_i} : \vec{T_i} \to U_i\}$.

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

The first judgement ensures that the two environments respect several rules. We first consider the expression environment, Γ. This environment is completely separate from the notion of channels in the language, and stores three kinds of information: mappings from object identifiers to classes $o : C$, mappings from local variables to other types $x : T$ and the mapping from the special expression $\mathtt{this}$ to the class of the current receiver $\mathtt{this} : C$. The channel environment may only contain mappings of channel names to channel types (ranged over τ).

To construct well-formed environments, only well-formed types and class types may be used. The judgement $\vdash \alpha : \mathtt{tp}$ provides this and is defined in Fig. 7.4 Note that CSig only contains well-formed types; and C is well-formed if its CSig entry is so.

E-Nil
$$\emptyset \vdash \mathtt{Env}$$

E-Var
$$\frac{\vdash T : \mathtt{tp} \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma, x : T \vdash \mathtt{Env}}$$

E-Oid
$$\frac{\vdash C : \mathtt{tp} \qquad o \notin \mathsf{dom}(\Gamma)}{\Gamma, o : C \vdash \mathtt{Env}}$$

E-This
$$\frac{\vdash C : \mathtt{tp} \qquad \mathtt{this} \notin \mathsf{dom}(\Gamma)}{\Gamma, \mathtt{this} : C \vdash \mathtt{Env}}$$

E-CNil
$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma;\emptyset \vdash \mathtt{Env}}$$

E-Chan
$$\frac{\vdash \tau : \mathtt{tp} \qquad \Gamma;\Delta \vdash \mathtt{Env} \qquad c \notin \mathsf{dom}(\Delta)}{\Gamma;\Delta, c : \tau \vdash \mathtt{Env}}$$

Fig. 7.3. Well-formed environments

Wf-Base
$$\frac{\vdash \texttt{Env}}{\vdash \texttt{void} : \texttt{tp} \quad \vdash \texttt{bool} : \texttt{tp} \quad \vdash \texttt{chan} : \texttt{tp}}$$

Wf-SC
$$\frac{\vdash U : \texttt{tp} \vee U \in \mathsf{CSig}}{\vdash \texttt{chanI}(U) : \texttt{tp} \quad \vdash \texttt{chanO}(U) : \texttt{tp} \quad \vdash \texttt{thunk}(U) : \texttt{tp} \quad \vdash \texttt{ret}(U) : \texttt{tp}}$$

Wf-Vec
$$\frac{\vdash U_i : \texttt{tp}}{\vdash \vec{U} : \texttt{tp}}$$

Wf-Ser
$$\frac{\vdash \vec{U} : \texttt{tp}}{\vdash \texttt{ser}(\vec{U}) : \texttt{tp}}$$

Wf-Sig
$$\frac{\mathsf{override}(\texttt{m}_\texttt{i}, D_i, \vec{T}_i \rightarrow U_i) \vdash D : \texttt{tp} \qquad \forall S \in \{\vec{T}, \vec{U}, \vec{T}_i\} \ \vdash S : \texttt{tp} \vee S \in \mathsf{dom}(\mathsf{CSig})}{\vdash \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}} \ \ \{\texttt{m}_\texttt{i} : \vec{T}_i \rightarrow U_i\} : \texttt{tp}}$$

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

Wf-Ctp
$$\frac{\vdash \mathsf{CSig}(C) : \texttt{tp}}{\vdash C : \texttt{tp}}$$

Fig. 7.4. Well-formed types

7.5 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 has type α in expression environment Γ

where α ranges over T, U and S.

The typing rules for values are given in Fig. 7.7. The rule **TV-Null** allows the assignment of any well-formed class type to the value `null`. **TV-Thunk** states that a thunked expression is typed $\texttt{thunk}(U)$ if the class table `CT` that it was packaged with is well-formed, and if the expression itself is of type U. A serialised object is well-typed by rule **TV-Blob** if the values it contains are well-typed, and if the store is well-typed according to the types of the object identifiers it contains. It must also be the case that every bound object identifier o_i has the type of a *local* class.

The rules for assigning types to expressions are given in Fig. 7.8. They rely only on the expression environment Γ and so should be very similar in form to the typing rules of other Java languages [18, 4, 9]. Note that there is no subsumption rule in the type

S-Nil
$$\Gamma \vdash \emptyset : \texttt{ok}$$

S-Var
$$\frac{\Gamma \vdash \sigma : \texttt{ok} \qquad \Gamma \vdash x : T \qquad \Gamma \vdash v : T' \qquad T' <: T \qquad x \notin \mathsf{dom}_\mathsf{v}(\sigma)}{\Gamma \vdash \sigma \cdot [x \mapsto v] : \texttt{ok}}$$

S-Oid
$$\frac{\Gamma \vdash \sigma : \texttt{ok} \qquad \Gamma \vdash o : C \qquad \Gamma \vdash v_i : T_i' \qquad T_i' <: T_i \qquad o \notin \mathsf{dom}_\mathsf{o}(\sigma) \qquad \mathsf{fields}(C) = \vec{T}\vec{f}}{\Gamma \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \texttt{ok}}$$

Fig. 7.5. Well-formed stores

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

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

CT-Nil
$$\vdash\emptyset:\texttt{ok}$$

CT
$$\frac{\vdash\texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f};K\vec{M}\}:\texttt{ok} \qquad \vdash\texttt{CT}:\texttt{ok}}{\vdash\texttt{CT}\cdot[C\mapsto\texttt{class}\ C\ \texttt{extends}\ D\{\vec{T}\vec{f};K\vec{M}\}]:\texttt{ok}}$$

Fig. 7.6. Well-formed class tables

system. Instead we explicitly annotate each place where subtypes can be used in place of a supertype. We only explain the rules which differ from [18, 4, 9].

TE-Fld restricts field accesses only for local classes if e is neither `this` or o. On the other hand, we can allow `this` has a remote class because `this` is always instantiated by the o-id whose store exists in that location (see **RC-MethInvoke**). This constraint, together with the initial condition defined in Section 9 guarantees that field access is always local. **TE-Return** and **TE-ReturnVoid** ensure that a return statement has the return type. The rule for serialisation, **TE-Serialize**, assigns types to the term $\texttt{serialize}(e)$. The rule for deserialisation, **TE-Defrost**, is the dual to this.

Among all typing rules, the most complicated rule is **TE-Freeze**. For an expression $\texttt{freeze}[t](e)$ to be well-typed with type $\texttt{thunk}(U)$, several conditions must hold:

– Firstly, the expression e must have the type U in environment Γ, written $\Gamma\vdash e:U$.
– Secondly, every free name or variable mentioned in the expression must have a primitive type or be a reference to an object identifier of a remote class This is guaranteed by $\{\vec{u}\}=\mathsf{fnv}(e)$, $\neg\mathsf{local}(T_i)$ and $\Gamma\vdash u_i:T_i$. This requirement simplifies

TV-Bool
$$\frac{\Gamma\vdash\texttt{Env}}{\Gamma\vdash\texttt{true}:\texttt{bool} \quad \Gamma\vdash\texttt{false}:\texttt{bool}}$$

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

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

TV-Empty
$$\frac{\Gamma\vdash\texttt{Env}}{\Gamma\vdash\varepsilon:\texttt{void}}$$

TV-Thunk
$$\frac{\vdash\texttt{CT}:\texttt{ok} \qquad \Gamma\vdash e:U}{\Gamma\vdash\ulcorner e\ \texttt{with CT from}\ l\urcorner:\texttt{thunk}(U)}$$

TV-Blob
$$\frac{\Gamma,\vec{o}:\vec{C}\vdash\vec{v}:\vec{U} \qquad \mathsf{local}(C_i) \qquad \Gamma,\vec{o}:\vec{C}\vdash\sigma:\texttt{ok} \qquad \{\vec{o}\}=\mathsf{dom_o}(\sigma)}{\Gamma\vdash\lambda\vec{o}.(\vec{v},\sigma,l):\texttt{ser}(\vec{U})}$$

Fig. 7.7. Rules for values

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

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

TE-Cond
$$\frac{\exists S : S_1 <: S \wedge S_2 <: S \quad \Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : S}$$

TE-Fld
$$\frac{\Gamma \vdash e : C \quad \vdash C : \texttt{tp} \quad e \neq \texttt{this}, o \implies \mathsf{local}(C) \quad \mathsf{fields}(C) = \vec{T}\vec{f}}{\Gamma \vdash e.\texttt{f}_\texttt{i} : T_i}$$

TE-Seq
$$\frac{\Gamma \vdash e : \texttt{void} \quad \Gamma \vdash e' : S}{\Gamma \vdash e; e' : S}$$

TE-Dec
$$\frac{\Gamma \vdash e : T' \quad \Gamma, x : T \vdash e_0 : S \quad T' <: T}{\Gamma \vdash T\ x = e\ ;\ e_0 : S}$$

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

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

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

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

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

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

TE-Serialize
$$\frac{\Gamma \vdash e : \vec{U}}{\Gamma \vdash \texttt{serialize}(e) : \texttt{ser}(\vec{U})}$$

TE-Deserialize
$$\frac{\Gamma \vdash e : \texttt{ser}(\vec{U})}{\Gamma \vdash \texttt{deserialize}(e) : \vec{U}}$$

TE-Freeze
$$\frac{\{\vec{u}\} = \mathsf{fnv}(e) \quad \mathsf{fav}(e) = \emptyset \quad \neg\mathsf{local}(T_i) \quad \Gamma \vdash e : U \quad \Gamma \vdash u_i : T_i}{\Gamma \vdash \texttt{freeze}[t](e) : \texttt{thunk}(U)}$$

TE-Defrost
$$\frac{\Gamma \vdash e : \texttt{thunk}(U)}{\Gamma \vdash \texttt{defrost}(e) : U}$$

TE-Pe
$$\frac{\Gamma \vdash pe : T}{\Gamma \vdash pe : \texttt{void}}$$

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

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

Fig. 7.8. Rules for expressions

the burden of the freezing operation - if local object identifiers were allowed to be passed as part of thunks then there would be the heavy requirement of serialising every object identifier in a similar fashion to remote method invocation.

– Finally, in order to prevent the variable replacement strategy of **RC-Freeze** from being too naïve (for example, $\texttt{freeze}[t](x := 5)$ could be re-written to

$\texttt{freeze}[t](6 := 5)$ using straight substitutions), we require that the expression to be frozen contain *no free assigned variables*. This condition is written by $\mathsf{fav}(e) = \emptyset$ which was defined in Definition 4.5.

The last two conditions are designed to ensure that a shipped thunk does not leak free local object identifiers and variables, satisfying the variable and o-id invariants defined in Definition 9.2. **TE-Defrost** is the dual to this.

TE-Pe is a rule for typing a sequential composition [4]. The typed context which starts from **TE-Hole** is used to type `await` c as explained in the next subsection. Other rules are obvious.

Remark 7.2 (Freezing). The operational semantics as well as the typing system of `freeze` is simplified assuming e does not contain any free variables as shown in the following rules.

$$\frac{\texttt{CT}' = \begin{cases} \mathsf{cg}(\texttt{CT}, \mathsf{fcl}(e)) & t = \texttt{eager} \\ \emptyset & t = \texttt{lazy} \end{cases}}{\texttt{freeze}[t](e), \sigma, \texttt{CT} \longrightarrow_l \ulcorner e \;\texttt{with}\; \texttt{CT}' \;\texttt{from}\; l \urcorner, \sigma, \texttt{CT}} \qquad \frac{\mathsf{fv}(e) = \emptyset \;\; \{\vec{u}\} = \mathsf{fn}(e) \;\; \mathsf{remote}(C_i) \quad \Gamma \vdash u_i : C_i \quad \Gamma \vdash e : U}{\Gamma \vdash \texttt{freeze}[t](e) : \texttt{thunk}(U)}$$

One can check these rules satisfy the same invariants specified in Section 9.

Also, for simplification, the current typing rule **TE-Freeze** does not allow creating code which contains free local object ids. However, by combining with serialisation primitives, we can type code which contains serialised object graph like:

$$T\ x = \texttt{serialize}(o)\ ;\ \texttt{freeze}[t](e')$$

Note that $\texttt{serialize}(o)$ creates a closed value. As such, combination of two kinds of distributed primitives offers a flexible high-level programming style.

7.6 Threads typing

Threads are assigned a type based on both the expression environment Γ and the channel environment Δ. The judgement takes a form of:

$\Gamma; \Delta \vdash P : \texttt{thread}$ $\quad$ P is well-typed (has type `thread`) in environment $\Gamma; \Delta$

The rules for assigning types to threads are shown in Fig. 7.9. **TT-Nil** states that the inactive process is well-typed in any well-formed environment. The key rule is **TT-Par**; we type a parallel compositions of threads if a composition of two channel environments preserve the linearity of channels. This is checked by $\Delta_1 \asymp \Delta_2$ (see § 7.2). **TT-Res** is a standard rule for typing restricted channel names [21, 15]. We only allow new channels to be assigned the type `chan`. Similarly we restrict a variable when it exists in the store. Related to this definition are the rule **TT-Weak** which states that it is safe to add as many "matched" channels to the environment, provided of course those names are not already present in the environment.

The rule **TT-Await** types the `await` c expression, which corresponds to the return point for method calls. It expects the channel c to be expecting an input of type U, which can be safely plugged into the waiting context. **TT-Return** types method bodies and can be thought of as the dual to **TT-Await** - it expects channel c to be used for output of a type U which can carry a supertype of return value of the method. The remainder of the rules handle the formalities of method call and remote invocation.

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

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

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

TT-Weak
$$\frac{\Gamma;\Delta \vdash P : \mathtt{thread} \qquad c \notin \mathsf{dom}(\Delta)}{\Gamma;\Delta, c : \mathtt{chan} \vdash P : \mathtt{thread}}$$

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

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

TT-GoSer
$$\frac{\Gamma \vdash o : C \qquad \Gamma \vdash \vec{v} : \vec{T}' \qquad \vec{T}' <: \vec{T} \qquad \mathsf{remote}(C) \qquad \mathsf{mtype}(\mathtt{m}, C) = \vec{T} \rightarrow U}{\Gamma; c : \mathtt{chanO}(U) \vdash \mathsf{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c : \mathtt{thread}}$$

TT-MethWith
$$\frac{\Gamma \vdash o : C \qquad \Gamma \vdash \vec{v} : \vec{T}' \qquad \vec{T}' <: \vec{T} \qquad \mathsf{mtype}(\mathtt{m}, C) = \vec{T} \rightarrow U}{\Gamma; c : \mathtt{chanO}(U) \vdash o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c : \mathtt{thread}}$$

TT-DeserWith
$$\frac{\begin{array}{c}\Gamma \vdash o : C \\ \Gamma \vdash \lambda\vec{o}.(\vec{v}, \sigma, l) : \mathtt{ser}(\vec{T}') \qquad \vec{T}' <: \vec{T} \qquad \mathsf{remote}(C) \qquad \mathsf{mtype}(\mathtt{m}, C) = \vec{T} \rightarrow U\end{array}}{\begin{array}{c}\Gamma; c : \mathtt{chanO}(U) \vdash o.\mathtt{m}(\mathtt{deserialize}(\lambda\vec{o}.(\vec{v}, \sigma, l)))\ \mathtt{with}\ c : \mathtt{thread} \\ \Gamma; c : \mathtt{chanO}(U) \vdash \mathsf{go}\ o.\mathtt{m}(\lambda\vec{o}.(\vec{v}, \sigma, l))\ \mathtt{with}\ c : \mathtt{thread}\end{array}}$$

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

TT-GoTo
$$\frac{\Gamma \vdash e : \mathtt{ser}(C') \quad C' <: C}{\Gamma; c : \mathtt{chanO}(C) \vdash \mathsf{go}\ e\ \mathtt{to}\ c : \mathtt{thread}}$$

Fig. 7.9. Rules for threads

7.7 Networks typing

Like threads, networks and configurations are typed in both the expression and channel environment. There are two judgements:

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

The typing rules are given in Fig. 7.10.

Restricted identifiers in configurations and networks are typed by the rules **TC-ResId** and **TN-ResId** respectively. They share a side-condition of a common form: $u \in \mathsf{dom}(F)$ ensures that we do not create spurious new identifiers that do not correspond to an element of the domain of the store of F, and likewise $u \in \mathsf{dom}(N)$ ensures a similar condition at the network level.

$$\textbf{TC-ResC}\ \frac{\Gamma;\Delta,c:\mathtt{chan}\vdash F:\mathtt{conf}}{\Gamma;\Delta\vdash(\nu c)F:\mathtt{conf}} \qquad \textbf{TC-ResId}\ \frac{\Gamma,u:T;\Delta\vdash F:\mathtt{conf}\quad u\in\mathsf{dom}(F)}{\Gamma;\Delta\vdash(\nu u)F:\mathtt{conf}} \qquad \textbf{TC-Conf}\ \frac{\Gamma;\Delta\vdash P:\mathtt{thread}\quad \Gamma\vdash\sigma:\mathtt{ok}\quad \vdash\mathtt{CT}:\mathtt{ok}\quad \mathtt{FCT}\subseteq\mathtt{CT}}{\Gamma;\Delta\vdash P,\sigma,\mathtt{CT}:\mathtt{conf}}$$

$$\textbf{TN-Nil}\ \frac{\Gamma;\emptyset\vdash\mathtt{Env}}{\Gamma;\emptyset\vdash\mathbf{0}:\mathtt{net}} \qquad \textbf{TN-Conf}\ \frac{\Gamma;\Delta\vdash F:\mathtt{conf}}{\Gamma;\Delta\vdash l[F]:\mathtt{net}} \qquad \textbf{TN-Par}\ \frac{\Gamma;\Delta_i\vdash N_i:\mathtt{net}\quad \Delta_1\asymp\Delta_2\quad \mathsf{dom}(N_1)\cap\mathsf{dom}(N_2)=\emptyset\quad \mathsf{loc}(N_1)\cap\mathsf{loc}(N_2)=\emptyset}{\Gamma;\Delta_1\odot\Delta_2\vdash N_1\,|\,N_2:\mathtt{net}}$$

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

Fig. 7.10. Rules for networks and configurations

The rule **TC-Conf** states that a configuration is only well-typed in an environment $\Gamma;\Delta$ if its threads, P and store σ are well-typed in the same environment. Its class table must also be well-formed, and must contain a copy of the foundation classes `FCT`.

The rule **TN-Nil** has the standard meaning - an inactive network is well-typed in any well-formed environment. The rule **TN-Par** states that the parallel composition of two networks is well-typed if their typing environments are composable, if they share no identifiers of stores and if they have disjoint sets of location names. The first condition $\Delta_1 \asymp \Delta_2$ is understood as **TT-Par**. The rest of the rules, **TN-ResC** and **TN-Weak** are understood as **TC-ResC** and **TT-Weak**, respectively.

8 Basic Properties

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

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

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

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

The first lemma we introduce is Lemma 8.1. This concerns the *canonical forms* of DJ. We prove that every typable network can be written in such a form. Intuitively, a

canonical form is one in which all restricted identifiers are moved out to the network level.

Lemma 8.1 (Canonical forms). *Suppose that* $\Gamma;\Delta \vdash N : \mathtt{net}$ *then*

$$N \equiv (\nu\,\vec{u})(\prod_{0\leq i<n} l_i[P_i,\sigma_i,\mathtt{CT}_i])$$

where n denotes the number of locations in N.

Proof. By induction on the number of networks in parallel, n. Suppose n is 0. Then by our assumptions $\Gamma;\Delta \vdash \mathbf{0} : \mathtt{net}$. This is trivially equivalent to the zero-product: $(\nu\,\vec{u})(\prod_{0\leq i<0} l_i[P_i,\sigma_i,\mathtt{CT}_i])$.

For the inductive step, we suppose that the statement holds for n networks and show that by adding the $n+1$, it is still equivalent to a normal form by the structural rules in Fig. 5.1. Suppose then, by assumption:

$$\Gamma;\Delta \vdash (\nu\,\vec{u}_{n+1})l_{n+1}[P_{n+1},\sigma_{n+1},\mathtt{CT}_{n+1}]\,|\,N' : \mathtt{net}$$

By the inductive hypothesis, we have that:

$$N' \equiv (\nu\,\vec{u}_n)(\prod_{0\leq i<n} l_i[P_i,\sigma_i,\mathtt{CT}_i])$$

We can apply alpha-equivalence to the restricted names $(\nu\,\vec{u}_{n+1})$ to ensure that they do not clash with any of those names in N'. This allows us to apply scope opening to obtain:

$$(\nu\,\vec{u}_{n+1})(l_{n+1}[P_{n+1},\sigma_{n+1},\mathtt{CT}_{n+1}]\,|\,N')$$

Again, we can apply alpha equivalence this time to ensure the names bound in $\vec{u}_n$ do not clash with those in location l_{n+1}. Apply structural equivalence to obtain:

$$(\nu\,\vec{u}_{n+1}\vec{u}_n)(l_{n+1}[P_{n+1},\sigma_{n+1},\mathtt{CT}_{n+1}]\,|\prod_{0\leq i<n} l_i[P_i,\sigma_i,\mathtt{CT}_i])$$

This can be straightforwardly rewritten to:

$$(\nu\,\vec{u}_{n+1}\vec{u}_n)(\prod_{0\leq i<n+1} l_i[P_i,\sigma_i,\mathtt{CT}_i])$$

This completes the case. □

8.1 Judgements

Lemma 8.2 lists some useful properties about judgements. We write J to stand for any one of the following judgements:

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

Lemma 8.2(3) has the useful property of ensuring that any channels appearing in the channel environment Δ and not in the judgement J must have the linear type chan.

Lemma 8.2 (Judgements).

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

Proof. By induction on the size of the judgement $\mathtt{J}$. All cases are straightforward. We only list the proof for weakening with the case $\mathtt{J} = P_1 \,|\, P_2 : \mathtt{thread}$. After applying rule **TT-Par** we have two cases; we can apply the inductive hypothesis to either the left branch or the right branch of the parallel composition. For example, choose the left branch. Therefore $\Gamma;\Delta_1,c:\mathtt{chan} \vdash P_1 : \mathtt{thread}$ and $\Delta_1,c:\mathtt{chan} \asymp \Delta_2$ as $c \notin \mathsf{dom}(\Delta_2)$. Apply **TT-Par** to yield $\Gamma;\Delta_1,c:\mathtt{chan} \odot \Delta_2 \vdash P_1 \,|\, P_2 : \mathtt{thread}$. Then $\Gamma;\Delta_1 \odot \Delta_2, c : \mathtt{chan} \vdash P_1 \,|\, P_2 : \mathtt{thread}$ by definition of $\odot$. The other case proceeds similarly. □

8.2 Stores

Lemma 8.3 states properties about the type-safety of store access. Store access are defined as adding new variable and object identifier mappings, updating the fields of objects and the value held by a variable, and also retrieving information from variables and object fields. Lemma 8.3(7) allows the concatenation of disjoint stores and is useful in typing the $\mathtt{deserialize}(e)$ operation.

Lemma 8.3 (Stores).

1. *Assume* $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash v : T'$ *with* $x \notin \mathsf{dom}(\Gamma)$ *and* $T' <: T$. *Then* $\Gamma,x:T \vdash \sigma \cdot [x \mapsto v] : \mathtt{ok}$.
2. *Assume* $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash x : T$ *and* $\Gamma \vdash v : T'$ *with* $T' <: T$. *Then* $\Gamma \vdash \sigma[x \mapsto v] : \mathtt{ok}$.
3. $\Gamma \vdash x : T$ *and* $\Gamma \vdash \sigma : \mathtt{ok}$ *imply* $\Gamma \vdash \sigma(x) : T'$ *with* $T' <: T$.
4. *Assume* $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash \vec{v} : \vec{T}'$ *with* $\mathsf{fields}(C) = \vec{T}\vec{f}$, $T_i' <: T_i$ *and* $o \notin \mathsf{dom}(\Gamma)$. *Then we have* $\Gamma,o:C \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \mathtt{ok}$.
5. *If* $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash o : C$ *and* $\Gamma \vdash v : T_i'$ *with* $\mathsf{fields}(C) = \vec{T}\vec{f}$ *and* $T_i' <: T_i$, *then* $\Gamma \vdash \sigma[o \mapsto \sigma(o)[f_i \mapsto v]] : \mathtt{ok}$.
6. *Assume* $\Gamma \vdash \sigma : \mathtt{ok}$ *and* $\Gamma \vdash o.\mathtt{f_i} : T_i$ *with* $\sigma(o) = (C, \vec{f} : \vec{v})$. *Then* $\Gamma \vdash v_i : T_i'$ *where* $T_i' <: T_i$.
7. *Suppose* $\Gamma \vdash \sigma : \mathtt{ok}$ *and* $\Gamma,\Gamma' \vdash \sigma' : \mathtt{ok}$ *with* $\mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset$, *then* $\Gamma,\Gamma' \vdash \sigma \cup \sigma' : \mathtt{ok}$.

Proof. By induction on the length of store σ, written length(σ). We only prove (4) and (7). Others are similar.

(4)

Case length$(\sigma) = 0$*:* Trivially, $o \notin \mathsf{dom_o}(\emptyset)$ and by our assumptions we can immediately conclude $\Gamma, o : C \vdash \emptyset \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \mathtt{ok}$ as required.

Case length$(\sigma) = n+1$*:* Suppose that $\sigma' = \sigma \cdot [\ldots \mapsto \ldots]$ (thus $\text{length}(\sigma') = \text{length}(\sigma) + 1$). Then the premises of the lemma state:

$$\Gamma \vdash \sigma' : \mathtt{ok} \qquad \text{with } o \notin \mathsf{dom}(\Gamma) \qquad (8.3.1)$$

$$\Gamma \vdash \vec{v} : \vec{T}' \qquad \text{with } \mathsf{fields}(C) = \vec{f}\vec{T} \text{ and } T_i' <: T_i \qquad (8.3.2)$$

$$\Gamma, o : C \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \mathtt{ok} \qquad \text{by the assumption} \qquad (8.3.3)$$

(8.3.3) must have been derived by **S-Oid** with premises:

$$\Gamma, o : C \vdash \sigma : \mathtt{ok} \qquad \text{with } o \notin \mathsf{dom_o}(\sigma) \qquad (8.3.4)$$

$$\Gamma, o : C \vdash o : C \qquad (8.3.5)$$

$$\Gamma, o : C \vdash \vec{v} : \vec{T}' \qquad \text{with } \mathsf{fields}(C) = \vec{f}\vec{T} \text{ and } T_i' <: T_i \qquad (8.3.6)$$

Applying Lemma 8.2(9) to (8.3.4) we obtain $\Gamma, o : C \vdash \mathtt{Env}$ which must have been derived from rule **E-Oid** with premise $\Gamma \vdash C : \mathtt{tp}$ with $o \notin \mathsf{dom}(\Gamma)$. By this side condition and (8.3.1), we can apply Lemma 8.2(5) to obtain:

$$\Gamma, o : C \vdash \sigma' : \mathtt{ok} \qquad \text{with } o \notin \mathsf{dom}(\sigma') \qquad (8.3.7)$$

To complete the case, we apply **S-Oid** to (8.3.5), (8.3.6) and (8.3.7), obtaining $\Gamma, o : C \vdash \sigma' \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \mathtt{ok}$, as required.

(7)

Case length$(\sigma') = 0$*:* This is the base case. If $\text{length}(\sigma') = 0$ it must be the case that σ' is empty, and this means that $\sigma \cup \sigma' = \sigma$. By Lemma 8.2(9), we can take $\Gamma, \Gamma' \vdash \sigma' : \mathtt{ok}$ and deduce $\Gamma, \Gamma' \vdash \mathtt{Env}$. Therefore $\Gamma, \Gamma' \vdash \sigma \cup \sigma' : \mathtt{ok}$ as required.

Case length$(\sigma') = n+1$*:* The inductive step. We assume that for a store of $\text{length}(\sigma') = n$ the statement holds, and prove for $\text{length}(\sigma' \cdot [. \mapsto ...]) = n+1$. We must perform a case analysis on the last item appended to the store. This can either be a variable to value mapping, or an object identifier to store object mapping. We shall only consider the case for variables; the other case is similar. Suppose, by the assumption:

$$\Gamma, \Gamma' \vdash \sigma \cup \sigma' : \mathtt{ok} \qquad \text{with } \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma') = \emptyset \qquad (8.3.8)$$

Now assuming the last item appended to σ' was a variable mapping, then by the premises of **S-Var** we have:

$$\Gamma, \Gamma', x : T \vdash \sigma' : \mathtt{ok} \qquad \text{with } x \notin \mathsf{dom_v}(\sigma') \qquad (8.3.9)$$

$$\Gamma, \Gamma', x : T \vdash x : T \qquad (8.3.10)$$

$$\Gamma, \Gamma', x : T \vdash v : T' \qquad \text{and } T' <: T \qquad (8.3.11)$$

We apply weakening to (8.3.11) to obtain:

$$\Gamma,\Gamma' \vdash v : T' \qquad \text{and } T' <: T \tag{8.3.12}$$

From (8.3.9) we can apply Lemma 8.2(9) to deduce that $\Gamma,\Gamma',x : T \vdash \texttt{Env}$. This means that $x \notin \mathsf{dom}(\Gamma,\Gamma')$. Given this fact, (8.3.8) and (8.3.12), we can apply Lemma 8.3(1) to obtain $\Gamma,\Gamma',x : T \vdash \sigma \cup \sigma' \cdot [x \mapsto v] : \texttt{ok}$, as required. □

8.3 Graphs

In DJ, two kinds of graph are computed: object graphs and class graphs. Lemma 8.4 proves the correctness of the algorithms presented in Definition 4.8 and Definition 4.12 respectively. These properties are the key to prove the type soundness theorem. Note that the definition of "complete" appears in Definition 4.12.

Lemma 8.4 (Graph computation).

1. *Assume* $\Gamma \vdash \sigma : \texttt{ok}$, $\Gamma \vdash o : C$ *and* $\sigma' = \texttt{og}(\sigma,o)$. *Then we have* $\Gamma \vdash \sigma' : \texttt{ok}$. *Also, for all* $o' \in \mathsf{dom_o}(\sigma')$ *such that* $\sigma'(o') = (C,\dots)$, *we have* $\mathsf{local}(C)$.
2. $\sigma' = \texttt{og}(\sigma,v)$ *implies* $\mathsf{og_comp}(\sigma,\sigma')$.
3. *Suppose* $\vdash \texttt{CT} : \texttt{ok}$ *with* $C \in \mathsf{dom}(\mathsf{CSig})$. *Then we have* $\vdash \texttt{cg}(\texttt{CT},C) : \texttt{ok}$.
4. $\mathsf{ct_comp}(\texttt{CT})$ *and* $\texttt{CT}' = \texttt{cg}(\texttt{CT},C)$ *imply* $\mathsf{ct_comp}(\texttt{CT}' \cup \texttt{FCT})$.

Proof. We show (1), (2) by induction on the length of σ, and (3), (4) by induction on the length of $\texttt{CT}$.

(1) We shall assume that $\Gamma \vdash \texttt{og}(\sigma_0,o) : \texttt{ok}$ such that $\text{length}(\sigma_0) < n$ and prove for $\text{length}(\sigma) = n$.

We shall also show that all objects in σ' are instances of *local* classes. Examining the definition of the object graph calculation algorithm we see there are two distinct cases. The base case, where the graph is the empty store $\emptyset$ is straightforward as $\Gamma \vdash \emptyset : \texttt{ok}$ always. The case where some computation happens is more difficult. Examining the definition of $\texttt{og}(\sigma,o)$ from Definition 4.8 we have that $\sigma_1 = \sigma \setminus \{o\}$. We can trivially re-order σ to ensure o is the last item. To infer the assumption $\Gamma \vdash \sigma : \texttt{ok}$, **S-Oid** must have been used with the premise that: $\Gamma \vdash \sigma_1 : \texttt{ok}$.

Examining the body of the algorithm we see $\text{length}(\sigma_{i+1}) < \text{length}(\sigma_i) < \text{length}(\sigma_1)$ where $\sigma' = \bigcup \texttt{og}(\sigma_i,o_i) \cup \sigma(o)$ with $\sigma_{i+1} = \sigma_i \setminus \mathsf{dom_o}(\texttt{og}(\sigma_i,o_i))$. By the inductive hypothesis, we immediately conclude that: $\Gamma \vdash \sigma_i : \texttt{ok}$ where $i > 1$. The singleton store $\sigma(o)$ is trivially well-formed in Γ. Therefore we can take $\Gamma \vdash \sigma(o) : \texttt{ok}$ by $\Gamma \vdash \sigma_i : \texttt{ok}$, in order to apply Lemma 8.3(7) obtaining $\Gamma \vdash \sigma' : \texttt{ok}$, as required.

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

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

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

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

(3) By induction on length$(\mathtt{CT})$. First consider a class graph of length 0, i.e. $\mathtt{CT} = \emptyset$. Suppose $\mathtt{CT}' = \mathsf{cg}(\emptyset, C)$, then by definition of the class graph algorithm $\mathtt{CT}' = \emptyset$. Clearly $\vdash \emptyset : \mathtt{ok}$ by **CT-Nil**. For the inductive step, assume that $\vdash \mathsf{cg}(\mathtt{CT}, C) : \mathtt{ok}$ for length$(\mathtt{CT}_n) = n$ and show for length$(\mathtt{CT}_{n+1}) = n + 1$. Suppose:

$$\vdash \mathtt{CT}_{n+1} : \mathtt{ok} \qquad \text{with } C \in \mathsf{dom}(\mathsf{CSig}) \tag{8.4.1}$$

$$\vdash \mathsf{cg}(\mathtt{CT}_n, C) : \mathtt{ok} \qquad \text{by the assumption} \tag{8.4.2}$$

We consider only the case where $C \in \mathsf{dom}(\mathtt{CT})$ and $C \notin \mathsf{dom}(\mathtt{FCT})$; the other is trivial. We shall prove: $\vdash \mathsf{cg}(\mathtt{CT}_{n+1}, C) : \mathtt{ok}$. Expanding the class graph algorithm by two steps, $\mathsf{cg}(\mathtt{CT}_{n+1}, C)$ is obtained as:

$$\mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, D) \cup \mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, \vec{M}) \cup [C \mapsto \mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}] \tag{8.4.3}$$

First we note that $\Gamma \vdash [C \mapsto \mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f}; K\vec{M}\}] : \mathtt{ok}$ by $C \in \mathsf{dom}(\mathtt{CT}_n)$ and $\Gamma \vdash \mathtt{CT}_n : \mathtt{ok}$. Also $\Gamma \vdash \mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, D)$ is proved by the inductive hypothesis. For the second item in the union, $\mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, \vec{M})$, we check for each method M_i in $\vec{M}$. Then we compute $\mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, U\, \mathtt{m_i}\, (\vec{T}\vec{x})\{e\})$ by expanding the algorithm by one step here yields:

$$\mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, e) \tag{8.4.4}$$

Then by inductive hypothesis, $\mathsf{cg}(\mathtt{CT}_{n+1} \setminus C, \vec{M})$ is well-defined. Finally by applying **CT**, we conclude the proof.

(4) Suppose length$(\mathtt{CT}_0) = 0$ then by definition, $\mathtt{CT}'$ is complete. Now, take length$(\mathtt{CT}_n) = n$. Given $\mathtt{CT}' = \mathsf{cg}(\mathtt{CT}_n, C)$ for some C is complete by assumption, we either have that $C \in \mathsf{dom}(\mathtt{CT}_n)$ and $\mathtt{CT}' \neq \emptyset$, or $C \notin \mathsf{dom}(\mathtt{CT}_n)$ and $\mathtt{CT}' = \emptyset$. For the inductive step we must show that when the length of the class table is $n+1$ the computed class graph remains complete. Extending the class table can be achieved by appending a new entry giving $\mathtt{CT}_{n+1} = \mathtt{CT}_n \cdot [C' \mapsto L]$ for some $C' \notin \mathsf{dom}(\mathtt{CT}_n)$. We assume that the superclass of C' is present in $\mathtt{CT}_n$, otherwise the new class table would not be complete and so the conclusion would hold by default. Then given $\mathtt{CT}' = \mathsf{cg}(\mathtt{CT}_{n+1}, C)$, if $C \neq C'$ then again $\mathtt{CT}'$ is complete by virtue of being empty. If $C = C'$ then by our assumption that the class table $\mathtt{CT}_{n+1}$ contains the direct superclass of C' then $\mathtt{CT}'$ must also be *complete*. □

Lemma 8.5 (Method body). *Suppose* $\mathsf{mbody}(\mathtt{m}, C, \mathtt{CT}) = (\vec{x}, e)$ *and* $\mathsf{mtype}(\mathtt{m}, C) = \vec{T} \to U$ *with* $\vdash \mathtt{CT} : \mathtt{ok}$. *Then for some* C' *where* $C <: C'$ *and some* U' *where* $U' <: U$ *then we have* $\vec{x} : \vec{T}, \mathtt{this} : C' \vdash e : \mathtt{ret}(U')$.

Proof. Straightforward. □

Lemma 8.6 (Context). $\Gamma \vdash E[\,]^U : \alpha$ *and* $\Gamma \vdash e : U'$ *with* $U' <: U$ *iff* $\Gamma \vdash E[e]^U : \alpha$.

Proof. By induction on the structure on E. □

8.4 Structural equivalence

An important property to be shown is that the application of the structural equality rules given in Fig. 5.1 preserves the typing of a term. This is proved in Lemma 8.8, but in order to do so we must make use of the equally important property shown in Lemma 8.7.

This lemma yields natural properties for the composability of environments and is used in many of the later proofs.

Lemma 8.7 (Commutativity of composition and composability).

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

Proof. In both proofs, without loss of generality we consider singleton environments such that $\Delta_1 = \{c : \mathtt{chanI}(U)\}$ and $\Delta_2 = \{c : \mathtt{chanO}(U)\}$ with $\Delta_1 \odot \Delta_2 = \{c : \mathtt{chan}\}$.

(1) For this case we show only the left-to-right direction, the opposite direction is similar. The only interesting case is that Δ_1 and Δ_2 share the same channels.

By the definition of $\asymp$, we know $c \notin \mathsf{dom}(\Delta_3)$. Since $\Delta_2 \odot \Delta_3 = \{c : \mathtt{chanO}(U)\} \cup \Delta_3$, we have that $\Delta_2 \asymp \Delta_3$ as required. We can also easily check $\Delta_1 \odot (\Delta_2 \odot \Delta_3)$ is defined, thus by definition of $\asymp$, we have $\Delta_1 \asymp (\Delta_2 \odot \Delta_3)$, as desired.

(2) Proceeds in a similar manner to (1), adopting the same singleton environments. We can easily check $(\Delta_1 \odot \Delta_2) \odot \Delta_3$ is defined and is equal to $\{c : \mathtt{chan}\} \cup \Delta_3$. Since $c \notin \mathsf{dom}(\Delta_3)$ then $\Delta_2 \odot \Delta_3 = \{c : \mathtt{chanO}(T)\} \cup \Delta_3$. By definition of $\odot$, $\Delta_1 \odot (\Delta_2 \odot \Delta_3) = \{c : \mathtt{chan}\} \cup \Delta_3 = (\Delta_1 \odot \Delta_2) \odot \Delta_3$ as required.

Lemma 8.8 (Structural equivalence preserves typability).

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

Proof. By induction on typing derivations paying attention to the last rule applied. Most cases are straightforward. We show one from each sub-lemma.

(1) We show the case $(\nu\, c)P,\sigma,\mathtt{CT} \equiv (\nu\, c)(P,\sigma,\mathtt{CT})$. Suppose:

$$\Gamma;\Delta \vdash (\nu\, c)P,\sigma,\mathtt{CT} : \mathtt{conf} \qquad c \notin \mathsf{fn}(\sigma) \cup \mathsf{fn}(\mathtt{CT}) \tag{8.8.1}$$

We shall prove:

$$\Gamma;\Delta \vdash (\nu\, c)(P,\sigma,\mathtt{CT}) : \mathtt{conf} \tag{8.8.2}$$

To infer (8.8.1), rule **TC-Conf** was applied with the premises:

$$\Gamma;\Delta \vdash (\nu\, c)P : \mathtt{thread} \tag{8.8.3}$$
$$\Gamma \vdash \sigma : \mathtt{ok} \tag{8.8.4}$$
$$\vdash \mathtt{CT} : \mathtt{ok} \tag{8.8.5}$$

(8.8.3) must have been derived by applying **TT-Res** with the following premise:

$$\Gamma;\Delta, c : \mathtt{chan} \vdash P : \mathtt{thread} \tag{8.8.6}$$

Applying **TC-Conf** to (8.8.6), (8.8.4) and (8.8.5), we obtain:

$$\Gamma;\Delta, c : \mathtt{chan} \vdash P,\sigma,\mathtt{CT} : \mathtt{conf} \tag{8.8.7}$$

To complete the case, apply **TC-ResC** to obtain (8.8.2) as required.

(2) An interesting case is $P_1 \,|\, (P_2 \,|\, P_3) \equiv (P_1 \,|\, P_2) \,|\, P_3$. The work associated with proving this case is handled by Lemma 8.7 as follows. Suppose:

$$\Gamma;\Delta_1 \odot (\Delta_2 \odot \Delta_3) \vdash P_1 \,|\, (P_2 \,|\, P_3) : \mathtt{thread} \qquad \text{with } \Delta_1 \asymp (\Delta_2 \odot \Delta_3) \tag{8.8.8}$$

We shall prove:

$$\Gamma;\Delta_1 \odot (\Delta_2 \odot \Delta_3) \vdash (P_1 \,|\, P_2) \,|\, P_3 : \mathtt{thread} \tag{8.8.9}$$

To derive (8.8.8), rule **TT-Par** must have been applied with premises:

$$\Gamma;\Delta_1 \vdash P_1 : \mathtt{thread} \tag{8.8.10}$$
$$\Gamma;\Delta_2 \odot \Delta_3 \vdash P_2 \,|\, P_3 : \mathtt{thread} \qquad \text{where } \Delta_2 \asymp \Delta_3 \tag{8.8.11}$$

By the side conditions of (8.8.8) and (8.8.11) we can apply Lemma 8.7(1) to obtain

$$\Delta_1 \asymp \Delta_2 \text{ and } (\Delta_1 \odot \Delta_2) \asymp \Delta_3 \tag{8.8.12}$$

To derive (8.8.11), **TT-Par** was also used, with premises:

$$\Gamma;\Delta_2 \vdash P_2 : \mathtt{thread} \tag{8.8.13}$$
$$\Gamma;\Delta_3 \vdash P_3 : \mathtt{thread} \tag{8.8.14}$$

By (8.8.12), we can apply **TT-Par** to (8.8.10) and (8.8.13) to yield:

$$\Gamma;\Delta_1 \odot \Delta_2 \vdash P_1 \,|\, P_2 : \texttt{thread} \tag{8.8.15}$$

Again by (8.8.12), we can take and (8.8.14) as the premises to rule **TT-Par** to obtain:

$$\Gamma;(\Delta_1 \odot \Delta_2) \odot \Delta_3 \vdash (P_1 \,|\, P_2) \,|\, P_3 : \texttt{thread} \tag{8.8.16}$$

Since (8.8.12), we can apply Lemma 8.7(2) to obtain (8.8.9) as required.

(3) An interesting case is $N \equiv N \,|\, \mathbf{0}$. Below we prove the right to left direction. The other direction is similar. Suppose:

$$\Gamma;\Delta_1 \odot \Delta_2 \vdash N \,|\, \mathbf{0} : \texttt{net} \qquad \text{with } \Delta_1 \asymp \Delta_2 \tag{8.8.17}$$

We shall prove:

$$\Gamma;\Delta_1 \odot \Delta_2 \vdash N : \texttt{net} \tag{8.8.18}$$

To infer (8.8.17), rule **TN-Par** must have been applied with the premises:

$$\Gamma;\Delta_1 \vdash N : \texttt{net} \tag{8.8.19}$$

$$\Gamma;\Delta_2 \vdash \mathbf{0} : \texttt{net} \tag{8.8.20}$$

Then, by Lemma 8.2(3) we have:

$$\Delta_2 = \{\vec{c} : \vec{\texttt{chan}}\} \tag{8.8.21}$$

$$\Delta_1 \odot \Delta_2 = \Delta_1 \cup \Delta_2 \qquad \text{with } \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset \tag{8.8.22}$$

By (8.8.22), there are no overlapping channels between Δ_1 and Δ_2. Therefore we can apply Lemma 8.2 to (8.8.19) to obtain (8.8.18) as required. □

Lemma 8.9 (Substitution).

1. *Assume* $\Gamma, x : T \vdash e : \alpha$ *and* $\Gamma \vdash v : T'$ *with* $x \notin \mathsf{fav}(e)$ *and* $T' <: T$*. Then we have* $\Gamma \vdash e[v/x] : \alpha'$ *for some* $\alpha' <: \alpha$*.*
2. $\Gamma, \texttt{this} : C \vdash e : \alpha$ *and* $\Gamma \vdash o : C'$ *with* $C' <: C$ *imply* $\Gamma \vdash e[o/\texttt{this}] : \alpha'$ *for some* $\alpha' <: \alpha$*.*

Proof. By induction on the structure of the expression e using Lemma 8.2 and Lemma 8.8.

(1) Below we will show the most interesting two cases.

Case $e \stackrel{\text{def}}{=} \texttt{freeze}[t](e_0)$*:* Suppose $\Gamma, x : T \vdash \texttt{freeze}[t](e_0) : \texttt{thunk}(U)$. Then this is derived from **TE-Freeze** with premises: $\Gamma, x : T \vdash e_0 : U$ with $\mathsf{fav}(e_0) = \emptyset$ and $\Gamma, x : T \vdash u_i : T_i$ with $\{\vec{u}\} = \mathsf{fnv}(e_0)$ and $\neg\mathsf{local}(T_i)$. Since $\mathsf{fav}(e_0) = \emptyset$, we can apply the inductive hypothesis to the former premise to obtain $\Gamma \vdash e_0[v/x] : U'$ for some $U' <: U$. To apply the rule **TE-Freeze** to $e_0[v/x]$, we have to have $\mathsf{fav}(e_0[v/x]) = \emptyset$ and $\Gamma \vdash u'_i : T'_i$ with $\{\vec{u}'\} = \mathsf{fnv}(e_0[v/x])$ and $\neg\mathsf{local}(T'_i)$. Note that $\mathsf{fv}(v) = \emptyset$ implies $\mathsf{fav}(e_0[v/x]) = \emptyset$ and $\vec{u}' = \vec{u} \setminus \{x\}$. Hence by the premise, we have $\neg\mathsf{local}(u'_i)$. Now we can apply **TE-Freeze** to $e_0[v/x]$ in order to obtain $\Gamma \vdash \texttt{freeze}[t](e_0[v/x]) : \texttt{thunk}(U')$. Since $\texttt{thunk}(U') <: \texttt{thunk}(U)$, this completes the case.

Case $e \stackrel{\text{def}}{=} E[\texttt{await}\ c]$*:* Suppose $\Gamma, x : T; \Delta, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c]^U : \texttt{thread}$. Then this is derived from **TT-Await** with the premise: $\Gamma, x : T; \Delta \vdash E[\ ]^U : \texttt{thread}$ with $c \notin \mathsf{dom}(\Delta)$. By assumption, $\mathsf{fav}(E[\texttt{await}\ c]^U) = \emptyset$, so by definition, $\mathsf{fav}(e) = \mathsf{fav}(E[\ ]^U) = \emptyset$. Therefore we can apply the inductive hypothesis obtaining $\Gamma; \Delta \vdash E[\ ]^U[v/x] : \texttt{thread}$. Since Δ is unchanged, the side condition $c \notin \mathsf{dom}(\Delta)$ still holds, and we can apply rule **TT-Await** to yield $\Gamma; \Delta, c : \texttt{chanI}(U) \vdash E[\texttt{await}\ c]^U[v/x] : \texttt{thread}$, as required.

(2) Most cases are trivially similar to those of (1). The key difference is in the base case, where $e \stackrel{\text{def}}{=} \texttt{this}$. However the proof remains straightforward. □

9 Network Invariants

This section studies the network invariants. We show that, if an initial network satisfies the initial conditions then well-typed reduction will preserve run-time invariants. These are important in showing safety and establishing the subject reduction theorem.

9.1 Network invariants and initial networks

We start from the definition of a property over networks, given in Definition 9.1.

Definition 9.1 (Properties). Let ψ denote a property over networks (i.e. ψ is a subset of networks). We write $N \models \psi$ if N satisfies ψ (i.e. if $N \in \psi$); we also write $N \not\models \psi$ if N does not satisfy ψ. We define the error property Err as the set of the networks which contain $\texttt{Error}$ as subexpression, i.e. $\mathsf{Err} = \{N \mid N \equiv (\nu\,\vec{u})(l[E[\texttt{Error}] \,|\, P, \sigma, \texttt{CT}] \,|\, N')\}$. We say ψ is *reduction closed*, if, whenever $N \models \psi$ and $N \longrightarrow\!\!\!\!\rightarrow N'$ such that $N' \not\models \mathsf{Err}$, we have $N' \models \psi$. We define ψ is a *network invariant with an initial property* ψ_0 if $\psi = \{N \mid \exists N_0.(N_0 \models \psi_0,\ N_0 \longrightarrow\!\!\!\!\rightarrow N,\ N \not\models \mathsf{Err})\}$ and, moreover, ψ is reduction-closed.

In order to ensure the correct execution of networks and the preservation of safety, we require certain properties to remain invariant. One of the key invariants in the presence of distribution of classes is that, when a class is *actually* called, 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 in l's class table, if its superclass D is not then this is also an unexpected state. This property is formalised by the completeness of the class table $\mathsf{comp}(C, \texttt{CT})$ defined in Definition 4.13.

The following definition formally states the above class invariant and others. Below $\mathsf{dom_v}(\sigma)$ (resp. $\mathsf{dom_o}(\sigma)$) denotes variables (resp. o-ids) of the domain of σ. Also we say *thread P inputs at c* if $P \equiv E[\texttt{await}\ c] \,|\, R$ for some E and R; dually *thread P outputs at c* if $P \equiv R \,|\, Q$ with $R \equiv \texttt{return}(c)\ e$ or $R \equiv [\texttt{go}]\ e\ \texttt{with/to}\ c$ for some Q and e.

Definition 9.2 (Network invariants). Given network $N \equiv (\nu\,\vec{u})(\prod_{0 \le i < n} l_i[F_i])$ with $F_i = (P_i, \sigma_i, \texttt{CT}_i)$, 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 $1 \le r \le 18$) as defined as follows:

Class invariants

1. $\texttt{FCT} \subseteq \texttt{CT}_i$
2. $P_i \equiv E[\texttt{new}\, C(\vec{v})] \,|\, Q_i \implies \mathsf{comp}(C, \texttt{CT}_i)$
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 $\mathsf{fcl}(\texttt{CT}_i(C)) = \{\vec{D}\}$

Value invariants

4. $P_i \equiv E[v] \,|\, P_i'$ then $\mathsf{fv}(v) = \emptyset$
5. $\sigma_i(x) = v \implies \mathsf{fv}(v) = \emptyset$
6. $\sigma_i(o) = (C, \vec{f} : \vec{v}) \implies \mathsf{fv}(v_i) = \emptyset$

State invariants

7. $\mathsf{fv}(P_i) \subseteq \mathsf{dom}_{\mathsf{v}}(\sigma_i) \subseteq \{\vec{u}\}$
8. $\mathsf{fv}(P_i) \cap \mathsf{fv}(P_j) = \emptyset$
9. $\mathsf{dom}_{\mathsf{v}}(\sigma_i) \cap \mathsf{dom}_{\mathsf{v}}(\sigma_j) = \emptyset$

Object identifier invariants

10. $o \in \mathsf{fn}(F_i) \cap \mathsf{fn}(F_j) \implies \exists! k.\ \sigma_k(o) = (C, ..) \wedge \mathsf{remote}(C)$
11. $o \in \mathsf{fn}(F_i) \wedge \exists k.\ \sigma_k(o) = (C, ..) \wedge \mathsf{local}(C) \implies k = i$
12. $o \in \mathsf{fn}(F_i) \implies \exists k\ 1 \leq k \leq n.\ o \in \mathsf{dom}_{\mathsf{o}}(\sigma_k)$
13. $\mathsf{dom}_{\mathsf{o}}(\sigma_i) \cap \mathsf{dom}_{\mathsf{o}}(\sigma_j) = \emptyset$

Method-invocation invariants

14. $P_i \equiv o.\mathtt{m}(\vec{v})\ \texttt{with}\ c \,|\, Q_i \implies \sigma_i(o) = (C, ..) \wedge \mathsf{comp}(C, \texttt{CT}_i)$

Field invariants

15. $P_i \equiv E[o.\mathtt{f}] \,|\, Q_i \implies \sigma_i(o) = (C, ..) \wedge \mathsf{comp}(C, \texttt{CT}_i)$.
16. $P_i \equiv E[o.\mathtt{f} := v] \,|\, Q_i \implies \sigma_i(o) = (C, ..) \wedge \mathsf{comp}(C, \texttt{CT}_i)$.

Linearity invariants

17. $P_i \equiv Q_i \,|\, R_i$ and Q_i inputs at $c \implies$ neither R_i nor P_j inputs at c.
18. $P_i \equiv Q_i \,|\, R_i$ and and Q_i outputs at $c \implies$ neither R_i nor P_j outputs at c.

Each invariant has a clear operational (and arguably engineering) meaning, as we illustrate below (each number corresponds to the invariant of the same number above).

Class invariants

1. (*Availability of foundation classes*) The class table at every location must contain a copy of the foundation classes.
2. (*Class availability*) Any class instantiated at a location as well as its all superclasses must be available in that location's class table.
3. (*Class name coherence*) Two classes with the same name distributed in the two different locations must have (1) the exactly same definition or (2) the same definition up to location tagging for that class name.

Value invariants

4. (*Value-closedness*) Values must not contain unbound variables.
5. (*Value-closedness in stores*) Store entries may not contain open variables.
6. (*Field value-closedness in stores*) Objects in a location's store must have closed fields.

State invariants

7. (*Availability of variable stores*) Programs at a location i must not mention variables that are not in σ_i. Also, all variables must be accounted for in the list of restricted names $\vec{u}$ for the entire network.

8. (*Locality of variables*) Local variables should not be shared between threads at different locations.
9. (*Locality of variable stores*) Local variables in the store should have globally unique names.

Object identifier invariants

10. (*Locality of local object ids*) If an object is referenced by threads in two different locations, it must be the case that the identifier is of a remote class.
11. (*Availability of local object ids*) If a thread at i has a reference to an instance of a local class, that object must necessarily be co-located in the store at i.
12. (*Availability of object ids*) An object must be located in the store at some remote or local location k.
13. (*Unicity of o-id stores*) Every object identifier in the system has a unique store. This ensures that there can be no ambiguity when determining the location that holds the store entry of a particular remote object reference.

Method-invocation invariants

14. (*Availability of a store and classes for method calls*) Any thread attempting to perform a method call will have the store and the code for that method body available.

Field invariants

15. (*Field access availability and locality*) Any field access always succeeds and will be made on objects local to the executing code.[1]
16. (*Field assignment availability and locality*) Any field assignment always succeeds and will be made to objects local to the executing code.

Linearity invariants

17. (*Unicity of await*) The network has a unique await for each name. This ensures that there can be no ambiguity when determining the place that a (remote) method invocation happened.
18. (*Unicity of output*) The network has a unique output thread for each name. This ensures that there is a unique return value for each method invocation.

Before proving the network invariants, 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 9.3 states these conditions formally.

Definition 9.3 (Initial network). We call network $N \equiv (\nu\,\vec{u})(\prod_{0\leq i<n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ *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 e in $\mathtt{freeze}[t](e)$ 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:

  Class invariants

  2′. (a) $\mathsf{fcl}(P_i) \subseteq \mathsf{dom}(\mathtt{CT}_i)$,
  (b) $C \in \mathsf{fcl}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_i) \implies \mathsf{comp}(C, \mathtt{CT}_i)$ and
  (c) $\sigma_i(o) = (C, \ldots) \implies \mathsf{comp}(C, \mathtt{CT}_i)$.

[1] Note that $o \in \mathsf{fn}(P_i)$ by the form of $E[\,]$. Similarly for $\mathsf{Inv}(16)$, $\mathsf{Inv}(15')$ and $\mathsf{Prog}(4)$–$\mathsf{Prog}(7)$.

– We also strengthen the field invariants $\mathsf{Inv}(15)$ and $\mathsf{Inv}(16)$ by replacing by:

Field invariants

$15'$. Let $\mathscr{C}$ be an arbitrary context. Then $P_i \equiv \mathscr{C}[o.\mathtt{f}] \implies o \in \mathsf{dom}(\sigma_i)$.

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

Note that, by combining $\mathsf{Inv}(2')$, the above condition subsumes $\mathsf{Inv}(15)$ and $\mathsf{Inv}(16)$.

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

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

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

We also note that during runs of programs, the initial properties may *not* be satisfied since classes can be downloaded lazily. A typical example is

$$(\mathtt{new}\,F^l(), \sigma, \mathtt{CT}) \longrightarrow (\mathtt{download}\,F\;\mathtt{from}\,l\;\mathtt{in}\;\mathtt{new}\,F(), \sigma, \mathtt{CT})$$

where F appears free in the r.l.s. expression, but $F \notin \mathsf{dom}(\mathtt{CT})$, hence it does not satisfy (b). Later we formalise this situation in Lemma 9.4 and prove the invariant $\mathsf{Inv}(2)$. The initial condition of $\mathsf{Inv}(15)$ is similarly understood as (c).

Proof method for invariants Some of the invariant properties are required for proofs of the subject reduction shown in the next section. This means type preservation must not be assumed to prove the invariant of $\mathsf{Inv}(r)$. Therefore for the inductive step goes through, we need to divide the proof routine into the following three steps:

Step 1 We prove one step invariant property for a typed network starting from the initial properties. This step has two sub-cases:

(i) *Assume* $\Gamma;\Delta \vdash N_0 : \mathtt{net}$ *and* N_0 *satisfies the initial properties. Then* $N_0 \longrightarrow N_1$ *implies* $N_1 \models \mathsf{Inv}(r)$ *for each* $1 \leq r \leq 18$ *if* $N_1 \not\models \mathsf{Err}$.

(ii) *Assume* $\Gamma;\Delta \vdash N_m : \mathtt{net}$ $(m \geq 1)$ *and* $N_m \models \mathsf{Inv}(r)$ *for all* $1 \leq r \leq 18$. *Then* $N_m \longrightarrow N_{m+1}$ *implies* $N_{m+1} \models \mathsf{Inv}(r)$ *for each* $1 \leq r \leq 18$ *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 following lemma lists key additional invariants related to dynamic downloading of classes, which are used for the main proofs of class invariants. We shall use the notation P_{im} to denote the threads at location i after m reduction steps.

Lemma 9.4 (Class invariants). *Assume* $\Gamma;\Delta \vdash N_k : \mathtt{net}$ *(*$0 \le k \le m$*) and* N_0 *satisfies the initial condition and* $N_k \in \mathsf{Inv}(r)$ *for* $1 \le r \le 18$ *(when* $k \ge 1$*). Assume* $N_0 \longrightarrow N_1 \longrightarrow N_2 \longrightarrow \cdots \longrightarrow N_{m-1} \longrightarrow N_m \equiv (\nu\, \vec{u_m})(\prod_{0\le i<n} l_i[P_{im}, \sigma_{im}, \mathtt{CT}_{im}]) \longrightarrow N_{m+1} \equiv (\nu\, \vec{u}_{m+1})(\prod_{0\le i<n} l_i[P_{im+1}, \sigma_{im+1}, \mathtt{CT}_{im+1}])$ *with (*$m \ge 1$*). Assuming* $0 \le j < n$ *where required then we have:*

1. *(Monotonicity of class tables)* $\mathtt{CT}_{im} \subseteq \mathtt{CT}_{im+1}$.
2. *(Eager thunks and class tables)*
   $P_{im+1} \equiv E[\ulcorner e\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ l_j \urcorner] \,|\, Q_{im+1} \implies \mathtt{CT}' \subseteq \mathtt{CT}_{jm+1}$.
3. *(Remote downloading of class tables)*
   $P_{im+1} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im+1} \implies \{\vec{C}\} \subseteq \mathsf{dom}(\mathtt{CT}_{jm+1})$ *and;*
   $P_{im+1} \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im+1} \implies \{\vec{C}\} \subseteq \mathsf{dom}(\mathtt{CT}_{jm+1})$.
4. *(Class downloading and availability of class tables)*
   $C \in \mathsf{fcl}(P_{im+1}) \implies (C \in \mathsf{dom}(\mathtt{CT}_{im+1}) \vee$
   $P_{im+1} \equiv E[\mathtt{download}\ C\ \mathtt{from}\ l_j\ \mathtt{in}\ \mathtt{new}\ C(\vec{v})] \,|\, Q_{im+1})$.
5. *(Availability of local classes)*
   $o \in \mathsf{fn}(P_{im+1}) \wedge \sigma_{im+1}(o) = (C, \dots)$
   $\implies ((C \in \mathsf{dom}(\mathtt{CT}_{im+1}) \wedge \forall D\ C <: D. D \in \mathsf{dom}(\mathtt{CT}_{im+1}))$
   $\vee P_{im+1} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ o] \,|\, Q_{im+1}$
   $\vee P_{im+1} \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ o] \,|\, Q_{im+1})$.
6. *(Availability of superclass tables)* $\exists N_k \equiv (\nu\, \vec{u}_k)(\prod_{0\le i<n} l_i[P_{ik}, \sigma_{ik}, \mathtt{CT}_{ik}])$ *with* $P_{ik} \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{ik}$ *and* $0 \le j < n$*. Then for each* $C_z \in \{\vec{C}\}$ *we have* $\forall C'\ C_z <: C'. \exists N_m \equiv (\nu\, \vec{u}_m)(\prod_{0\le i<n} l_i[P_{im}, \sigma_{im}, \mathtt{CT}_{im}])$ *such that* $N_k \longrightarrow\!\!\!\!\rightarrow N_m$ *and* $P_{im} \equiv E[\mathtt{resolve}\ \vec{D}\ \mathtt{from}\ l_j\ \mathtt{in}\ e] \,|\, Q_{im}$ *with* $C' \in \{\vec{D}\}$ *and* $C' \in \mathsf{dom}(\mathtt{CT}_{im})$.

Remark: The final statement requires slightly different conditions from others as we need to track a sequence of download-and-resolve reductions.

Proof. Induction on k. Below we omit σ and/or $\mathtt{CT}$ from the reduction step if they are not used for it.

(1) Straightforward by examining **RN-Download** and **RC-Defrost** since class tables are only changeable by either rule.

(2),(3) Obvious by investigating **RC-NewR**, **RN-Download** and **RC-Defrost** together with the inductive hypothesis N_k satisfies (1) monotonicity of the class table.

(4) For the base case, where $k = 0$ we have that by $\mathsf{Inv}(2')$ if $C \in \mathsf{fcl}(P_{i0})$ then it must be the case that $C \in \mathsf{dom}(\mathtt{CT}_{i0})$. If $C \notin \mathsf{fcl}(P_{i0})$ then the property holds trivially. For the inductive step, where $k = m$, we can assume that the property holds after m reduction steps. There are three sub-cases to considers:

(a) $C \in \mathsf{fcl}(P_{im})$ and $C \in \mathsf{dom}(\mathtt{CT}_{im})$. Then by (1) we have $C \in \mathsf{dom}(\mathtt{CT}_{im+1})$ as required.
(b) $C \in \mathsf{fcl}(P_{im})$ and $P_{im} \equiv E[\mathtt{download}\ C\ \mathtt{from}\ l_j\ \mathtt{in}\ \mathtt{new}\ C(\vec{v})] \,|\, Q_{im}$. By (3) we have $C \in \mathsf{dom}(\mathtt{CT}_{jm})$. By rule **RN-Download** we have that $\{\vec{D}\} = \{C\} \setminus \mathsf{dom}(\mathtt{CT}_{im})$. Assume $C \notin \mathsf{dom}(\mathtt{CT}_{im})$ then we have $C \in \{\vec{D}\}$ and so $C \in \mathsf{dom}(\mathtt{CT}')$ with $\mathtt{CT}' \subseteq \mathtt{CT}_{jm}$. By definition of downloading, $\mathtt{CT}_{im+1} = \mathtt{CT}_{im} \cup \mathtt{CT}'$ and therefore $C \in \mathsf{dom}(\mathtt{CT}_{im+1})$ as required.

(c) Assume $C \notin \mathsf{fcl}(P_{im})$, but $C \in \mathsf{fcl}(P_{im+1})$. Examining the definition of free class names and the reduction rules, we see that the only reduction that generates a new free class name is **RC-NewR**. Therefore we can deduce that:

$$P_{im} \equiv E[\texttt{new}\, C^{l_j}(\vec{v})] \,|\, Q_{im} \longrightarrow_{l_i} P_{im+1} \equiv E[\texttt{download}\, C\, \texttt{from}\, l_j\, \texttt{in}\, \texttt{new}\, C(\vec{v})] \,|\, Q_{im+1}$$

as required.

(5) Suppose $o \in \mathsf{fn}(P_{im+1}) \wedge \sigma_{im+1}(o) = (C, \dots)$. We have three situations.

(a) If o is created locally by evaluation of $\texttt{new}\ C(\vec{v})$ from the original program text, then by Lemma 9.4(1) and $\mathsf{Inv}(2)$ we have that $\forall D\ C <: D.D \in \mathsf{dom}(\texttt{CT}_{im+1})$.

(b) If o is created by the deserialisation of an object, there exists some k such that $P_{ik} \equiv E[\texttt{download}\ \vec{F}\ \texttt{from}\ l_j\ \texttt{in}\ o] \,|\, Q_{ik}$ where $\vec{F}$ contains all the classes of the objects in σ' as defined in **RN-Download**. Then by Lemma 9.4(6) we have that $\forall D\ C <: D.D \in \mathsf{dom}(\texttt{CT}_{il})$ for some $k < l \leq m$ as required. By Lemma 9.4(1) the multi-step reduction $N_l \longrightarrow\!\!\!\!\!\rightarrow N_m$ cannot have removed class table entries, and so $\forall D\ C <: D.D \in \mathsf{dom}(\texttt{CT}_{im+1})$ as required.

(c) The case (b) reduces to $E[\texttt{resolve}\ \vec{D}\ \texttt{from}\ l_j\ \texttt{in}\ o] \,|\, Q_{ik}$ by **RC-Resolve**. The reasoning is the same as (b) by using Lemma 9.4(6).

(6) Obvious by repeating **RN-Download** and **RC-Resolve** until we reach to resolve C'. Note that these reductions terminates as the inheritance relations in a well-formed class table is acyclic. □

9.2 Proofs of the network invariants

Now we are ready to prove **Step 1** using Lemma 9.4. It is only necessary to show sub-case (ii) of this step for each invariant, $\mathsf{Inv}(2)$, $\mathsf{Inv}(15)$ and $\mathsf{Inv}(16)$ excepted. For the cases of $\mathsf{Inv}(15)$ and $\mathsf{Inv}(16)$, we prove the stronger initial condition $\mathsf{Inv}(15')$ always hold. Hence $\mathsf{Inv}(15)$ and $\mathsf{Inv}(16)$ are derived as a corollary. On the other hand, in the case of $\mathsf{Inv}(2)$, we cannot assume that this particular invariant holds — it may be the case that the previous network was N_0 where $\mathsf{Inv}(2')$ holds instead. However, we must still show that $\mathsf{Inv}(2)$ is established by the reduction step.

Case $\mathsf{Inv}(1)$*:* By Lemma 9.4(1).

Case $\mathsf{Inv}(2)$*:* Suppose $P_{im} \not\equiv E[\texttt{new}\ C(\vec{v})] \,|\, Q_{im} \longrightarrow_{l_i} P_{im+1} \equiv E[\texttt{new}\ C(\vec{v})] \,|\, Q_{im}$.
There are two candidates for the reduction rule applied above:

(a) Suppose the rule applied was **RC-Cong**. Therefore $P_{im} \equiv E[\texttt{new}\ C(\vec{v}', e)] \,|\, Q_{im}$. This structure indicates that this expression was part of the original program text and so by Lemma 9.4(4) we have that $C \in \mathsf{dom}(\texttt{CT}_{im})$ and so $C \in \mathsf{dom}(\texttt{CT}_{i0})$. By $\mathsf{Inv}(2')$ we have that $\forall D\ C <: D.D \in \mathsf{dom}(\texttt{CT}_{i0})$ and so all super-classes of C must be present in $\texttt{CT}_{i0}$. Then by Lemma 9.4(1) we have that $\forall D\ C <: D.D \in \mathsf{dom}(\texttt{CT}_{im+1})$ as required.

(b) Suppose that the rule applied was **RC-DownloadNothing**. Therefore we set $P_{im} \equiv E[\texttt{download}\ C\ \texttt{from}\ l_j\ \texttt{in}\ \texttt{new}\ C(\vec{v})] \,|\, Q_{im}$. This means there is a sequence of download-resolve-download steps. Considering this pattern of behaviour, we can conclude that the chain must have started by the application of rule **RC-NewR**. By Lemma 9.4(6) we have that all super-classes of C must have been downloaded into $\texttt{CT}_{im}$. Therefore by Lemma 9.4(1) these classes are also in $\texttt{CT}_{im+1}$ as required.

Case $\mathsf{Inv}(3)$*:* We prove by the rule induction of $\longrightarrow$. We have two sub-cases.

(a) The last applied rule is **RN-Download**. Assume, with $\{\vec{D}\} = \{\vec{C}\} \setminus \mathsf{dom}(\mathtt{CT}_{im})$ and $\mathtt{CT}' = \mathtt{CT}_{jm}(\vec{D})$ we have:

$$l_i[E[\mathtt{download}\,\vec{C}\,\mathtt{from}\,l_j\,\mathtt{in}\,e]\,|\,P,\sigma_{im},\mathtt{CT}_{im}]\,|\,l_j[P_{jm},\sigma_{jm},\mathtt{CT}_{jm}] \longrightarrow$$
$$l_i[E[\mathtt{resolve}\,\vec{C}\,\mathtt{from}\,l_j\,\mathtt{in}\,e]\,|\,P,\sigma_{im+1},\mathtt{CT}_{im}\cup\mathtt{CT}']\,|\,l_j[P_{jm+1},\sigma_{jm+1},\mathtt{CT}_{jm+1}]$$

First we note that, by the same reasoning as the proof of Lemma 9.4(4), $\mathtt{CT}'$ is well-defined. Here we have to show $C \in \mathsf{dom}(\mathtt{CT}_{im}\cup\mathtt{CT}')\cap\mathsf{dom}(\mathtt{CT}_{jm})$ implies $(\mathtt{CT}_{im}\cup\mathtt{CT}')(C) = \mathtt{CT}_{jm}(C)$. However by $\mathtt{CT}' \subseteq \mathtt{CT}_{jm}$ and the inductive hypothesis such that $\mathtt{CT}_{im}(C) = \mathtt{CT}_{jm}(C)$, it is obvious that $(\mathtt{CT}_{im}\cup\mathtt{CT}')(C) = \mathtt{CT}_{jm}(C)$, concluding the case.

(b) The last applied rule is **RC-Defrost**. Similar with the above with Lemma 9.4(2).

Case $\mathsf{Inv}(4)$*:* The only interesting cases are when values are newly created by the reduction. Hence we only have to investigate **RC-Var**, **RC-Fld**, **RC-Ass**, **Serialize**, **RC-Freeze** and **RC-Defrost**. Cases **RC-Var** and **RC-Fld** are proved by the induction such that $N_m \in \mathsf{Inv}(5)$ and $N_m \in \mathsf{Inv}(6)$. The only interesting case is the last applied rule was **RC-Freeze**. Assume

$$\mathtt{freeze}[t](e),\sigma,\mathtt{CT} \longrightarrow_{l_i} \ulcorner e[\vec{v}/\vec{x}]\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ i\urcorner,\sigma,\mathtt{CT}$$

Since $\mathsf{fv}(v_i) = \emptyset$ by induction such that $N_m \in \mathsf{Inv}(5)$, we know, by the side condition of $\{\vec{x}\} = \mathsf{fv}(e)$, $\mathsf{fv}(e[\vec{v}/\vec{x}]) = \emptyset$. Hence $\mathsf{fv}(\ulcorner e[\vec{v}/\vec{x}]\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ i\urcorner) = \emptyset$. Other cases are straightforward by induction such that $N_m \in \mathsf{Inv}(4)$.

Case $\mathsf{Inv}(5)$*:* We only have to consider the rules where a store whose domain is a variable is modified, i.e. **RC-Dec**, **RC-Ass** and **RC-FldAss**. All are straightforward by induction.

Case $\mathsf{Inv}(6)$*:* We only have to consider rules where a store with an object id and field variables is modified, i.e. **RC-Fld**, **RC-FldAss**, **RC-FldAss** and **Deserialize**. The only interesting case is the last applied rule was **Deserialize**. Assume:

$$\begin{array}{c}\mathtt{deserialize}(\lambda\vec{o}.(\vec{v},\sigma',m)),\sigma_{im},\mathtt{CT}_{im}\\ \longrightarrow_{l_i} (\nu\,\vec{o})(\mathtt{download}\,C\,\mathtt{from}\,m\,\mathtt{in}\,\vec{v},\sigma\cup\sigma',\mathtt{CT}_{im+1})\end{array}$$

Without loss of generality, consider some $o \in \vec{v}$ such that $\sigma'(o) = (C,\dots)$ and $\vec{o} \notin \mathsf{dom}_{\mathsf{o}}(\sigma)$. Since the store fragment σ' is generated in some network N_k (where $k < m$) and $N_k \in \mathsf{Inv}(6)$ by induction. Therefore we see that adding this closed fragment to an already closed store σ_{im}, it must be the case that σ_{im+1} is also closed. Hence $N_{m+1} \in \mathsf{Inv}(6)$ as required.

Case $\mathsf{Inv}(7)$*:* We only have to check the last applied rule which changes a store.

(a) The last applied rule was **RC-Dec**. Suppose for some $0 \leq i < n$ we have $\Gamma;\Delta \vdash (\nu\,\vec{u}_m)l_i[E[T\ x := v\ ;\ e]\,|\,Q_{im},\sigma_{im},\mathtt{CT}_{im}] : \mathtt{net}$. Without loss of generality we can assume $E \equiv [\ ]$ and $Q_{im} \equiv \mathbf{0}$. Then after reduction, we have

$(\nu\, x\vec{u}_m)l[e, \sigma_{im} \cdot [x \mapsto v], \mathtt{CT}_{im+1}]$. We must show that $\mathsf{fv}(e) \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [x \mapsto v]) \subseteq \{x\vec{u}\}$. Since $\mathsf{Inv}(4)$ holds by assumption, then by definition $\mathsf{fv}(T\ x := v\ ;\ e) = \mathsf{fv}(e) \setminus \{x\}$. If $x \in \mathsf{fv}(e)$, then clearly $x \in \mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [x \mapsto v])$ by definition of the variable domain, and so we have $\mathsf{fv}(e) \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [x \mapsto v])$. If $x \notin \mathsf{fv}(e)$ then the argument is similar. The vector of restricted names, $\vec{u}_m$, is extended with x. Therefore we also have that $\mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [x \mapsto v]) \subseteq \{x\vec{u}_m\}$ as required.

(b) The last applied rule was **RC-Ass**. By assumption we have that
   $\Gamma;\Delta \vdash (\nu\,\vec{u}_m)l_i[E[x := v] \,|\, Q_{im}, \sigma_{im}, \mathtt{CT}_{im}] : \mathtt{net}$; and
   $(\nu\,\vec{u}_m)l_i[E[x := v] \,|\, Q_{im}, \sigma_{im}, \mathtt{CT}_{im}] \longrightarrow_{l_i} (\nu\,\vec{u}_{m+1})l_i[E[v] \,|\, Q_{im+1}, \sigma_{im+1}, \mathtt{CT}_{im+1}]$.
   Without loss of generality, let $Q_{im} \equiv \mathbf{0}$ and $E \equiv [\ ]$. We must show that $\mathsf{fv}(v) \subseteq \mathsf{dom}_\mathsf{v}(\sigma) \subseteq \{\vec{u}\}$. From $\mathsf{Inv}(4)$, $\mathsf{fv}(v) = \emptyset$ and so $\mathsf{fv}(x := v) = \{x\}$. By assumption $\{x\} \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im}) \subseteq \{\vec{u}_m\}$. After reduction, trivially $\mathsf{fv}(v) \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im}[x \mapsto v])$, and given that the vector of restricted names is preserved, $\mathsf{dom}_\mathsf{v}(\sigma_{im}[x \mapsto v]) \subseteq \{\vec{u}_{m+1}\}$.

(c) The last applied rule was **Deserialize**. This case is straightforward—although the store changes we know by Lemma 8.4 (the graph computation lemma) that the store appended contains no variables. Therefore $\mathsf{dom}_\mathsf{v}(\sigma_{im} \cup \sigma') = \mathsf{dom}_\mathsf{v}(\sigma_{im})$, and because $\mathsf{fv}(\mathtt{download}\ C\ \mathtt{from}\ m\ \mathtt{in}\ \vec{v}) = \emptyset$, we have $\emptyset \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im} \cup \sigma') \subseteq \{\vec{o}\vec{u}_m\}$ as required.

(d) The last applied rule was **RC-MethInvoke**. Again, without loss of generality take $Q_{im} \equiv \mathbf{0}$ and $E \equiv [\ ]$. Given $\Gamma;\Delta \vdash (\nu\,\vec{u}_m)l_i[o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c, \sigma_{im}, \mathtt{CT}_{im}](\stackrel{\text{def}}{=} N) : \mathtt{net}$ and $N \longrightarrow (\nu\,\vec{x}\vec{u}_m)l_i[e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}], \sigma_{im} \cdot [\vec{x} \mapsto \vec{v}], \mathtt{CT}_{im+1}]$, we must show that
   $\mathsf{fv}(e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}]) \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [\vec{x} \mapsto \vec{v}]) \subseteq \{\vec{x}\vec{u}_m\}$.
   Note that $\mathsf{fv}(e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}]) = \mathsf{fv}(e)$ by definition of substitution. Given $\mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [\vec{x} \mapsto \vec{v}]) = \mathsf{dom}_\mathsf{v}(\sigma_{im}) \cup \{\vec{x}\}$ and $\mathsf{dom}_\mathsf{v}(\sigma_{im}) \subseteq \{\vec{u}\}$ then we have that $\mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [\vec{x} \mapsto \vec{v}]) \subseteq \{\vec{x}\vec{u}_m\}$. By our assumption that the initial network is well-typed, we can conclude $\vdash \mathtt{CT}_{im} : \mathtt{ok}$. Since $(\vec{x}, e) = \mathsf{mbody}(\mathtt{m}, C, \mathtt{CT})$ for some C, we must have that $\mathsf{fv}(e) = \{\vec{x}\}$. Clearly $\{\vec{x}\} \subseteq \mathsf{dom}_\mathsf{v}(\sigma_{im} \cdot [\vec{x} \mapsto \vec{v}])$, finishing the case.

Case $\mathsf{Inv}(8)$*:* Since $N_m \in \mathsf{Inv}(5)$, we only have to investigate that the last applied rules where terms, values or classes are transferred across the different locations. Then there are two sub-cases.

(a) The last applied rule was **RN-Leave**. Suppose that $P_{im} \equiv \mathtt{go}\ o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, Q_{im}$. Then we have, for some $\Gamma' = \Delta, \vec{x} : \vec{T}$ and $\Delta' = \Delta, \vec{c} : \vec{\mathtt{chan}}$,
   (i) $\Gamma';\Delta' \vdash o : C$ and $\mathsf{remote}(C)$.
   (ii) (a) $v_i = o_i$ and $\Gamma';\Delta' \vdash o_i : C_i$ or (b) $\mathsf{fnv}(v_i) = \emptyset$.
   Note that (ii-b) uses the induction such that $N_m \in \mathsf{Inv}(4)$. Hence after applying **RN-Leave**, $\mathsf{fv}(v_i) = \emptyset$ still holds, completing $N_{m+1} \in \mathsf{Inv}(8)$.

(b) The last applied rule was **RN-Return**. Similar with the above sub-case.

Case $\mathsf{Inv}(9)$*:* By $\mathsf{Inv}(7)$, $\mathsf{Inv}(8)$ and the inductive hypothesis.

Case $\mathsf{Inv}(10)$*:* Again, as in the proof of $\mathsf{Inv}(8)$, we only have to consider the cases the last applied rule is either **RN-Leave** or **RN-Return**. But this is again derived from (i) and (ii-a) in the proof of $\mathsf{Inv}(8)$.

Case $\mathsf{Inv}(11)$*:* We only have to investigate the case that the last applied rule is **Deserialize**. It is mechanical by a similar reasoning with the case $\mathsf{Inv}(6)$.

Case $\mathsf{Inv}(12)$*:* Obvious since we cannot apply the garbage collection.

Case $\mathsf{Inv}(13)$*:* By $\mathsf{Inv}(10)$ and $\mathsf{Inv}(11)$.

Case $\mathsf{Inv}(14)$*:* Suppose $P_{im+1} \equiv o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, Q_{im+1}$. Then by **RC-MethInvoke**, we have $\sigma_{im+1}(o) = (C, \dots)$. Now we know $o \in \mathsf{fn}(P_{im+1})$ and so by Lemma 9.4(5) and the shape of P_{im+1}, we have $C \in \mathsf{dom}(\mathtt{CT}_{im+1}) \wedge \forall D\ C <: D.D \in \mathsf{dom}(\mathtt{CT}_{im+1})$, as required.

Case $\mathsf{Inv}(15')$*:* We only consider the case for the field access. The case for the field assignment is just the same. There are three cases:

(1) The case $P_{im} \equiv \mathscr{C}[o.\mathtt{f}]$ and $P_{im+1} \equiv \mathscr{C}'[o.\mathtt{f}]$. I.e. $\longrightarrow$ is applied for the context or other networks. This case is obvious by the assumption.

(2) The case $P_{im} \equiv E[e.\mathtt{f}] \,|\, Q_{im}$ and $P_{im+1} \equiv E[o.\mathtt{f}] \,|\, Q_{im}$ with $e \neq o$ and $e \neq \mathtt{this}$. The only interesting cases are the last applied rule is either **RC-Var**, **RC-Fld**, **RC-New** or **Deserialize**. We only show the cases of **RC-Var** and **RC-New**.

(a) Suppose the last applied rule is **RC-Var**. Then:

$$(E[x.\mathtt{f}] \,|\, Q_{im}, \sigma_{im}, \mathtt{CT}_{im}) \longrightarrow_{l_i} (E[o.\mathtt{f}] \,|\, Q_{im}, \sigma_{im}, \mathtt{CT}_{im})$$

with $\sigma_{im}(x) = o$. For $x.\mathtt{f}$ typable, by **TE-Fld**, we know $\Gamma \vdash x : C$ with $\mathsf{local}(C)$ and $\Gamma \vdash \sigma_{im} : \mathtt{ok}$ for some Γ. Since $[x \mapsto o] \in \sigma_{im}$ by **RC-Var**, we have $\Gamma \vdash o : C$ with $\mathsf{local}(C)$. Also by the indctive hypothesis, P_{im} satisfies $\mathsf{Inv}(11)$ and $\mathsf{Inv}(12)$. By $\mathsf{Inv}(12)$, there exists $1 \leq k \leq n$ such that $[o \mapsto (C, ..)] \in \sigma_{km}$. Then by $\mathsf{Inv}(11)$, $o \in \mathsf{fn}(\sigma_{im})$ implies $i = k$, which means $o \in \mathsf{dom}(\sigma_{im})$, as desired.

(b) Suppose the last applied rule is **RC-New**. Then:

$$\begin{aligned}&(E[\mathtt{new}\ C(\vec{v}).\mathtt{f}] \,|\, Q_{im}, \sigma_{im}, \mathtt{CT}_{im})\\ &\qquad\longrightarrow_{l_i} (\nu\, o)(E[o.\mathtt{f}] \,|\, Q_{im}, \sigma_{im} \cdot [o \mapsto (C, \vec{f} : \vec{v})], \mathtt{CT}_{im})\end{aligned}$$

Then obviously $o \in \mathsf{dom}(\sigma_{im})$. The case for **Deserialize** is similar.

(3) The last applied rule is **RC-MethInvoke**. I.e. we have:

$$\begin{aligned}&o'.\mathtt{m}(\vec{v})\ \mathtt{with}\ c, \sigma_{im}, \mathtt{CT}_{im}\\ &\qquad\longrightarrow_{l_i} (\nu\, \vec{x})(e[o'/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}], \sigma_{im} \cdot [\vec{x} \mapsto \vec{v}], \mathtt{CT}_{im})\end{aligned}$$

with $\sigma_{im}(o') = (C, \dots)$ and $\mathsf{mbody}(\mathtt{m}, C, \mathtt{CT}_{im}) = (\vec{x}, e)$. Since $\mathsf{fn}(e) = \emptyset$, $P_{im+1} \equiv \mathscr{C}[o.\mathtt{f}]$ implies $o' = o$ and $\mathtt{this}$ is substituted by o. This means $\sigma_{im}(o) = (C, \dots)$, as required.

Case $\mathsf{Inv}(17)$*,* $\mathsf{Inv}(18)$*:* Straightforward by the definition of $\Delta_1 \asymp \Delta_2$ and the analysis on **TT-Res**, **TT-Await**, **TT-Return** and **TT-GoSer**. □

We can derive the following progress properties immediately from the invariants. Note that the linear invariants also guarantee the determinacy of remote method invocation and return points, strengthening usual progress properties as found in $\mathsf{Prog}(10)$.

Definition 9.5 (Progress invariants). Given network $N \equiv (\nu\,\vec{u})(\prod_{0\leq i<n} l_i[P_i,\sigma_i,\mathtt{CT}_i])$, and assuming $0 \leq k < n$, we define property $\mathsf{Prog}(r)$ as a set which satisfy the following condition.

Class

1. $P_i \equiv E[\mathtt{new}\ C(\vec{v})]\,|\,Q_i \implies C \in \mathsf{dom}(\mathtt{CT}_i)$
2. $P_i \equiv E[\mathtt{download}\ \vec{C}\ \mathtt{from}\ l_k\ \mathtt{in}\ e]\,|\,Q_i \implies C_{ij} \in \mathsf{dom}(\mathtt{CT}_i) \cup \mathsf{dom}(\mathtt{CT}_k)$.
3. $P_i \equiv E[\mathtt{resolve}\ \vec{C}\ \mathtt{from}\ m\ \mathtt{in}\ e]\,|\,Q_i \implies C_{ij} \in \mathsf{dom}(\mathtt{CT}_i)$.

Field

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

Variable

6. $P_i \equiv E[x]\,|\,Q_i \implies x \in \mathsf{dom}(\sigma_i)$
7. $P_i \equiv E[x := v]\,|\,Q_i \implies x \in \mathsf{dom}(\sigma_i)$

Method-invocation

8. $P_i \equiv o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c\,|\,Q_i \wedge \sigma_i(o) = (C,\dots) \implies \mathsf{mbody}(\mathtt{m},C,\mathtt{CT}_i)$ defined.
9. $P_i \equiv \mathsf{go}\ o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c\,|\,Q_i \implies \exists! k.\ o \in \mathsf{dom}(\mathtt{CT}_k)$.

Return

10. $P_i \equiv \mathsf{go}\ v\ \mathtt{to}\ c\,|\,Q_i\ \wedge\ c \in \{\vec{u}\} \implies \exists! k.\ P_k \equiv E[\mathtt{await}\ c]\,|\,Q_k$.

We explain these properties briefly below:

Class

1. (*Class availability*) Classes are always available for instantiation.
2. (*Download locates required classes*) Download operations always succeed in retrieving the required classes from the specified location.
3. (*Resolution is coherent*) No attempt is made to resolve classes that are not available in the local class table.

Field

4. (*Field access availability and locality*) No attempt is made to invoke a field access on the store if the class of the store does not provide that field.
5. (*Field assignment availability and locality*) No attempt is made to invoke a field access on the store if the class of the store does not provide that field.

Variable

6. (*Variable access availability and locality*) Expressions only access variables they are local to.
7. (*Variable assignment availability and locality*) Expressions only assign to variables they are local to.

Method-invocation

8. (*Objects understand messages*) No attempt is made to invoke a method on an object of a given class if that class does not provide that method.
9. (*Remote invocations have a destination*) Remote method invocations always refer to a unique live location in the network.

Return

10. (*Linear return*) If a method return exists, there must be exactly one location waiting for it on that channel.

Proposition 9.6 (Progress). *Assume* $N_0 \models \mathsf{Init}$ *and* $N_0 \longrightarrow\!\!\!\!\rightarrow N_m$ *with* $\Gamma;\Delta \vdash N_k : \mathtt{net}$ *for all* $0 \le k \le m$*. Then* $N_m \longrightarrow N_{m+1}$ *implies* $N_{m+1} \models \mathsf{Prog}(r)$ *(*$1 \le r \le 10$*).*

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 **RN-Download**. $\mathsf{Prog}(4)$ and $\mathsf{Prog}(5)$ are proved by $\mathsf{Inv}(15)$ and by $\mathsf{Inv}(16)$, respectively. $\mathsf{Prog}(6)$ and $\mathsf{Prog}(7)$ are obvious by $\mathsf{Inv}(7)$. $\mathsf{Prog}(8)$ is derived from $\mathsf{Inv}(14)$. $\mathsf{Prog}(9)$ is by combining $\mathsf{Inv}(12)$ and $\mathsf{Inv}(13)$. $\mathsf{Prog}(10)$ is straightforward by combining $\mathsf{Inv}(17)$ and $\mathsf{Inv}(18)$. □

10 Type Soundness

This section proves the subject reduction theorem. As a corollary, we derive the network invariants and progress properties. There are three key points on the proof of the theorem, which are not found in those for the sequential languages [18, 4, 9]; first we directly use the network invariants (Definition 9.1 in Section 9) for the cases of code mobility (freeze and defrost), remote method invocations and field access; secondly we use the linearity of channels for the cases of parallel compositions of threads and networks; finally we use the correctness of class and object graphs (Lemmas 8.4 in § 8.3) to ensure the typability of thunks and serialised objects.

Following the proof method in Section 9, we assume **Step 1** in the paragraph **Proof method for invariants** to prove **Step 2** (i.e. Subject Reduction Theorems). We start from the expression.

Theorem 10.1 (Subject reduction for expressions). *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* $\vdash \mathtt{CT}' : \mathtt{ok}$*.*

Proof. By induction on the derivation $F \longrightarrow_l F'$ with a case analysis on the final typing rules. Proofs are laid out in the following manner: definition of the configuration F (before reduction), definition of the configuration F' after reduction and finally the premises and conditions that must have held for the reduction to take place. We omit to prove $\Gamma,\vec{u}':\vec{T}' \vdash \sigma' : \mathtt{ok}$ and/or $\vdash \mathtt{CT}' : \mathtt{ok}$ when the stores and/or class tables are unchanged during the reduction.

Case ***RC-Var****:*

$F \stackrel{\mathsf{def}}{=} x,\sigma,\mathtt{CT}$
$F' \stackrel{\mathsf{def}}{=} \sigma(x),\sigma,\mathtt{CT}$

From the shape of F, the last expression typing rule applied was **TE-Var**. This states that $\Gamma \vdash x : T$ and, with the assumption that $\Gamma \vdash \sigma : \mathtt{ok}$, we can immediately apply Lemma 8.3(3) to conclude $\Gamma \vdash \sigma(x) : T'$ for some $T' <: T$.

Case ***RC-Cond****:*

$F \stackrel{\mathsf{def}}{=} \mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2,\sigma,\mathtt{CT}$
$F' \stackrel{\mathsf{def}}{=} e_1,\sigma,\mathtt{CT}$

We consider only the case where the boolean test is `true`; the case for `false` is identical. From the structure of F, the last typing rule applied must have been **TE-Cond** with premise $\Gamma \vdash e_1 : S'$ and $S' <: S$. Hence $\Gamma \vdash e_1 : S$, as required.

*Case **RC-Fld**:*

$F \stackrel{\text{def}}{=} o.\mathtt{f_i}, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} v_i, \sigma, \mathtt{CT}$

First by assuming F satisfies $\mathsf{Prog}(4)$, we have $o \in \mathsf{dom}(\sigma)$. Then by the premise of **RC-Fld**, $\sigma(o) = (C, \vec{f} : \vec{v})$. Then the proof is straightforward by Lemma 8.3(6), the assumptions $\Gamma \vdash o.\mathtt{f_i} : T_i$ and $\Gamma \vdash \sigma : \mathtt{ok}$ with the side condition $\sigma(o) = (C, \vec{f} : \vec{v})$. We obtain the judgement $\Gamma \vdash v_i : T_i'$ for some $T_i' <: T_i$, completing the case.

*Case **RC-Seq**:*

$F \stackrel{\text{def}}{=} e_1; e_2, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, \vec{u})(e_2, \sigma', \mathtt{CT}')$

The premise of **RC-Seq** states that $e_1, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(v, \sigma', \mathtt{CT}')$ with $\vec{u} \notin \mathsf{fnv}(e_2)$. From structure of F, the last typing rule applied must have been **TE-Seq** with premises: $\Gamma \vdash e_1 : \mathtt{void}$ and $\Gamma \vdash e_2 : S$. By the side condition $\vec{u} \notin \mathsf{fnv}(e_2)$, we can apply Lemma 8.2(5) to obtain $\Gamma, \vec{u} : \vec{T} \vdash e_2 : S$. By the inductive hypothesis, we have $\Gamma \vdash \sigma' : \mathtt{ok}$ and $\vdash \mathtt{CT}' : \mathtt{ok}$, finishing the case.

*Case **RC-Dec**:*

$F \stackrel{\text{def}}{=} T\ x = v\ ;\ e, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, x)(e, \sigma \cdot [x \mapsto v], \mathtt{CT})$

By the premise of **RC-Dec** we have $x \notin \mathsf{dom_v}(\sigma)$. From the shape of F, the last typing rule applied was **TE-Dec**, with the premises: $\Gamma, x : T \vdash e : S$ and $\Gamma \vdash v : T'$ such that $T' <: T$. The latter gives us type preservation immediately. Then by assumption $\Gamma \vdash \sigma : \mathtt{ok}$ and by side condition $x \notin \mathsf{dom_v}(\sigma)$, we can apply Lemma 8.3(1) to obtain $\Gamma, x : T \vdash \sigma \cdot [x \mapsto v] : \mathtt{ok}$.

*Case **RC-Ass**:*

$F \stackrel{\text{def}}{=} x := v, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} v, \sigma[x \mapsto v], \mathtt{CT}$

Examining F, the last typing rule applied was **TE-Ass**, $\Gamma \vdash x := v : T'$, with premises $\Gamma \vdash v : T'$ and $\Gamma \vdash x : T$ such that $T' <: T$. By the former, type preservation is immediate. Also by applying Lemma 8.3(2) to these premises and assumption $\Gamma \vdash \sigma : \mathtt{ok}$, we can derive $\Gamma \vdash \sigma[x \mapsto v] : \mathtt{ok}$.

*Case **RC-FldAss**:*

$F \stackrel{\text{def}}{=} o.\mathtt{f_i} := v, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} v, \sigma[o \mapsto \sigma(o)[f_i \mapsto v]], \mathtt{CT}$

From the shape of F, the last typing rule applied to the expression was **TE-FldAss** giving judgement $\Gamma \vdash o.\mathtt{f_i} := v : T_i'$ with the premises: $\Gamma \vdash o.\mathtt{f_i} : T_i$ and $\Gamma \vdash v : T_i'$ with $T_i' <: T_i$. The latter immediately derives type preservation. Hence we only have to show that the new store is well-formed. In order to derive the former premise, the rule **TE-Fld** must have been used, with premises: $\Gamma \vdash o : C$ with $\mathsf{fields}(C) = \vec{T}\vec{f}$ and $\vdash C : \mathtt{tp}$. Note assuming F satisfies $\mathsf{Prog}(5)$, we have $o \in \mathsf{dom}(\sigma)$. Then together with assumptions $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash v : T_i'$ and by **RC-FldAss**, we can apply Lemma 8.3(5) to obtain $\Gamma \vdash \sigma[o \mapsto \sigma(o)[f_i \mapsto v]] : \mathtt{ok}$, finishing the case.

Case ***RC-New****:*

$F \stackrel{\text{def}}{=} \texttt{new}\ C^{\text{-}}(\vec{v}), \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, o)(o, \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})], \texttt{CT})$

The premise of **RC-New** states that $\mathsf{fields}(C) = \vec{T}\vec{f}$ and $C \notin \mathsf{dom}(\texttt{CT})$. By examining the structure of F, the last rule applied in the derivation $\Gamma \vdash \texttt{new}\ C^{\text{-}}(\vec{v}) : C$ was **TE-New**, with premises: $\vdash C : \texttt{tp}$ with $\mathsf{fields}(C) = \vec{T}\vec{f}$ and $\Gamma \vdash \vec{v} : \vec{T}'$ such that $T_i' <: T_i$. By this, **E-Oid** derives $\Gamma, o : C \vdash \texttt{Env}$. It is then possible to apply **TV-Oid** to derive $\Gamma, o : C \vdash o : C$, which shows type preservation. It remains to show that the new store is well-formed. Given the premises and assumption $\Gamma \vdash \sigma : \texttt{ok}$, we can apply Lemma 8.3(4) to obtain $\Gamma, o : C \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})] : \texttt{ok}$, as required.

Case ***RC-NewR****:*

$F \stackrel{\text{def}}{=} \texttt{new}\ C^{m}(\vec{v}), \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} \texttt{download}\ C\ \texttt{from}\ m\ \texttt{in}\ \texttt{new}\ C(\vec{v}), \sigma, \texttt{CT}$

By the premise of **RC-NewR**, we also have that $C \notin \mathsf{dom}(\texttt{CT})$. However, this case is straightforward: by examining the structure of F, the last rule applied was **TE-New**. This judgement is of the form $\Gamma \vdash \texttt{new}\ C^{m}(\vec{v}) : C$. Its premises can be used for the application of **TE-ClassLoad** to derive $\Gamma \vdash \texttt{download}\ C\ \texttt{from}\ m\ \texttt{in}\ \texttt{new}\ C(\vec{v}) : C$, concluding the case.

Case ***RC-Cong****:*

$F \stackrel{\text{def}}{=} E[e], \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, \vec{u})(E[e'], \sigma', \texttt{CT}')$

RC-Cong has side conditions $e, \sigma, \texttt{CT} \longrightarrow_l (\nu\, \vec{u})(e', \sigma', \texttt{CT}')$ and $\vec{u} \notin \mathsf{fnv}(E)$. Straightforward by Lemma 8.6.

Case ***Serialize****:*

$F \stackrel{\text{def}}{=} \texttt{serialize}(\vec{v}), \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} \lambda \vec{o}.(\vec{v}, \sigma', l), \sigma, \texttt{CT}$

By the premises of **Serialize** we also have that $\sigma' = \bigcup \mathsf{og}(\sigma, v_i)$ and $\{\vec{o}\} = \mathsf{dom}_{\mathsf{o}}(\sigma')$.

From the shape of F, and by the assumptions, we conclude that the last typing rule applied was **TE-Serialize**. This gives the judgement $\Gamma \vdash \texttt{serialize}(\vec{v}) : \texttt{ser}(\vec{U})$ with the condition $\Gamma \vdash \vec{v} : \vec{U}$. By this together with the assumptions $\Gamma \vdash \sigma : \texttt{ok}$ and $\sigma' = \bigcup \mathsf{og}(\sigma, v_i)$, we can apply Lemma 8.4(1) (soundness of object graph computation) to obtain $\Gamma \vdash \sigma' : \texttt{ok}$.

Set $\Gamma = \Gamma', \vec{o} : \vec{C}$. Then we have $\Gamma', \vec{o} : \vec{C} \vdash \sigma' : \texttt{ok}$ and $\Gamma', \vec{o} : \vec{C} \vdash \vec{v} : \vec{U}$. Note that, by Lemma 8.4(1), all object identifiers in store σ' must be instances of local classes, establishing $\mathsf{local}(C_i)$. Then **TV-Blob** gives us $\Gamma' \vdash \lambda \vec{o}.(\vec{v}, \sigma', l) : \texttt{ser}(\vec{U})$. Finally by the weakening lemma, we derive $\Gamma \vdash \lambda \vec{o}.(\vec{v}, \sigma', l) : \texttt{ser}(\vec{U})$ as desired.

Case ***Deserialize****:*

$F \stackrel{\text{def}}{=} \texttt{deserialize}(\lambda\vec{o}.(\vec{v},\sigma',m)),\sigma,\texttt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\,\vec{o})(\texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in}\ \vec{v},\sigma\cup\sigma',\texttt{CT})$

with $\{\vec{F}\} = \{C \mid \sigma'(o_i) = (C,\dots)\}$ and $\vec{o} \notin \mathsf{dom}(\sigma)$.

The structure of F shows that the last expression typing rule applied was **TE-Deserialize**, giving the judgement $\Gamma \vdash \texttt{deserialize}(\lambda\vec{o}.(\vec{v},\sigma',m)) : \vec{U}$. This is derived from the premise $\Gamma \vdash \lambda\vec{o}.(\vec{v},\sigma',m) : \texttt{ser}(\vec{U})$.

To infer this, **TV-Blob** must have been employed, with premises: $\Gamma,\vec{o}:\vec{C} \vdash \vec{v} : \vec{U}$, $\mathsf{local}(C_i), \Gamma,\vec{o}:\vec{C} \vdash \sigma : \texttt{ok}$ and $\{\vec{o}\} = \mathsf{dom}_{\mathsf{o}}(\sigma)$. By Lemma 8.2(9) (implied judgement), we have $\Gamma,\vec{o}:\vec{C} \vdash \texttt{Env}$, which is derived from $\Gamma \vdash o_i : C_i$ by **TV-Oid**. We can also check by $\vec{F} \subseteq \vec{C}$ and $\Gamma,\vec{o}:\vec{C} \vdash \sigma' : \texttt{ok}$ imply $\vdash F_i : \texttt{tp}$ by **S-Oid** and **TV-Oid**. Now the application of **TE-ClassLoad** leads to $\Gamma,\vec{o}:\vec{C} \vdash \texttt{download}\ \vec{F}\ \texttt{from}\ m\ \texttt{in}\ \vec{v} : \vec{U}$.

For the well-formedness of the store, we note $\Gamma \vdash \sigma : \texttt{ok}$ and $\{\vec{o}\} = \mathsf{dom}_{\mathsf{o}}(\sigma')$ imply $\mathsf{dom}_{\mathsf{o}}(\sigma) \cap \mathsf{dom}_{\mathsf{o}}(\sigma') = \emptyset$. Hence by Lemma 8.3(7), we can obtain $\Gamma,\vec{o}:\vec{C} \vdash \sigma\cup\sigma' : \texttt{ok}$, completing the case.

Case ***RC-Resolve****:*

$F \stackrel{\text{def}}{=} \texttt{resolve}\ \vec{C}\ \texttt{from}\ l'\ \texttt{in}\ e,\sigma,\texttt{CT}$
$F' \stackrel{\text{def}}{=} \texttt{download}\ \vec{D}\ \texttt{from}\ l'\ \texttt{in}\ e,\sigma,\texttt{CT}$

By premise of **RC-Resolve** we have $\texttt{CT}(C_i) = \texttt{class}\ C_i\ \texttt{extends}\ D_i\{\vec{T}\vec{f};K\vec{M}\}$ and $C \in \mathsf{dom}(\texttt{CT})$. Examining F, the last typing rule applied was **TE-ClassLoad**, $\Gamma \vdash \texttt{resolve}\ \vec{C}\ \texttt{from}\ m\ \texttt{in}\ e : \vec{U}$, with the conditions $\vdash \vec{C} : \texttt{tp}$, $\Gamma \vdash e : \vec{U}$ and $\vdash \texttt{CT} : \texttt{ok}$. To apply **TE-ClassLoad**, we have to show $\vdash \vec{D} : \texttt{tp}$, which is proved immediately by the premise of **Wf-Sig**.

Case ***RC-Freeze****:*

$F \stackrel{\text{def}}{=} \texttt{freeze}[\texttt{eager}](e),\sigma,\texttt{CT}$
$F' \stackrel{\text{def}}{=} \ulcorner e[\vec{v}/\vec{x}]\ \texttt{with}\ \texttt{CT}'\ \texttt{from}\ l\urcorner,\sigma,\texttt{CT}$

The side conditions of this rule state that $\{\vec{x}\} = \mathsf{fv}(e)$, $v_i = \sigma'(x_i)$ and $\texttt{CT}' = \mathsf{cg}(\texttt{CT},\mathsf{fcl}(e))$. We assume the `eager` mode of operation: the case for `lazy` is similar. In this proof, we use network invariant property $\mathsf{Inv}(4)$ in Definition 9.2.

Examining the structure of F, we see that the last typing judgement for expressions was $\Gamma \vdash \texttt{freeze}[\texttt{eager}](e) : \texttt{thunk}(U)$ as a result of application of **TE-Freeze**. This rule has the premises, $\Gamma \vdash e : U$ with $\mathsf{fav}(e) = \emptyset$ and $\Gamma \vdash u_i : T_i$ with $\{\vec{u}\} = \mathsf{fnv}(e)$ and $\neg\mathsf{local}(T_i)$. From our initial assumptions that $\Gamma \vdash \sigma : \texttt{ok}$ and $\{\vec{x}\} = \mathsf{dom}_{\mathsf{v}}(\sigma')$, it must be the case that Γ is of the form $\Gamma_1,\vec{x}:\vec{T},\Gamma_2$. Knowing this, we can apply Lemma 8.3(3) to derive $\Gamma \vdash v_i : T_i'$ where $v_i = \sigma(x_i)$ and $T_i' <: T_i$. From this and $\Gamma \vdash e : U$, we can apply Lemma 8.9(1) (substitution) to obtain $\Gamma \vdash e[\vec{v}/\vec{x}] : U'$ and $\mathsf{fav}(e[\vec{v}/\vec{x}]) = \emptyset$, $U' <: U$. Note that by network invariant property, $\mathsf{Inv}(4)$, $\vec{v}$ must be closed, which allows us to conclude the above side condition.

To complete the case we must finally show that $\vdash \texttt{CT}' : \texttt{ok}$. By the assumptions, we know $\vdash \texttt{CT} : \texttt{ok}$ and $\texttt{CT}' = \mathsf{cg}(\texttt{CT},\mathsf{fcl}(e))$. From this and the fact that all classes in $\mathsf{fcl}(e)$ must be well-formed in the class signature CSig, we can apply Lemma 8.4(3)

(correctness of class graph computation) to deduce $\vdash \texttt{CT}' : \texttt{ok}$. Now we can apply **TV-Thunk** to obtain $\Gamma \vdash \ulcorner e[\vec{v}/\vec{x}]\ \texttt{with}\ \texttt{CT}'\ \texttt{from}\ l \urcorner : \texttt{thunk}(U')$. Since $U' <: U$ then $\texttt{thunk}(U') <: \texttt{thunk}(U)$ so this concludes the case.

Case ***RC-Defrost****:*

$F \stackrel{\text{def}}{=} \texttt{defrost}(\ulcorner e\ \texttt{with}\ \texttt{CT}'\ \texttt{from}\ m \urcorner), \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} e[\vec{C}^m/\vec{C}], \sigma, \texttt{CT} \cup \texttt{CT}'$

The premises of **RC-Defrost** state $\{\vec{C}\} = \mathsf{fcl}(e) \setminus \mathsf{dom}(\texttt{CT}')$. From the structure of the expression in F, we see that the last typing rule applied was **TE-Defrost** giving the judgement $\Gamma \vdash \texttt{defrost}(\ulcorner e\ \texttt{with}\ \texttt{CT}'\ \texttt{from}\ m \urcorner) : U$. This derives from the premise $\Gamma \vdash \ulcorner e\ \texttt{with}\ \texttt{CT}'\ \texttt{from}\ m \urcorner : \texttt{thunk}(U')$ with $U' <: U$. In order to infer this thunk, **TV-Thunk** was used with conditions $\vdash \texttt{CT}' : \texttt{ok}$ and $\Gamma \vdash e : U'$. Straightforwardly we can deduce that $\Gamma \vdash e[\vec{C}^m/\vec{C}] : U'$.

In order to complete the case, first we must show that $\vdash \vec{C} : \texttt{tp}$ where $\vec{C}$ was obtained from the side condition of **RC-Defrost** above. $\vdash \texttt{CT} \cup \texttt{CT}' : \texttt{ok}$ follows directly from network invariant $\mathsf{Inv}(3)$ in Definition 9.2.

Case ***RC-DownloadNothing****:*

$F \stackrel{\text{def}}{=} \texttt{download}\ \vec{C}\ \texttt{from}\ l'\ \texttt{in}\ e, \sigma, \texttt{CT}$
$F' \stackrel{\text{def}}{=} e, \sigma, \texttt{CT}$

The premise of **RC-DownloadNothing** has that $C_i \in \mathsf{dom}(\texttt{CT})$. Proof is straightforward. Examining F we see that the last typing rule applied was **TE-ClassLoad** with the premise $\Gamma \vdash e : \vec{U}$. This immediately yields type preservation.

Theorem 10.2 (Subject reduction for threads). *Assume $\Gamma; \Delta \vdash F : \texttt{conf}$, $F \longrightarrow_l F'$ and $F' \not\models \mathsf{Err}$. Then we have $\Gamma; \Delta \vdash F' : \texttt{conf}$.*

Proof. By induction on the derivation $F \longrightarrow_l F'$ with a case analysis on the final typing rules.

Case ***RC-Res****:*

$F \stackrel{\text{def}}{=} (\nu\, u\vec{u})(P, \sigma, \texttt{CT})$
$F' \stackrel{\text{def}}{=} (\nu\, u\vec{u}')(P', \sigma', \texttt{CT}')$

We consider only the case where the reduction occurs under a restricted channel name, i.e. $u = c$. The cases for object identifiers and variables are similar. Let $\Delta = \Delta, c : \texttt{chan}$. By the premises **RC-Res** we have $(\nu\, u\vec{u})(P, \sigma, \texttt{CT}) \longrightarrow_l (\nu\, u\vec{u}')(P', \sigma', \texttt{CT}')$. Examining the shape of F, there are two cases.

The first case is that the last applied rule was **TC-ResC**. Then we have $\Gamma; \Delta', c : \texttt{chan} \vdash (\nu\, \vec{u})(P, \sigma, \texttt{CT}) : \texttt{conf}$ By the inductive hypothesis, we have $\Gamma; \Delta', c : \texttt{chan} \vdash (\nu\, u\vec{u})(P', \sigma', \texttt{CT}') : \texttt{conf}$. Then we can apply **TC-ResC** to obtain the required result.

The second case is that the last applied rule was **TT-Weak**, i.e. we have $\Gamma; \Delta', c' : \texttt{chan} \vdash F : \texttt{conf}$ with the premise $\Gamma; \Delta' \vdash F : \texttt{conf}$. Then by inductive hypothesis, we have $\Gamma; \Delta' \vdash F' : \texttt{conf}$. Noting $c \notin \mathsf{fn}(F')$, the application of **TT-Weak** gives us $\Gamma; \Delta', c : \texttt{chan} \vdash F' : \texttt{conf}$.

Remark Since the case that the last applied rule is **TT-Weak** is trivial, hereafter we omit this case from analysis.

Case ***RC-Str****:* Suppose that $F \equiv F_0 \longrightarrow_l F_0' \equiv F'$. This case follows straightforwardly from the proof of Lemma 8.8(1) (structural equivalence preserves typing).

Case ***RC-Par****:*

$F \stackrel{\text{def}}{=} P_1 \,|\, P_2, \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, \vec{u})(P_1' \,|\, P_2, \sigma', \mathtt{CT}')$

By the premises of **RC-Par** we have that $\vec{u} \notin \mathsf{fnv}(P_2)$ and $P_1, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(P_1', \sigma', \mathtt{CT}')$. This case is similar to **RC-Cong** except a treatment on channel environments. Examining the structure of the configuration F we see that the last thread typing rule applied was **TT-Par**. Let $\vec{u} = \vec{o}\vec{x}\vec{c}$. This gives a judgement of the form $\Gamma; \Delta \vdash P_1 \,|\, P_2 : \mathtt{thread}$, with the premise $\Gamma; \Delta_i \vdash P_i : \mathtt{thread}$ with $\Delta_1 \asymp \Delta_2$. By the premise of the rule, we know $\Gamma, \vec{o} : \vec{C}, \vec{x} : \vec{T}; \Delta_1, \vec{c} : \vec{\mathtt{chan}} \vdash P_1' : \mathtt{thread}$. Note that $u_i \notin \mathsf{fnv}(P_2)$. Hence we can apply the weakening lemma in order to obtain $\Gamma, \vec{o} : \vec{C}, \vec{x} : \vec{T}; \Delta_2 \vdash P_2 : \mathtt{thread}$.

Since $\Delta_1 \asymp \Delta_2$, we can apply **TT-Par** and **TT-Res**, to derive $\Gamma; \Delta \vdash (\nu\, \vec{u})(P_1 \,|\, P_2, \sigma, \mathtt{CT}')$.

Case ***RC-MethLocal****:*

$F \stackrel{\text{def}}{=} E[o.\mathtt{m}(\vec{v})], \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, c)(E[\mathtt{await}\ c] \,|\, o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c, \sigma, \mathtt{CT})$

By **RC-MethLocal** we have c fresh with $o \notin \mathsf{dom_o}(\sigma)$. Inspecting the structure of configuration F, the last rule applied was that for contexts giving rise to the judgement $\Gamma; \Delta \vdash E[o.\mathtt{m}(\vec{v})] : \mathtt{thread}$. By Lemma 8.6, we can assume this rule has the premises $\Gamma \vdash o.\mathtt{m}(\vec{v}) : U$ and $\Gamma; \Delta \vdash E[\,]^U : \mathtt{thread}$.

Choosing a fresh channel name c, we can apply **TT-Await** to $E[\,]$ to infer: $\Gamma; \Delta, c : \mathtt{chanI}(U) \vdash E[\mathtt{await}\ c]^U : \mathtt{thread}$ with $c \notin \mathsf{dom}(\Delta)$.

Next we see that the first premise must have been inferred from rule **TE-Meth** with conditions $\Gamma \vdash o : C$ with $\mathsf{mtype}(\mathtt{m}, C) = \vec{T} \to U$ and $\Gamma \vdash \vec{v} : \vec{T}'$ such that $T_i' <: T_i$. From these, and by picking the same channel name c as before, we can apply **TT-MethWith** in order to derive $\Gamma; c : \mathtt{chanO}(U) \vdash o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c : \mathtt{thread}$.

It is important to note that $\Delta, c : \mathtt{chanI}(U) \asymp c : \mathtt{chanO}(U)$ by definition and the fact that $c \notin \mathsf{dom}(\Delta)$. By this, we can now apply **TT-Par** to obtain $\Gamma; \Delta, c : \mathtt{chan} \vdash E[\mathtt{await}\ c]^U \,|\, o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c : \mathtt{thread}$.

Case ***RC-MethRemote****:*

$F \stackrel{\text{def}}{=} E[o.\mathtt{m}(\vec{v})], \sigma, \mathtt{CT}$
$F' \stackrel{\text{def}}{=} (\nu\, c)(E[\mathtt{await}\ c] \,|\, \mathsf{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c, \sigma, \mathtt{CT})$

By the premises of **RC-MethRemote** c is fresh and $o \notin \mathsf{dom_o}(\sigma)$. Without loss of generality, we set $P \equiv \mathbf{0}$. The initial parts of this case are very similar to the case for **RC-MethLocal**. Here we use the network invariant property $\mathsf{Inv}(10)$ in Definition 9.2.

From the shape of F, and the similarity to the case for local method calls, we can choose a similarly "fresh" c and immediately conclude:

$$\Gamma; \Delta, c : \mathtt{chanI}(U) \vdash E[\mathtt{await}\ c]^U : \mathtt{thread}\ \text{with}\ c \notin \mathsf{dom}(\Delta)$$

Again, we see that **TE-Meth** was applied in the derivation, with premises: $\Gamma \vdash o : C$ with $\mathsf{mtype}(\mathtt{m}, C) = \vec{T} \to U$ and $\Gamma \vdash \vec{v} : \vec{T}'$ with $T_i' <: T_i$.

Given the assumption that $o \notin \mathsf{dom_o}(\sigma)$, then by the network invariant property $\mathsf{Inv}(10)$, it must be the case that $\mathsf{remote}(C)$. With this and the above premises, we can apply rule **TT-GoSer** to derive: $\Gamma; c : \mathtt{chanO}(U) \vdash \mathtt{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c : \mathtt{thread}$ with $\mathsf{remote}(C)$.

Now by $\Delta, c : \mathtt{chanI}(U) \asymp c : \mathtt{chanO}(U)$, we can apply **TT-Par** to derive $\Gamma; \Delta, c : \mathtt{chan} \vdash E[\mathtt{await}\ c] \,|\, \mathtt{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c : \mathtt{thread}$, completing the case.

Case ***RC-MethInvoke****:*

$F \stackrel{\mathsf{def}}{=} o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c, \sigma, \mathtt{CT}$
$F' \stackrel{\mathsf{def}}{=} (\nu\, \vec{x})(e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}], \sigma \cdot [\vec{x} \mapsto \vec{v}], \mathtt{CT})$

By the application of **RC-MethInvoke**, we have that $\sigma(o) = (C, \dots)$ and $\mathsf{mbody}(\mathtt{m}, C, \mathtt{CT}) = (\vec{x}, e)$. From the shape of F, the last rule applied to type the thread component must have been **TT-MethWith**, $\Gamma; \Delta \vdash o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c : \mathtt{thread}$. This is inferred from the premises $\Gamma \vdash o : C$ with $\mathsf{mtype}(\mathtt{m}, C) = \vec{T} \to U$ and $\Gamma \vdash v_i : T_i'$ such that $T_i' <: T_i$.

From our assumption that $\Gamma \vdash \sigma : \mathtt{ok}$, we can apply Lemma 8.2 to obtain $\Gamma \vdash \mathtt{Env}$. Then, choosing $\vec{x} \notin \mathsf{dom}(\Gamma)$, from the assumption that $\Gamma \vdash \sigma : \mathtt{ok}$, $\Gamma \vdash v_i : T_i'$ and $T_i' <: T_i$, Lemma 8.3(1) extends the store with the new variable bindings to $\Gamma, \vec{x} : \vec{T} \vdash \sigma \cdot [\vec{x} \mapsto \vec{v}] : \mathtt{ok}$.

Now we must show preservation of the thread type. By our assumption $\vdash \mathtt{CT} : \mathtt{ok}$, we have that $\vec{x} : \vec{T}, \mathtt{this} : C \vdash e : \mathtt{ret}(U')$ with $U' <: U$. By alpha conversion, we can choose $\vec{x}$ to be the same vector chosen to prove well-formedness of the new store, and so we can apply weakening to $\Gamma \vdash o : C$ to obtain $\Gamma, \vec{x} : \vec{T} \vdash o : C$. Assume $\mathtt{this} \notin \mathsf{dom}(\Gamma)$. Then again we can apply weakening to $\Gamma, \mathtt{this} : C \vdash e : \mathtt{ret}(U')$ to derive $\Gamma, \vec{x} : \vec{T}, \mathtt{this} : C \vdash e : \mathtt{ret}(U')$. With these two premises, we can apply Lemma 8.9(2) to obtain: $\Gamma, \vec{x} : \vec{T} \vdash e[o/\mathtt{this}] : \mathtt{ret}(U'')$ where $U'' <: U' <: U$. Now we can apply **TT-Return**, giving $\Gamma, \vec{x} : \vec{T}; c : \mathtt{chanO}(U) \vdash e[o/\mathtt{this}][\mathtt{return}(c)/\mathtt{return}] : \mathtt{thread}$ to complete the case. The case $\mathtt{this} \in \mathsf{dom}(\Gamma)$ is similar by using Lemma 8.2(3) to obtain $\Gamma, \vec{x} : \vec{T} \vdash e[o/\mathtt{this}] : \mathtt{ret}(U')$.

Case ***RC-Await****:*

$F \stackrel{\mathsf{def}}{=} E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v, \sigma, \mathtt{CT}$
$F' \stackrel{\mathsf{def}}{=} E[v], \sigma, \mathtt{CT}$

Examining the structure of F, the last thread typing rule applied was **TT-Par**. So supposing $\Delta = \Delta', c : \mathtt{chan} = \Delta_1 \odot \Delta_2$ we have $\Gamma; \Delta_1 \odot \Delta_2 \vdash E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v : \mathtt{thread}$ with $\Delta_1 \asymp \Delta_2$. In order for this to be derived, the following premises must hold: $\Gamma; \Delta_1 \vdash E[\mathtt{await}\ c] : \mathtt{thread}$ with $\Delta_1 = \Delta_1', c : \mathtt{chanI}(U), \Delta_1''$; and $\Gamma; \Delta_2 \vdash \mathtt{return}(c)\ v : \mathtt{thread}$ with $\Delta_2 = c : \mathtt{chanO}(U)$.

The environment Δ_1 can be trivially reordered (using the reordering lemma) to the form $\Delta_1', \Delta_1'', c : \mathtt{chanI}(U)$. To derive the first premise, **TT-Await** must have been used, with the conditions $\Gamma, \Delta_1', \Delta_1'' \vdash E[\,]^U : \mathtt{thread}$ and $c \notin \mathsf{dom}(\Delta_1', \Delta_1'')$.

To derive the second premise, **TT-Return** must have been used with the premise $\Gamma \vdash \mathtt{return}\ v : \mathtt{ret}(U')$ and $U' <: U$. Likewise, for this to be derived, **TE-Return** was used with this premise $\Gamma \vdash v : U'$ and $U' <: U$.

We can apply reordering to environment Δ_1', Δ_1'' to obtain Δ'. Then we can use Lemma 8.6 and Weakening to obtain $\Gamma;\Delta' \vdash E[v] : \texttt{thread}$, finishing the case.

Theorem 10.3 (Subject reduction for networks). *Assume* $\Gamma;\Delta \vdash N : \texttt{net}$*,* $N \longrightarrow N'$ *and* $N' \not\models \mathsf{Err}$. *Then we have* $\Gamma;\Delta \vdash N' : \texttt{net}$.

Proof. By induction on the derivation $N \longrightarrow N'$ with a case analysis on the final typing rule applied.

Case ***RN-Conf****:*

$N \stackrel{\mathsf{def}}{=} l[F]$
$N' \stackrel{\mathsf{def}}{=} l[F']$

By the premises of **RN-Conf**, $F \longrightarrow_l F'$. From the structure of N, we see that the last typing rule applied must have been **TN-Conf** with premise $\Gamma;\Delta \vdash F : \texttt{conf}$. Given this and the assumption that $F \longrightarrow_l F'$ we can apply Theorem 10.2 to obtain $\Gamma;\Delta \vdash F' : \texttt{conf}$. We can then re-apply **TN-Conf** to deduce $\Gamma;\Delta \vdash l[F'] : \texttt{net}$ as required.

Case ***RN-Par****:*

$N \stackrel{\mathsf{def}}{=} N_1 \,|\, N_2$
$N' \stackrel{\mathsf{def}}{=} N_1' \,|\, N_2$

By structure of N we see that the last typing rule applied was **TN-Par**. This is inferred from the following premises: $\Gamma;\Delta_1 \vdash N_1 : \texttt{net}$ and $\Gamma;\Delta_2 \vdash N_2 : \texttt{net}$, supposing that $\Delta = \Delta_1 \odot \Delta_2$. We can apply the inductive hypothesis to obtain $\Gamma;\Delta_1 \vdash N_1' : \texttt{net}$. Then we can apply **TN-Par** to derive $\Gamma;\Delta \vdash N_1' \,|\, N_2 : \texttt{net}$, completing the case.

Case ***RN-Res****:*

$N \stackrel{\mathsf{def}}{=} (\nu\, c)N_0$
$N' \stackrel{\mathsf{def}}{=} (\nu\, c)N_0'$

By **RN-Res**, $N_0 \longrightarrow N_0'$. We consider the case where the restricted name is a channel. The case for identifiers is similar. Suppose the last typing rule applied was **TN-ResC**. This is inferred from the premise $\Gamma;\Delta, c : \texttt{chan} \vdash N_0 : \texttt{net}$. We can apply the inductive hypothesis giving $\Gamma;\Delta, c : \texttt{chan} \vdash N_0' : \texttt{net}$. Apply **TN-ResC** to obtain $\Gamma;\Delta \vdash (\nu\, c)N_0' : \texttt{net}$ as required.

Case ***RN-Str****:* Suppose $N \equiv N_0 \longrightarrow N_0' \equiv N'$. Follow straightforwardly from the proof of Lemma 8.8.

Case ***RN-Download****:*

$N \stackrel{\mathsf{def}}{=} l_1[E[\texttt{download}\, \vec{C}\, \texttt{from}\, l_2\, \texttt{in}\, e], \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2]$
$N' \stackrel{\mathsf{def}}{=} l_1[E[\texttt{resolve}\, \vec{D}\, \texttt{from}\, l_2\, \texttt{in}\, e], \sigma_1, \texttt{CT}_1 \cup \texttt{CT}'] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2]$

By the premises of **RN-Download** $\{\vec{D}\} = \{\vec{C}\} \setminus \mathsf{dom}(\texttt{CT}_1)$ and $\texttt{CT}' = \texttt{CT}_2(\vec{D})[\vec{C}^{l_2}/\vec{C}]$. Without loss of generality, we assume $P_1 \equiv P_2 \equiv \mathbf{0}$. This case uses the invariance $\mathsf{Inv}(3)$ in Definition 9.2.

Examining the structure of N, the last typing rule applied was **TN-Par**. Supposing $\Delta = \Delta_1 \odot \Delta_2$ with $\Delta_1 \asymp \Delta_2$, then the following premises must hold: $\Gamma;\Delta_1 \vdash l_1[E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e], \sigma_1, \texttt{CT}_1] : \texttt{net}$ and $\Gamma;\Delta_2 \vdash l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net}$. From the first premise, we see that this is derived (eventually) from the facts that $\Gamma;\Delta_1 \vdash E[\texttt{download}\ \vec{C}\ \texttt{from}\ l_2\ \texttt{in}\ e] : \texttt{thread}$ and $\vdash \texttt{CT}_1 : \texttt{ok}$. From the structure of these judgements we can deduce that rule **TE-ClassLoad** was used with the condition that $\vdash \vec{C} : \texttt{tp}$. The second premise is ultimately derived from $\vdash \texttt{CT}_2 : \texttt{ok}$. From the condition that $\texttt{CT}' \subseteq \texttt{CT}_2$, we can infer $\vdash \texttt{CT}' : \texttt{ok}$ because the renaming $[C_i^{l_2}/C_i]$ does not affect well-formedness. Also from invariant $\mathsf{Inv}(3)$ in Definition 9.2, we know that if $\mathsf{dom}(\texttt{CT}') \cap \mathsf{dom}(\texttt{CT}_1)$ is non-empty, then all overlapping classes must have the same definition. This means we can immediately derive that $\vdash \texttt{CT}_1 \cup \texttt{CT}' : \texttt{ok}$. To complete the case, we rebuild the network using **TC-Conf** and **TN-Par**.

*Case **RN-Leave**:*

$N \stackrel{\mathsf{def}}{=} l_1[\mathsf{go}\ o.\texttt{m}(\vec{v})\ \texttt{with}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2]$
$N' \stackrel{\mathsf{def}}{=} l_1[P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[o.\texttt{m}(\texttt{deserialize}(\vec{v}))\ \texttt{with}\ c \,|\, P_2, \sigma_2, \texttt{CT}_2]$

The premise of **RN-Leave** states that $o \in \mathsf{dom_o}(\sigma_2)$.We shall prove:

$$\Gamma;\Delta \vdash l_1[P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[o.\texttt{m}(\texttt{deserialize}(\vec{v}))\ \texttt{with}\ c \,|\, P_2, \sigma_2, \texttt{CT}_2] : \texttt{net} \quad (10.3.1)$$

To derive $\Gamma;\Delta \vdash N : \texttt{net}$, the last typing rule applied must have been **TN-Par** with the following premises:

$$\Gamma;\Delta_1 \vdash l_1[\mathsf{go}\ o.\texttt{m}(\vec{v})\ \texttt{with}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] : \texttt{net} \quad (10.3.2)$$
$$\Gamma;\Delta_2 \vdash l_2[P_2, \sigma_2, \texttt{CT}_2] : \texttt{net} \quad (10.3.3)$$

We have, by assumption that $\Delta = \Delta_1 \odot \Delta_2$ and $\Delta_1 \asymp \Delta_2$. In order to derive (10.3.2) and (10.3.3) we must have ultimately shown that:

$$\Gamma;\Delta_{11} \vdash \mathsf{go}\ o.\texttt{m}(\vec{v})\ \texttt{with}\ c : \texttt{thread} \qquad \text{with } \Delta_{11} \asymp \Delta_2 \quad (10.3.4)$$
$$\Gamma;\Delta_{12} \vdash P_1 : \texttt{thread} \quad (10.3.5)$$
$$\Gamma;\Delta_2 \vdash P_2 : \texttt{thread} \quad (10.3.6)$$

It is then straightforward to construct l_1 by rule **TC-Conf**:

$$\Gamma;\Delta_{12} \vdash l_1[P_1, \sigma_1, \texttt{CT}_1] : \texttt{net} \quad (10.3.7)$$

Constructing the second location l_2 is more difficult. By using Lemma 8.7, we obtain that $\Delta_{11} \asymp \Delta_2$. Given (10.3.4) and (10.3.6) and this fact, we can apply **TT-Par** to obtain $\Gamma;\Delta_{11} \odot \Delta_2 \vdash \mathsf{go}\ o.\texttt{m}(\vec{v})\ \texttt{with}\ c \,|\, P_2 : \texttt{thread}$. We can then apply **TC-Conf** followed by **TN-Par** to obtain (10.3.1) as required.

*Case **RN-Return**:*

$N \stackrel{\mathsf{def}}{=} l_1[\mathsf{go}\ v\ \texttt{to}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2]$
$N' \stackrel{\mathsf{def}}{=} l_1[P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[\texttt{return}(c)\ \texttt{deserialize}(v) \,|\, P_2, \sigma_2, \texttt{CT}_2]$

We also have that $c \in \mathsf{fn}(P_2)$ by the premise of **RN-Return**. This case issimilar to **RN-Leave**.

Finally we achieve:

Corollary 10.4 (Network invariants and progress properties). $\mathsf{Inv}(r)$ *(*$1 \leq r \leq 18$*) and* $\mathsf{Prog}(r)$ *(*$1 \leq r \leq 10$*) are network invariants with the initial property* Init *defined in Definition 9.3.*

The final corollary specifies the form of the network when all threads terminate.

Corollary 10.5 (Normal Forms). *Assume* $N_0 \models \mathsf{Init}$ *and* $N_0 \longrightarrow\!\!\!\!\rightarrow N \not\longrightarrow$ *and* $N \not\models \mathsf{Err}$*. Then we have* $N \equiv (\nu\,\vec{u})(\prod_{0 \leq i < n} l_i[P_i, \sigma_i, \mathtt{CT}_i])$ *with* $P_i \equiv \prod_{0 \leq j_i < n_i} \mathtt{go}\ v_{j_i}\ \mathtt{to}\ c_{j_i}$*.*

Proof. By induction on N. By the initial condition Init, we can set $\Delta = \vec{c}' : \vec{\mathtt{chan}} \cup \vec{c}_i : \mathtt{chanO}(\vec{U}_i)$. The proof is direct from the progress properties. We only investigate the cases that the reduction happens across different networks. Suppose, for example, by contradiction, that $N \not\longrightarrow$ but there exists P_i such that $P_i \equiv o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, Q_i$. If o is the local object id, then $N \longrightarrow N'$ by $\mathsf{Prog}(8)$. Assume that o is a remote o-id and $o \notin \mathsf{dom_o}(\sigma_i)$. This time by **RC-MethRemote**, $N \longrightarrow N'$, contradiction. Next suppose there exists P_i such that $P_i \equiv \mathtt{go}\ v\ \mathtt{to}\ c \,|\, Q_i$ with $c \in \{\vec{u}\}$ or $c : \mathtt{chan} \in \Delta$. Then by $\mathsf{Prog}(10)$, there exists k such that $P_k \equiv E[\mathtt{await}\ c] \,|\, Q_k$. Then we can apply **RN-Return**, hence a contradiction. The unicity of $\mathtt{go}\ v_{j_i}\ \mathtt{to}\ c_{j_i}$ is derived by $\mathsf{Inv}(18)$. Other cases are also mechanical. □

11 Related Work

Obliq [6] 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. As such, there is no class loading mechanism to consider, which forms an important part of DJ. On the other hand, Obliq has code-passing primitives, as procedures and agents can be passed by value and then executed (Obliq treats local variable assignment within passed code: this feature can also be consistently added to DJ by relaxing **TE-Freeze** and the variable and store invariants in Definition 9.2). DJ models two important concerns in distributed class-based OOPL missing from Obliq, that is dynamic class loading and serialisation (the same term used in [6] refers to serialisation in the sense of transaction theory). Another important difference is that the semantics of Obliq is given in an informal manner in terms of examples, whereas DJ is given a formal operational semantics, which is used for precise examination of new primitives. As a result we have established a typing system and its type soundness, which may not have been done for Obliq so far.

Merro et al [24] encode 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. Their work is orthogonal to the present work, in the sense that ours offers a direct formal semantics and typing system at the language level, by which a detailed analysis on a subtle interplay between distributed OOPL features (including inheritance) is possible.

Yet it would be interesting to find appropriate typed (HO) π-calculi [32] into which dynamics and types of DJ can be encoded faithfully.

Gordon and Hankin [13] 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 that they only use a single location. For this reason and because the calculus is not class-based, they do not treat dynamic class loading or serialisation, which are among the main interests of the present work. Jeffrey [20] treats an extension of [13] for the study of locality with static and dynamic type checking. The aim of his work is quite different, and he does not treat dynamic class loading and object serialisation (though he treats transactional serialisation as in [6]).

Zhao et al [40] propose the SJ calculus for a study of containment in real-time Java. They provide primitives for explicit memory management, which are crucial in the context of their work. The SJ calculus proposes a new 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, they propose a typing system that can statically prevent some scope invariants being broken. Their formalism has similarities with DJ in that it also models an extension of the imperative Java calculus based on FJ [18]. However their study focuses on real-time concurrency in a single location, while ours on dynamic distribution of code in multiple locations. DJ also guarantees similar scoping properties by invariants, for example $\mathsf{Inv}(10)$ in Definition 9.2 ensures that identifiers for local objects do not leak to other locations.

Ohori and Kato [29] 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 such as serialisation and the code passing associated with inheritance and class downloading, whose subtle interplay is a main concern of our paper.

The representation of runtime in formal semantics is not limited to distributed programs, as found in the analysis of an execution model of the .NET CLR by Gordon and Syme [14] and Yu et al [39].

The JavaSeal [34] 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 thunk passing proposed in the present paper.

Class loading and downloading are crucial to many useful Java RMI applications, since they offer a convenient mechanism for distributing code to remote consumers while preserving type-safety. An orthogonal subject is class verification and the maintenance of type safety during linking [23, 31].

Our formulation of class loading is simple, but modular; for example, modifying the class graph definition in Definition 4.12, which follows the "verification *off*" framework in [10, 11], can be consistently replaced by another class loading mechanism such

as "verification *on*" in [10, 11]. For example, in rule **RC-Resolve** the vector $\vec{F}$ is constructed from the direct superclasses of the classes being resolved. Java verification checks subtypes for method receivers and method parameters, therefore as a first approximation we could extend $\vec{F}$ to include class names of a method declaration's formal parameters.

Relatedly, we set the class invariant $\mathsf{Inv}(3)$ in Definition 9.2 for simplicity, but we can easily relax it so as to allow the situation where programs can take advantage of (for example) the latest version of a library without recompilation if the new version is binary compatible with the old. We can control different situations in distributed binary compatibility using invariants as a guidance for consistent refinements of operational semantics and the typing systems.

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 [22] propose a class splitting and pre-fetching algorithm to reduce this latency. Their specific example is applet loading: if the time spent in an interactive portion of an applet could be used to download classes that may be needed in future, it is better to download them ahead of time so that the user does not encounter a large delay, sharing the motivation for our (eager) thunk primitive. The partly eager class loading in their approach is implicit, but requires control flow information about the program in question in order 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.

12 Conclusions and Further Work

This paper introduced a Java-like core language with constructs for distribution including dynamic class loading and serialisation, presented its formal semantics and typing system, and established its basic safety properties through the use of invariants. A new language primitive for distribution, thunk passing, was proposed and consistently integrated into the language with a simple typing rule.

The invariants can be used as a "prescription" of global and local state of a language and runtime which a system designer expects to be guaranteed; if it is not satisfied by his implementation, he can correct or strengthen the typing rules or relax the prescription itself. For example, to make the field access local, we modified the ordinal field access typing rule by adding one constraint ($e \neq \mathtt{this}, o \implies \mathsf{local}(C)$ in **TE-Fld**), guided by the value and object identifiers invariants in Section 9.

The class-based language considered in the present work does not include such language features as casting [18, 4], exceptions [3], synchronisation and polymorphism [18, 5]. These features can be represented by straightforward enrichment of the present syntax and types, even though their precise interplay with distributed language constructs needs examination. An important topic is enrichment of invariants and type

structures to strengthen safety properties (e.g. for security). There are two orthogonal directions. The first concerns mobility. As can be seen in the second example in Section 2, the current type structure of a thunk (e.g. `thunk(int)`) tells the consumer little about the behaviour of the code s/he is about to execute, which can be dangerous. 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, and prior execution, to determine suitable privileges [35, 36]. To ensure the integrity of resources, we can dynamically check an invariant when code arrives (e.g. by adding the constraint in **RC-Defrost**), or adding more fine-grained information on accessibility of methods in the class signatures along the line of [38] for static checking.

The second issue is to extend the syntax and operational semantics to allow complex, structured, communications. For this purpose we have been studying *session types* [16, 33] 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 remote invocation due to network partition (defined by **Err**-rules in Fig. 5.11), but there is no consideration of how to recover from such errors. Also, the class of network errors considered does not cover problems such as the duplication of method calls, return values being lost, etc. In the latter situation, some notion of time-out is generally used to determine whether to re-transmit or fail, and different invocation semantics (for example at-most-once) can be investigated using DJ.

In the future we intend to implement our new primitives for code mobility. An initial version will probably take the form of a source-to-source translator, compiling the $\texttt{freeze}[t](e)$ and $\texttt{defrost}(e)$ operations into standard Java source. Eager class loading via RMI will most likely require modification to the class loading mechanism by installing a custom class loader to work 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 two examples in Section 2 lead to a question on expressiveness between serialisation and freezing constructs: are these two programs semantically equivalent in the sense that all executions which do not involve an error state, can derive the same result? As ongoing work, we are investigating behavioural equivalences in our language using the technique established in the π-calculus [17]. The correctness of the source-to-source translation mentioned above can be investigated using the developed theory. Study along this line would be worthwhile when considering optimisations of RMI interaction patterns [37] or articulations and comparisons of newly proposed language constructs on the basis of formal semantic foundations.

Acknowledgements We thank Mariangiola Dezani and Sophia Drossopoulou for their on-going collaboration on Sessions Types. We also thank Luca Cardelli, Susan Eisenbach, Kohei Honda, Paul Kelly and members of the SLURP group for their discussions. 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) and EPSRC GR/R33465/02, GR/S55538/01 and GR/T04724/01.

References

1. MetaML home page. http://www.cse.ogi.edu/PacSoft/projects/metaml.
2. Martn Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Simplifying types in a calculus for Java exceptions. Technical report, DISI - Università di Genova, 2002.
4. Gavin Bierman, Matthew Parkinson, and Andrew Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
5. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA 98*, October 1998.
6. Luca Cardelli. Obliq: A language with distributed scope. Technical Report 122, Systems Research Center, Digital Equipment Corporation, 1994.
7. Microsoft Corporation. .NET Framework Developer's Guide. http://msdn.microsoft.com/net.
8. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design*. Addison-Wesley, 2001.
9. Sophia Drossopoulou. Advanced issues in object oriented languages course notes. http://www.doc.ic.ac.uk/˜scd/Teaching/AdvOO.html.
10. Sophia Drossopoulou and Susan Eisenbach. Manifestations of Dynamic Linking. In *The First Workshop on Unanticipated Software Evolution (USE 2002)*, Málaga, Spain, June 2002. http://joint.org/use2002/proceedings.html.
11. Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *12th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, April 2003.
12. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.
13. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. Technical Report 457, University of Cambridge Computer Laboratory, February 1999.
14. Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 248–260. ACM Press, 2001.
15. Kohei Honda. Composing processes. In *Proceedings of POPL'96*, pages 344–357, 1996.
16. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 22–138, 1998.
17. Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *TCS*, 151, 1995.
18. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

19. ECMA International. ECMA and ISO C# and common language infrastructure standards. http://msdn.microsoft.com/net/ecma.
20. Alan Jeffrey. A Distributed Object Calculus. In *FOOL*. ACM Press, 2000.
21. Naoki Kobayashi, Benjamin Pierce, and David Turner. Linear types and π-calculus. In *Proceedings of POPL'96*, pages 358–371, 1996.
22. Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing transfer delay using Java class file splitting and prefetching. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 276–291. ACM Press, 1999.
23. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44. ACM Press, 1998.
24. Massimo Merro, Josva Kleist, and Uwe Nestmann. Mobile objects as mobile processes. *Information and Computation*, 177(2):195–241, 2002.
25. Sun Microsystems Inc. Discover the secrets of the Java Serialization API. http://java.sun.com/developer/technicalArticles/Programming/serialization/.
26. Sun Microsystems Inc. Java Remote Method Invocation (RMI) specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.
27. Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.
28. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Info. & Comp.*, 100(1), 1992.
29. Atsushi Ohori and Kazuhiko Kato. Semantics for communication primitives in a polymorphic language. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–112. ACM Press, 1993.
30. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
31. Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 325–336. ACM Press, 2000.
32. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.
33. Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511, 2004.
34. Jan Vitek, Ciarán Bryce, and Walter Binder. Designing JavaSeal or how to make Java safe for agents. *Electronic Commerce Objects*, 1998.
35. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.
36. Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 116–128. ACM Press, 1997.
37. Kwok Yeung and Paul Kelly. Optimizing Java RMI programs by communication restructuring. In *Middleware'03*, volume 2672 of *Lecture Notes in Computer Science*, pages 324–343, 2003.
38. Nobuko Yoshida. Channel dependency types for higher-order mobile processes. In *POPL '04, Conference Record of the 31st Annual Symposium on Principles of Programming Languages*, pages 147–160. ACM Press, 2004. Full version available at www.doc.ic.ac.uk/˜yoshida.
39. Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .net common language runtime. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 39–51. ACM Press, 2004.

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

A Operational Semantics

This appendix lists the operational semantics presented in Section 5 and the look-up functions defined in Section 4.

[Structual Equivalences]

Configurations

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

Threads

$$P \,|\, \mathbf{0} \equiv P$$
$$P \,|\, P_0 \equiv P_0 \,|\, P$$
$$P \,|\, (P_0 \,|\, P_1) \equiv (P \,|\, P_0) \,|\, P_1$$
$$(\nu u)(P \,|\, P_0) \equiv (\nu u)P \,|\, P_0 \qquad u \notin \mathsf{fn}(P_0)$$
$$(\nu c)\mathbf{0} \equiv \mathbf{0}$$
$$(\nu u)(\nu u')P \equiv (\nu u')(\nu u)P$$
$$\texttt{return}(d)\ \varepsilon \equiv \texttt{return}(d)$$
$$\varepsilon; e \equiv e$$
$$\texttt{return}\ \varepsilon \equiv \texttt{return}$$

Networks

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

[Lookup Functions]

Field lookup

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

Method type lookup

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

Method body lookup

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

Valid method overriding

$$\frac{\mathsf{mtype}(\texttt{m}, D) = \vec{T} \to U \text{ implies } \vec{T} = \vec{T}' \text{ and } U = U'}{\mathsf{override}(\texttt{m}, D, \vec{T}' \to U')}$$

[Expression]

RC-Var
$x, \sigma, \mathtt{CT} \longrightarrow_l \sigma(x), \sigma, \mathtt{CT}$

RC-Cond
$\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2, \sigma, \mathtt{CT} \longrightarrow_l e_1, \sigma, \mathtt{CT}$
$\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2, \sigma, \mathtt{CT} \longrightarrow_l e_2, \sigma, \mathtt{CT}$

RC-Fld
$$\frac{\sigma(o) = (C, \vec{f} : \vec{v})}{o.\mathtt{f_i}, \sigma, \mathtt{CT} \longrightarrow_l v_i, \sigma, \mathtt{CT}}$$

RC-Seq
$$\frac{e_1, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(v, \sigma', \mathtt{CT}')}{e_1; e_2, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(e_2, \sigma', \mathtt{CT}')}\ \vec{u} \notin \mathsf{fnv}(e_2)$$

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

RC-Ass
$x := v, \sigma, \mathtt{CT} \longrightarrow_l v, \sigma[x \mapsto v], \mathtt{CT}$

RC-FldAss
$$\frac{\sigma' = \sigma[o \mapsto \sigma(o)[f \mapsto v]]}{o.\mathtt{f} := v, \sigma, \mathtt{CT} \longrightarrow_l v, \sigma', \mathtt{CT}}\ o \in \mathsf{dom_o}(\sigma)$$

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

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

RC-Cong
$$\frac{e, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(e', \sigma', \mathtt{CT}')}{E[e], \sigma, \mathtt{CT} \longrightarrow_l (\nu\,\vec{u})(E[e'], \sigma', \mathtt{CT}')}\ \vec{u} \notin \mathsf{fnv}(E)$$

[Method Invocation]

RC-MethLocal
$E[o.\mathtt{m}(\vec{v})] \,|\, P, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,c)(E[\mathtt{await}\ c] \,|\, o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, P, \sigma, \mathtt{CT})\ \ c\ \text{fresh}, o \in \mathsf{dom_o}(\sigma)$

RC-MethRemote
$E[o.\mathtt{m}(\vec{v})] \,|\, P, \sigma, \mathtt{CT} \longrightarrow_l (\nu\,c)(E[\mathtt{await}\ c] \,|\, \mathsf{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c \,|\, P, \sigma, \mathtt{CT})$
$c\ \text{fresh}, o \notin \mathsf{dom_o}(\sigma)$

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

RC-Await
$E[\mathtt{await}\ c] \,|\, \mathtt{return}(c)\ v, \sigma, \mathtt{CT} \longrightarrow_l E[v], \sigma, \mathtt{CT}$

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

RN-Leave
$l_1[\mathsf{go}\ o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c \,|\, P_1, \sigma_1, \mathtt{CT_1}] \,|\, l_2[P_2, \sigma_2, \mathtt{CT_2}]$
$\longrightarrow l_1[P_1, \sigma_1, \mathtt{CT_1}] \,|\, l_2[o.\mathtt{m}(\mathtt{deserialize}(\vec{v}))\ \mathtt{with}\ c \,|\, P_2, \sigma_2, \mathtt{CT_2}] \qquad o \in \mathsf{dom_o}(\sigma_2)$

RN-Return
$l_1[\mathsf{go}\ v\ \mathtt{to}\ c \,|\, P_1, \sigma_1, \mathtt{CT_1}] \,|\, l_2[P_2, \sigma_2, \mathtt{CT_2}]$
$\longrightarrow l_1[P_1, \sigma_1, \mathtt{CT_1}] \,|\, l_2[\mathtt{return}(c)\ \mathtt{deserialize}(v) \,|\, P_2, \sigma_2, \mathtt{CT_2}] \qquad c \in \mathsf{fn}(P_2)$

[Serialisation]

Serialize

$$\frac{\sigma' = \bigcup \mathsf{og}(\sigma, v_i) \qquad \{\vec{o}\} = \mathsf{dom_o}(\sigma')}{\mathtt{serialize}(\vec{v}), \sigma, \mathtt{CT} \longrightarrow_l \lambda \vec{o}.(\vec{v}, \sigma', l), \sigma, \mathtt{CT}}$$

Deserialize

$$\frac{\{\vec{F}\} = \{C \mid \sigma'(o_i) = (C, \dots)\} \qquad \vec{o} \notin \mathsf{dom}(\sigma)}{\mathtt{deserialize}(\lambda \vec{o}.(\vec{v}, \sigma', m)), \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{o})(\mathtt{download}\ \vec{F}\ \mathtt{from}\ m\ \mathtt{in}\ \vec{v}, \sigma \cup \sigma', \mathtt{CT})}$$

[Code Creation]

RC-Freeze

$$\frac{\{\vec{x}\} = \mathsf{fv}(e) \quad v_i = \sigma(x_i) \qquad \mathtt{CT}' = \begin{cases} \mathsf{cg}(\mathtt{CT}, \mathsf{fcl}(e)) & t = \mathtt{eager} \\ \emptyset & t = \mathtt{lazy} \end{cases}}{\mathtt{freeze}[t](e), \sigma, \mathtt{CT} \longrightarrow_l \ulcorner e[\vec{v}/\vec{x}]\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ l \urcorner, \sigma, \mathtt{CT}}$$

RC-Defrost

$$\frac{\{\vec{C}\} = \mathsf{fcl}(e) \setminus \mathsf{dom}(\mathtt{CT}')}{\mathtt{defrost}(\ulcorner e\ \mathtt{with}\ \mathtt{CT}'\ \mathtt{from}\ m \urcorner), \sigma, \mathtt{CT} \longrightarrow_l e[\vec{C^m}/\vec{C}], \sigma, \mathtt{CT} \cup \mathtt{CT}'}$$

[Class Downloading]

RC-Resolve

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

RN-Download

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

RC-DownloadNothing

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

[Threads]

RC-Par

$$\frac{P_1, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(P_1', \sigma', \mathtt{CT}')}{P_1 \mid P_2, \sigma, \mathtt{CT} \longrightarrow_l (\nu\, \vec{u})(P_1' \mid P_2, \sigma', \mathtt{CT}')}\ \vec{u} \notin \mathsf{fnv}(P_2)$$

RC-Str

$$\frac{F \equiv F_0 \longrightarrow_l F_0' \equiv F'}{F \longrightarrow_l F'}$$

RC-Res

$$\frac{(\nu\, \vec{u})(P, \sigma, \mathtt{CT}) \longrightarrow_l (\nu\, \vec{u}')(P', \sigma', \mathtt{CT}')}{(\nu\, u\vec{u})(P, \sigma, \mathtt{CT}) \longrightarrow_l (\nu\, u\vec{u}')(P', \sigma', \mathtt{CT}')}$$

[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
$$\frac{N \equiv N_0 \longrightarrow N_0' \equiv N'}{N \longrightarrow N'}$$

[Errors]

Err-NullFld
$$\texttt{null.f}, \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$$

Err-NullFldAss
$$\texttt{null.f} := v, \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$$

Err-NullMeth
$$\texttt{null.m}(\vec{v}), \sigma, \texttt{CT} \longrightarrow_l \texttt{Error}, \sigma, \texttt{CT}$$

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

Err-MLossLeave
$$l_1[\text{go}\ o.\texttt{m}(\vec{v})\ \texttt{with}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \longrightarrow l_1[\texttt{Error} \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \quad o \in \mathsf{dom_o}(\sigma_2)$$

Err-MLossReturn
$$l_1[\text{go}\ v\ \texttt{to}\ c \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \longrightarrow l_1[\texttt{Error} \,|\, P_1, \sigma_1, \texttt{CT}_1] \,|\, l_2[P_2, \sigma_2, \texttt{CT}_2] \quad c \in \mathsf{fn}(P_2)$$

B Typing System

This appendix presents the typing systems.

[Types] $\boxed{\vdash S : \texttt{tp}}$

Wf-Base
$$\frac{-}{\begin{array}{l} \vdash \texttt{void} : \texttt{tp} \\ \vdash \texttt{bool} : \texttt{tp} \\ \vdash \texttt{chan} : \texttt{tp} \end{array}}$$

Wf-SC
$$\frac{\vdash U : \texttt{tp} \vee U \in \mathsf{CSig}}{\begin{array}{l} \vdash \texttt{chanI}(U) : \texttt{tp} \\ \vdash \texttt{chanO}(U) : \texttt{tp} \\ \vdash \texttt{ret}(U) : \texttt{tp} \\ \vdash \texttt{thunk}(U) : \texttt{tp} \end{array}}$$

Wf-Vec
$$\frac{\vdash U_i : \texttt{tp}}{\vdash \vec{U} : \texttt{tp}}$$

Wf-Ser
$$\frac{\vdash \vec{U} : \texttt{tp}}{\vdash \texttt{ser}(\vec{U}) : \texttt{tp}}$$

Wf-Sig
$$\frac{\begin{array}{l} \mathsf{override}(\texttt{m}_\texttt{i}, D_i, \vec{T}_i \to U_i) \quad \vdash D : \texttt{tp} \\ \forall S \in \{\vec{T}, \vec{U}, \vec{T}_i\}.\ \vdash S : \texttt{tp} \vee S \in \mathsf{dom}(\mathsf{CSig}) \end{array}}{\vdash \texttt{extends}\ D\ [\texttt{remote}]\ \vec{T}\vec{\texttt{f}}\ \ \{\texttt{m}_\texttt{i} : \vec{T}_i \to U_i\} : \texttt{tp}}$$

Wf-Ctp
$$\frac{\vdash \mathsf{CSig}(C) : \texttt{tp}}{\vdash C : \texttt{tp}}$$

Wf-Csig
$$\frac{\begin{array}{l} \forall C \in \mathsf{dom}(\mathsf{CSig}) \\ \vdash C : \texttt{tp} \end{array}}{\vdash \mathsf{CSig} : \texttt{ok}}$$

[Subtyping] $\boxed{C <: D}$

ST-Refl
$$\frac{-}{T <: T}$$

ST-Trans
$$\frac{\begin{array}{l} C <: D \\ D <: E \end{array}}{C <: E}$$

ST-Vec
$$\frac{\begin{array}{l} U_i' <: U_i \\ 0 \leq i \lneq n \end{array}}{\vec{U}' <: \vec{U}}$$

ST-Ser
$$\frac{\vec{U}' <: \vec{U}}{\mathtt{ser}(\vec{U}') <: \mathtt{ser}(\vec{U})}$$

ST-Expr
$$\frac{U' <: U}{\begin{array}{l} \mathtt{thunk}(U') <: \mathtt{thunk}(U) \\ \mathtt{ret}(U') <: \mathtt{ret}(U) \end{array}}$$

ST-Class
$$\frac{\mathsf{CSig}(C) = \mathtt{extends}\ D\ [\mathtt{remote}]\ \vec{T}\vec{\mathtt{f}}\ \ \{\mathtt{m_i} : \vec{T_i} \rightarrow U_i\}}{C <: D}$$

[Environments] $\boxed{\Gamma;\Delta \vdash \mathtt{Env}}$

E-Nil
$$\frac{-}{\emptyset \vdash \mathtt{Env}}$$

E-Var
$$\frac{\begin{array}{l} \vdash T : \mathtt{tp} \\ x \notin \mathsf{dom}(\Gamma) \end{array}}{\Gamma, x : T \vdash \mathtt{Env}}$$

E-Oid
$$\frac{\begin{array}{l} \vdash C : \mathtt{tp} \\ o \notin \mathsf{dom}(\Gamma) \end{array}}{\Gamma, o : C \vdash \mathtt{Env}}$$

E-This
$$\frac{\begin{array}{l} \vdash C : \mathtt{tp} \\ \mathtt{this} \notin \mathsf{dom}(\Gamma) \end{array}}{\Gamma, \mathtt{this} : C \vdash \mathtt{Env}}$$

E-CNil
$$\frac{-}{\Gamma;\emptyset \vdash \mathtt{Env}}$$

E-Chan
$$\frac{\begin{array}{l} \vdash \tau : \mathtt{tp} \\ \Gamma;\Delta \vdash \mathtt{Env} \\ c \notin \mathsf{dom}(\Delta) \end{array}}{\Gamma;\Delta, c : \tau \vdash \mathtt{Env}}$$

[Stores] $\boxed{\Gamma;\Delta \vdash \mathtt{Env}}$

S-CNil
$$\frac{-}{\Gamma \vdash \emptyset : \mathtt{ok}}$$

S-Var
$$\frac{\begin{array}{ll} \Gamma \vdash \sigma : \mathtt{ok} & \\ \Gamma \vdash x : T & x \notin \mathsf{dom_v}(\sigma) \\ \Gamma \vdash v : T' & T' <: T \end{array}}{\Gamma \vdash \sigma \cdot [x \mapsto v] : \mathtt{ok}}$$

S-Var
$$\frac{\begin{array}{ll} \Gamma \vdash \sigma : \mathtt{ok} & o \notin \mathsf{dom_o}(\sigma) \\ \Gamma \vdash o : C & \mathsf{fields}(C) = \vec{T}\vec{f} \\ \Gamma \vdash v_i : T_i' & T_i' <: T_i \end{array}}{\Gamma \vdash \sigma \cdot [o \mapsto (C, \vec{f} : \vec{v})]}$$

[Values] $\boxed{\Gamma \vdash v : U}$

TV-Bool
$$\frac{\Gamma \vdash \mathtt{Env}}{\begin{array}{l} \Gamma \vdash \mathtt{true} : \mathtt{bool} \\ \Gamma \vdash \mathtt{false} : \mathtt{bool} \end{array}}$$

TV-Null
$$\frac{\vdash C : \mathtt{tp}}{\Gamma \vdash \mathtt{null} : C}$$

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

TV-Empty
$$\frac{\Gamma \vdash \mathtt{Env}}{\Gamma \vdash \varepsilon : \mathtt{void}}$$

TV-Thunk
$$\frac{\begin{array}{l} \vdash \mathtt{CT} : \mathtt{ok} \\ \Gamma \vdash e : U \end{array}}{\Gamma \vdash \ulcorner e\ \mathtt{with}\ \mathtt{CT}\ \mathtt{from}\ l \urcorner : \mathtt{thunk}(U)}$$

TV-Blob
$$\frac{\begin{array}{ll} \Gamma, \vec{o} : \vec{C} \vdash \vec{v} : \vec{U} & \mathsf{local}(C_i) \\ \Gamma, \vec{o} : \vec{C} \vdash \sigma : \mathtt{ok} & o \in \{\vec{o}\} = \mathsf{dom_o}(\sigma) \end{array}}{\Gamma \vdash \lambda \vec{o}.(o, \sigma, l) : \mathtt{ser}(C)}$$

[Expressions] $\boxed{\Gamma \vdash e : S}$

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

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

TE-Cond
$$\frac{\exists S : S_1 <: S \wedge S_2 <: S \quad \Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : S}$$

TE-Fld
$$\frac{\Gamma \vdash e : C \quad \vdash C : \texttt{tp} \quad e \neq \texttt{this}, o \implies \mathsf{local}(C) \quad \mathsf{fields}(C) = \vec{T}\vec{f}}{\Gamma \vdash e.\texttt{f}_\texttt{i} : T_i}$$

TE-Seq
$$\frac{\Gamma \vdash e : \texttt{void} \quad \Gamma \vdash e' : S}{\Gamma \vdash e; e' : S}$$

TE-Dec
$$\frac{\Gamma \vdash e : T \quad T <: T' \quad \Gamma, x : T \vdash e_0 : S}{\Gamma \vdash T'\ x = e\ ;\ e_0 : S}$$

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

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

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

TE-Meth
$$\frac{\mathsf{mtype}(\texttt{m}, C) = \vec{T} \rightarrow U \quad \Gamma \vdash e_0 : C \quad \Gamma \vdash \vec{e} : \vec{T'} \quad T'_i <: T_i}{\Gamma \vdash e_0.\texttt{m}(\vec{e}) : U}$$

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

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

TE-Serialize
$$\frac{\Gamma \vdash e : \vec{U}}{\Gamma \vdash \texttt{serialize}(e) : \texttt{ser}(\vec{U})}$$

TE-Deserialize
$$\frac{\Gamma \vdash e : \texttt{ser}(\vec{U})}{\Gamma \vdash \texttt{deserialize}(e) : \vec{U}}$$

TE-DeserVal
$$\frac{\Gamma \vdash e : U \quad U \neq \texttt{ser}(C)}{\Gamma \vdash \texttt{deserialize}(e) : U}$$

TE-Freeze
$$\frac{\{\vec{u}\} = \mathsf{fnv}(e) \quad \mathsf{fav}(e) = \emptyset \quad \Gamma \vdash e : U \quad \Gamma \vdash u_i : T_i \quad \neg\mathsf{local}(T_i)}{\Gamma \vdash \texttt{freeze}[t](e) : \texttt{thunk}(U)}$$

TE-Defrost
$$\frac{\Gamma \vdash e : \texttt{thunk}(U)}{\Gamma \vdash \texttt{defrost}(e) : U}$$

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

TE-Pe
$$\frac{\Gamma \vdash pe : T}{\Gamma \vdash pe : \texttt{void}}$$

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

[Threads] $\boxed{\Gamma;\Delta,c:\mathtt{chan}\vdash P:\mathtt{thread}}$

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

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

TT-Weak
$$\frac{\Gamma;\Delta\vdash P:\mathtt{thread}\quad c\notin\mathsf{dom}(\Delta)}{\Gamma;\Delta,c:\mathtt{chan}\vdash P:\mathtt{thread}}$$

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

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

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

TT-GoSer
$$\frac{\Gamma\vdash o:C\quad \Gamma\vdash\vec{v}:\vec{T}'\quad \vec{T}'<:\vec{T}\quad \mathsf{remote}(C)\quad \mathsf{mtype}(\mathtt{m},C)=\vec{T}\rightarrow U}{\Gamma;c:\mathtt{chanO}(U)\vdash\mathsf{go}\ o.\mathtt{m}(\mathtt{serialize}(\vec{v}))\ \mathtt{with}\ c:\mathtt{thread}}$$

TT-MethWith
$$\frac{\Gamma\vdash o:C\quad \Gamma\vdash v_i:T_i'\quad T_i'<:T_i\quad \mathsf{mtype}(\mathtt{m},C)=\vec{T}\rightarrow U}{\Gamma;c:\mathtt{chanO}(U)\vdash o.\mathtt{m}(\vec{v})\ \mathtt{with}\ c:\mathtt{thread}}$$

TT-DeserWith
$$\frac{\Gamma\vdash o:C\quad \Gamma\vdash\lambda\vec{o}.(\vec{v},\sigma,l):\mathsf{ser}(\vec{T}')\quad \vec{T}'<:\vec{T}\quad \mathsf{remote}(C)\quad \mathsf{mtype}(\mathtt{m},C)=\vec{T}\rightarrow U}{\begin{array}{l}\Gamma;c:\mathtt{chanO}(U)\vdash o.\mathtt{m}(\mathtt{deserialize}(\lambda\vec{o}.(\vec{v},\sigma,l)))\ \mathtt{with}\ c:\mathtt{thread}\\ \Gamma;c:\mathtt{chanO}(U)\vdash\mathsf{go}\ o.\mathtt{m}(\lambda\vec{o}.(\vec{v},\sigma,l))\ \mathtt{with}\ c:\mathtt{thread}\end{array}}$$

TT-ValTo
$$\frac{\Gamma\vdash v:U'\quad U'<:U\quad \neg\mathsf{local}(U')}{\begin{array}{l}\Gamma;c:\mathtt{chanO}(U)\vdash\mathsf{go}\ \mathtt{serialize}(v)\ \mathtt{to}\ c:\mathtt{thread}\\ \Gamma;c:\mathtt{chanO}(U)\vdash\mathsf{go}\ v\ \mathtt{to}\ c:\mathtt{thread}\end{array}}$$

TT-GoTo
$$\frac{\Gamma\vdash e:\mathsf{ser}(C')\quad C'<:C}{\Gamma;c:\mathtt{chanO}(C)\vdash\mathsf{go}\ e\ \mathtt{to}\ c:\mathtt{thread}}$$

[Configuration] $\boxed{\Gamma;\Delta,c:\mathtt{chan}\vdash F:\mathtt{conf}}$

TC-ResC
$$\frac{\Gamma;\Delta,c:\mathtt{chan}\vdash F:\mathtt{conf}}{\Gamma;\Delta\vdash(\nu c)F:\mathtt{conf}}$$

TC-ResId
$$\frac{\Gamma,u:T;\Delta\vdash F:\mathtt{conf}\quad u\in\mathsf{dom}(F)}{\Gamma;\Delta\vdash(\nu u)F:\mathtt{conf}}$$

TC-Conf
$$\frac{\Gamma;\Delta\vdash P:\mathtt{thread}\quad \Gamma\vdash\sigma:\mathtt{ok}\quad \vdash\mathtt{CT}:\mathtt{ok}\quad \mathtt{FCT}\subseteq\mathtt{CT}}{\Gamma;\Delta\vdash P,\sigma,\mathtt{CT}:\mathtt{conf}}$$

[Network] $\boxed{\Gamma;\Delta,c:\mathtt{chan}\vdash N:\mathtt{net}}$

TN-Nil
$$\frac{\Gamma;\emptyset\vdash\mathtt{Env}}{\Gamma;\emptyset\vdash\mathbf{0}:\mathtt{net}}$$

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

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

TN-Weak
$$\frac{\Gamma;\Delta\vdash N:\mathtt{net}\quad c\notin\mathsf{dom}(\Delta)}{\Gamma;\Delta,c:\mathtt{chan}\vdash N:\mathtt{net}}$$

TN-ResId
$$\frac{\Gamma,u:T;\Delta\vdash N:\mathtt{net}\quad u\in\mathsf{dom}(N)}{\Gamma;\Delta\vdash(\nu\,u)N:\mathtt{net}}$$

TN-ResC
$$\frac{\Gamma;\Delta,c:\mathtt{chan}\vdash N:\mathtt{net}}{\Gamma;\Delta\vdash(\nu\,c)N:\mathtt{net}}$$

[Method] $\boxed{\Gamma\vdash M:\mathtt{ok}\ \mathtt{in}\ C}$

M-ok
$$\frac{\mathsf{mtype}(\mathtt{m},C)=\vec{T}\to U\quad U'<:U\quad \mathtt{this}:C,\vec{x}:\vec{T}\vdash e:\mathtt{ret}(U')}{\mathtt{this}:C\vdash U\,\mathtt{m}(\vec{T}\vec{x})\{e\}:\mathtt{ok}\ \mathtt{in}\ C}$$

[Class] $\boxed{\vdash L:\mathtt{ok}}$

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

[Class Table] $\boxed{\vdash\mathtt{CT}:\mathtt{ok}}$

CT-Nil
$$\frac{-}{\vdash\emptyset:\mathtt{ok}}$$

CT
$$\frac{\vdash\mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f};K\vec{M}\}:\mathtt{ok}\quad \vdash\mathtt{CT}:\mathtt{ok}}{\vdash\mathtt{CT}\cdot[C\mapsto\mathtt{class}\ C\ \mathtt{extends}\ D\{\vec{T}\vec{f};K\vec{M}\}]:\mathtt{ok}}$$