A distributed object calculus

Alan Jeffrey
DePaul University
ajeffrey@cs.depaul.edu

December 1999
To appear in Proc. FOOL 2000

1 Introduction

Distributed object-oriented languages are becoming increasingly accepted as network programming languages. The success of Java, and the Java security model in particular, has made the study of type systems for OO languages critical to the success of distributed programming.

Abadi and Cardelli [1] have provided an excellent framework for the study of object-based OO languages in the object calculus. Gordon and Hankin [5] have shown how this work can be extended to deal with concurrent languages, but to date there is no work on a distributed semantics for the object calculus.

There have been many semantics for distributed calculi, notably Cardelli and Gordon's [4] ambient calculus, Fournet et al.'s [3] distributed join calculus, Hennessy and Riely's [11, 12] Dπ, Yoshida and Hennessy's [15] $D\lambda\pi$, Sewell's [13] distributed π, and Vitek and Castagna's [14] Seal calculus. None of these languages are directly object-oriented, although there are strong parallels between them and OO languages, using codings of OO into the π-calculus such as [2, 6, 8, 9, 10].

In this paper, these two strands of research are brought together, to provide a model for distributed object-based languages. In doing so, two features of distributed programming become clear:

- the importance of separating *serializable* and *non-serializable* data using types, and
- the use of *located objects* to model remote objects.

These features are not specific to OO languages: they will arise in any distributed language with mutable state, for example any system supporting RPC. In Java, serializable data is part of the persistency API, and remote objects are implemented as stub-skeleton pairs as part of RMI.

The main result of each part of this paper is a subject reduction result, which shows that well-typedness is preserved by reduction. This, together with type safety properties of the type system mean that well-typed programs cannot violate the safety and security policies of the network.

This paper starts by reviewing Gordon and Hankin's concurrent object calculus in Section 2.

The new material then starts in Section 3, where the distributed object calculus is presented. We justify the use of serializable objects, and show how the type system can be used to ban object references from leaving their home location.

Although the distributed object calculus is safe, it is not very useful, since it does not allow object references to be transmitted to remote locations. In Section 4 we develop a type system which uses a serializable type $\mathsf{located}\ \tau$ for located objects, and show that the resulting language is powerful enough to encode a distributed π-calculus. There are two approaches which could be taken here:

- *Static binding*: provide a notion of 'resource name' which includes the home location of the resource: the resource can then only be used at the resource's home location. This corresponds to a type '$\exists l : \mathsf{location}\,.\,(\tau\ \mathsf{at}\ l)$': a resource of type τ which exists at one place on the network. We will call this a *located object type*. Java supports static binding using stub-skeleton pairs.

- *Dynamic binding*: provide a notion of 'resource name', and a concept of resource usage that can take place anywhere on the network. This corresponds to a type '$\forall l : \mathsf{location}\,.\,(\tau\ \mathsf{at}\ l)$': a resource of type τ which can be used anywhere on the network. For example, the ambient calculus [4] can be regarded as a calculus of dynamic resources, and Java supports dynamic binding using object registries.

In this paper, we shall investigate static binding, and leave dynamic binding for future work.

In a trusted network, we can provide a static type system which guarantees type safety. In the presence of attackers, however, we need to use some run-time type-checking to ban malicious code. In Section 5, we use a modification of Hennessy and Riely's [12] run-time typing to remove malicious code from untrusted sites. The run-time type system presented here is much simpler than theirs, but at the cost of requiring every distributed object to carry its type.

This paper is part of the SafetyNet project [7] to design and implement a network programming language. Much of the material discussed here has been implemented in SafetyNet, which has a compiler to Java.

I would like to thank Andy Gordon, Matthew Hennessy, Andy Moran, Tim Owen, James Riely, Ian Wakeman, the DePaul Foundations of Programming Languages group, and the anonymous referees for comments and discussions about this paper and related topics.

2 Concurrent object calculus

In this section, we present (with slightly different syntax) Gordon and Hankin's [5] concurrent object calculus. The main difference is we use an explicit syntactic category of *configurations*, which represents a computation running within a single location.

A configuration consists of a *heap* and a *thread pool*. The heap defines a collection of *object references* denoted $p \mapsto O$, and the thread pool is modeled as a collection of expressions evaluating in parallel:

$$p_1 \mapsto O_1 \parallel \cdots \parallel p_m \mapsto O_m \parallel E_1 \parallel \cdots \parallel E_n$$

Threads can contain private object references, denoted in the π-calculus style as $\nu p\,.\,C$.

2.1 Syntax

We assume grammars for *variables* x, y, z, *names* p, q, r and *labels* l.

We also assume some base values (to model integers, floating point numbers, and so on) ranged over by b. We require at least the base value $*$ to represent the canonical value of unit type.

We introduce a grammar of *values*:

$$\begin{array}{rcl} V, W & ::= & x \\ & | & p \\ & | & b \end{array}$$

a grammar of *objects*:

$$O \quad ::= \quad [l_1 = M_1, \ldots, l_n = M_n]$$

a grammar of *methods*:

$$M \quad ::= \quad \varsigma x\,.\,E$$

a grammar of *expressions*:

$$\begin{array}{rcl} E, F, G & ::= & V \\ & | & \mathsf{let}\ x = E; F \\ & | & \mathsf{new}\ O \\ & | & V.l \\ & | & V.l \Leftarrow M \\ & | & \mathsf{clone}\ V \\ & | & \mathsf{die} \\ & | & \mathsf{spawn}\ E \end{array}$$

and a grammar of *configurations*:

$$\begin{array}{rcl} C & ::= & E \\ & | & p \mapsto O \\ & | & 0 \\ & | & C_1 \parallel C_2 \\ & | & \nu p\,.\,C \end{array}$$

We shall view terms up to α-conversion.

We shall use syntax sugar to make examples more readable. Following Moggi's monadic meta-language, we can extend the language to allow expressions where only values are allowed, for example $E.l$ as short-hand for $\mathsf{let}\ x = E; x.l$.

We shall write E for the method $\varsigma\mathsf{this}\,.\,E$.

We shall write $E;F$ for $\mathsf{let}\ x = E; F$ for fresh x.

2.2 Examples

We recall from Abadi and Cardelli [1] that there is a coding of the recursive λ-calculus into the object calculus.

We recall from Gordon and Hankin [5] that there is a coding of the asynchronous π-calculus into the concurrent object calculus (which requires a primitive for locks, but we shall not discuss those in this paper) using types for channels:

$$\begin{array}{rcl} \mathsf{chan}\ \tau & = & [\mathsf{read} : \tau, \mathsf{write} : \tau \to \bot] \\ \mathsf{rochan}\ \tau & = & [\mathsf{read} : \tau] \\ \mathsf{wochan}\ \tau & = & [\mathsf{write} : \tau \to \bot] \end{array}$$

We can use channels to simulate call-cc (in a rather inefficient manner where we create a new thread every time call-cc is used):

$$\frac{\Gamma, k : \tau \to \bot \vdash E : \bot}{\Gamma \vdash \mathsf{callcc}\ k\,.\,E : \tau}$$

defined:

$$\begin{array}{rcl} \mathsf{callcc}\ k\,.\,E & = & \mathsf{let}\ l = \mathsf{buff1.build}; \\ & & \mathsf{let}\ k = l.\mathsf{write}; \\ & & \mathsf{spawn}\ E; \\ & & l.\mathsf{read} \end{array}$$

2.3 Reduction

We use a structural congruence on configurations, which allows us to view configurations as a multiset of located threads. The structural congruence $C \equiv C'$ is the smallest congruence on configurations satisfying the axioms in Figure 1.

The reduction precongruence $C \mapsto C'$ is the smallest precongruence on configurations satisfying the axioms in Figure 2. Note that the reduction rules have been organized so that there are *no* evaluation contexts on expressions: reduction is only between configurations. This makes proving properties about reduction much simpler, since we do not have to consider let as an evaluation context.

We define $C \to C'$ to be $C \equiv\mapsto\equiv C'$.

2.4 Type system

In this paper, for simplicity we shall not consider recursive types, polymorphism and variance annotations. These should not pose any technical difficulties, since the material in Abadi and Cardelli should apply here.

We assume a grammar of *base types* ranged over by B. We require at least the base type $*$ to model the unit type. In examples we will use $\bot$ to model the empty type: this is useful for modelling methods such as asynchronous send which do not return values.

$$C_1 \parallel 0 \equiv C_1 \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_3 \qquad C_1 \parallel (\nu p\,.\,C_2) \equiv \nu p\,.\,(C_1 \parallel C_2)$$

Figure 1: Structural congruence rules (where p is not free in C_1)

$$\begin{array}{rcll}
\mathsf{let}\ x = V; E & \mapsto & E[V/x] & \text{(Redn Let }\beta\text{)} \\
\mathsf{let}\ y = (\mathsf{let}\ x = E; F); G & \mapsto & \mathsf{let}\ x = E; (\mathsf{let}\ y = F; G) & \text{(Redn Let Assoc)} \\
\mathsf{let}\ x = \mathsf{new}\ O; E & \mapsto & \nu p\,.\,(p \mapsto O \parallel \mathsf{let}\ x = p; E) & \text{(Redn New)} \\
p \mapsto [\vec{l} = \vec{M}, l = \varsigma y\,.\,F] \parallel \mathsf{let}\ x = p.l; E & \mapsto & p \mapsto [\vec{l} = \vec{M}, l = \varsigma y\,.\,F] \parallel \mathsf{let}\ y = F[p/y]; F & \text{(Redn Access)} \\
p \mapsto [\vec{l} = \vec{M}, l = M] \parallel \mathsf{let}\ x = (p.l \Leftarrow M'); E & \mapsto & p \mapsto [\vec{l} = \vec{M}, l = M'] \parallel \mathsf{let}\ x = p; E & \text{(Redn Override)} \\
p \mapsto O \parallel \mathsf{let}\ x = \mathsf{clone}\ p; E & \mapsto & \nu q\,.\,(p \mapsto O \parallel q \mapsto O \parallel \mathsf{let}\ x = q; E) & \text{(Redn Clone)} \\
\mathsf{let}\ x = \mathsf{spawn}\ F; E & \mapsto & F \parallel \mathsf{let}\ x = *; E & \text{(Redn Spawn)} \\
\mathsf{let}\ x = \mathsf{die}; E & \mapsto & 0 & \text{(Redn Die)}
\end{array}$$

Figure 2: Reduction rules

$$[\vec{l} : \vec{\tau}, \vec{l}' : \vec{\tau}'] \quad <: \quad [\vec{l} : \vec{\tau}] \quad \text{(Subtype Object)}$$

Figure 3: Subtyping rules

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{(Type Val Id)} \qquad \frac{\Gamma \vdash V : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash V : \tau_2}\ \text{(Type Val Subsum)}$$

Figure 4: Type rules for values $\Gamma \vdash V : \tau$

$$\frac{\Gamma \vdash M_1 : \tau \to \tau_1 \quad \ldots \quad \Gamma \vdash M_n : \tau \to \tau_n}{\Gamma \vdash [l_1 = M_1, \ldots, l_n = M_n] : \tau}\ \text{(Type Object)} \qquad \frac{\Gamma, x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \varsigma x\,.\,E : \tau_1 \to \tau_2}\ \text{(Type Method)}$$

$$\frac{\Gamma \vdash E : \tau_1 \quad \Gamma, x : \tau_1 \vdash F : \tau_2}{\Gamma \vdash \mathsf{let}\ x = E; F : \tau_2}\ \text{(Type Exp Let)} \qquad \frac{\Gamma \vdash O : \sigma}{\Gamma \vdash \mathsf{new}\ O : \sigma}\ \text{(Type Exp New)}$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash V.l_i : \tau_i}\ \text{(Type Exp Access)} \qquad \frac{\Gamma \vdash V : \tau \quad \Gamma \vdash M : \tau \to \tau_i}{\Gamma \vdash V.l_i \Leftarrow M : \tau}\ \text{(Type Exp Override)}$$

$$\frac{}{\Gamma \vdash \mathsf{die} : \sigma}\ \text{(Type Exp Die)} \qquad \frac{\Gamma \vdash E : \sigma}{\Gamma \vdash \mathsf{spawn}\ E : *}\ \text{(Type Exp Spawn)}$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathsf{clone}\ V : \tau}\ \text{(Type Exp Clone)} \qquad \frac{\Gamma \vdash E : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash E : \tau_2}\ \text{(Type Exp Subsum)}$$

Figure 5: Type rules for expressions $\Gamma \vdash E : \sigma$, where $\tau = [l_1 : \tau_1, \ldots, l_n : \tau_n]$

$$\frac{\Gamma \vdash O : \tau}{\Gamma \vdash (p \mapsto O) : (p : \tau)}\ \text{(Type Config Heap)} \qquad \frac{\Gamma \vdash E : \tau}{\Gamma \vdash E : ()}\ \text{(Type Config Thread)}$$

$$\frac{}{\Gamma \vdash 0 : ()}\ \text{(Type Config Empty)} \qquad \frac{\Gamma \vdash C_1 : \Gamma_1 \quad \Gamma \vdash C_2 : \Gamma_2}{\Gamma \vdash (C_1 \parallel C_2) : (\Gamma_1, \Gamma_2)}\ \text{(Type Config Concur)}$$

$$\frac{\Gamma, p : \tau \vdash C : (\Gamma', p : \tau)}{\Gamma \vdash (\nu p\,.\,C) : \Gamma'}\ \text{(Type Config Scope)}$$

Figure 6: Type rules for configurations $\Gamma \vdash C : \Gamma'$

We introduce a grammar of *types* (for distinct l_i):

$$\begin{array}{rcl}\tau & ::= & [l_1 : \tau_1, \ldots, l_n : \tau_n] \\ & | & B\end{array}$$

and a grammar of *type environments* (for distinct x_i):

$$\Gamma \quad ::= \quad x_1 : \tau_1, \ldots, x_n : \tau_n$$

We shall view object types and type environments up to reordering.

The type judgments extend the judgments for base values $b : B$ by the rules in Figures 3–6.

2.5 Subject reduction

We can now show our first subject reduction result, from Gordon and Hankin [5], that configuration typing is preserved by reduction:

Proposition 1 *If* $C \to C'$ *and* $\Gamma \vdash C : \Gamma$ *then* $\Gamma \vdash C' : \Gamma$.

3 Distributed object calculus

We will now add a notion of *location* to the language, and allow threads to migrate between locations. We do this by introducing a notion of *network*, which is a collection of configurations each running at a separate location:

$$L_1[\![C_1]\!] \parallel \cdots \parallel L_n[\![C_n]\!]$$

In this paper, we are considering a very simple, fixed flat set of locations, with no routing information, dynamic location creation or hierarchy of locations. As far as the material in this section is concerned, there are few technical difficulties with allowing a richer notion of location, but this would complicate Section 5 quite considerably, since we would not be able to partition locations into good and bad as easily. We leave consideration of richer notions of location as future work.

The main problem we will address in this section is that we want to avoid configurations where an object at one location is being accessed at another, for example:

$$K[\![p \mapsto [\mathsf{val} = V]]\!] \parallel L[\![\mathsf{let}\, x = p.\mathsf{val}; E]\!]$$

To ensure that each configuration can only access local state, we use the following type rule for configurations:

$$\frac{\Gamma \vdash C : \Gamma}{\vdash L[\![C]\!] : \mathsf{network}}$$

This type rule ensures that malformed networks such as the above cannot be type-checked. Unfortunately, it is very easy to define type systems where subject reduction does not hold, because well-typed configurations can be reduced to malformed networks such as the above.

3.1 Serializable types

In this paper, we use a type system based on dividing objects into *serializable* objects (which can be sent across the network) and *non-serializable* objects (which cannot). We will first justify this decision, by introducing a number of unsafe type systems and showing why they do not satisfy subject reduction.

First attempt Add a new expression at L spawn E which will execute the thread E at location L, with type:

$$\frac{\Gamma \vdash V : \mathsf{location} \quad \Gamma \vdash E : \tau}{\Gamma \vdash \mathsf{at}\, V\, \mathsf{spawn}\, E : *}$$

and reduction rule:

$$\begin{array}{rl} & L[\![\mathsf{let}\, x = \mathsf{at}\, L'\, \mathsf{spawn}\, E; F \parallel C]\!] \parallel L'[\![C']\!] \\ \mapsto & L[\![\mathsf{let}\, x = *; F \parallel C]\!] \parallel L'[\![E \parallel C']\!]\end{array}$$

Unfortunately subject reduction fails:

$$\begin{array}{l}\mathsf{let\ local} = \mathsf{new}\, [\mathsf{val} = V]; \\ \mathsf{at\ somewhere\ spawn}\ (\\ \quad \mathsf{let}\, x = \mathsf{local.val}; E \\ )\end{array}$$

This will spawn a remote thread at a remote location which will attempt to access the local object local. Since the type system does not allow access to non-local objects, subject reduction fails.

Second attempt The problem with the first attempt is the rule for remote spawn, which allows the spawned thread to carry all its local context with it, even pointers to objects which do not exist at the remote location. One attempt to fix this is to only allow base types to be sent to remote locations:

$$\frac{\Gamma, \Gamma' \vdash V : \mathsf{location} \quad \Gamma \vdash E : \tau}{\Gamma, \Gamma' \vdash \mathsf{at}\, V\, \mathsf{spawn}\, E : *}[\Gamma \text{ only contains base types}]$$

This type system does preserve subject reduction, but is very restrictive, since only base types can be carried between locations. In particular, recursion is implemented using objects, so it is impossible in this type system to write a program such as:

$$\begin{array}{l}\mathsf{hop} = \lambda x : \mathsf{locationList}\,.\,\mathsf{case}\, x\, \mathsf{of}\ (\\ \qquad \mathsf{Empty}\ () \to \mathsf{die} \\ \qquad \mathsf{Cons}\ (y, ys) \to \mathsf{at}\, y\, \mathsf{spawn}\, E; \mathsf{hop}(ys) \\ )\end{array}$$

This program is intended to hop through a list of locations, performing some computation E at each place. Unfortunately, since hop is not of base type, the remote spawn will not type check using this type system.

Third attempt We introduce a new class of *serializable objects* which can be sent across the network. We do this by adding a new kind of value:

$$\mathsf{serial}\ [\vec{l} = \vec{M}]$$

with type:

$$\mathsf{serial}\ [\vec{l} : \vec{\tau}]$$

We use essentially the same type rules for serializable objects, for example the rule for creating new objects is:

$$\frac{\Gamma \vdash O : [\vec{l} : \vec{\tau}]}{\Gamma \vdash \mathsf{serial}\, O : \mathsf{serial}\ [\vec{l} : \vec{\tau}]}$$

$$N_1 \parallel 0 \equiv N_1 \qquad N_1 \parallel N_2 \equiv N_2 \parallel N_1 \qquad N_1 \parallel (N_2 \parallel N_3) \equiv (N_1 \parallel N_2) \parallel N_3 \qquad N_1 \parallel (\nu p\,.\,N_2) \equiv \nu p\,.\,(N_1 \parallel N_2) \qquad L[\![\nu p\,.\,N]\!] \equiv \nu p\,.\,L[\![N]\!]$$

Figure 7: Structural congruence for networks (where p is not free in N_1)

$$\begin{array}{rcll}
L_1[\![C_1 \parallel \mathsf{let}\ x = \mathsf{at}\ L_2\ \mathsf{spawn}\ E;F]\!] \parallel L_2[\![C_2]\!] & \mapsto & L_1[\![C_1 \parallel \mathsf{let}\ x = *;F]\!] \parallel L_2[\![C_2 \parallel E]\!] & \text{(Redn Remote Spawn)} \\
L[\![C \parallel \mathsf{let}\ x = \mathsf{localhost};E]\!] & \mapsto & L[\![C \parallel \mathsf{let}\ x = L;E]\!] & \text{(Redn Localhost)} \\
\mathsf{let}\ x = \mathsf{serial}\ O.l;E & \mapsto & \mathsf{let}\ x = F[\mathsf{serial}\ O/y];E & \text{(Redn Serial Access)} \\
\mathsf{let}\ x = (\mathsf{serial}\ O.l \Leftarrow M);E & \mapsto & \mathsf{let}\ x = (\mathsf{serial}\ [\vec{l} = \vec{M}, l = M]);E & \text{(Redn Serial Update)}
\end{array}$$

Figure 8: Reduction semantics (where $O = [\vec{l} = \vec{M}, l = \varsigma y\,.\,F]$

$$\mathsf{serial}\ [\vec{l} : \vec{\tau}, \vec{l}' : \vec{\tau}'] \quad <: \quad \mathsf{serial}\ [\vec{l} : \vec{\tau}] \quad \text{(Subtype Serial)}$$

Figure 9: Subtype rules

$$\frac{\Sigma \vdash O : [\vec{l} : \vec{\tau}]}{\Gamma, \Sigma \vdash \mathsf{serial}\ O : \mathsf{serial}\ [\vec{l} : \vec{\tau}]} \quad \text{(Type Val Serial)}$$

Figure 10: Type rules for values $\Gamma \vdash V : \tau$

$$\frac{\Gamma, \Sigma \vdash V : \mathsf{location} \quad \Sigma \vdash E : \sigma}{\Gamma, \Sigma \vdash \mathsf{at}\ V\ \mathsf{spawn}\ E : *} \ \text{(Type Exp Remote Spawn)} \qquad \frac{}{\Gamma \vdash \mathsf{localhost} : \mathsf{location}} \ \text{(Type Exp Localhost)}$$

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash V.l_i : \tau_i} \ \text{(Type Exp Serial Access)} \qquad \frac{\Gamma, \Sigma \vdash V : \tau \quad \Sigma \vdash M : \tau \to \tau_i}{\Gamma, \Sigma \vdash V.l_i \Leftarrow M : \tau} \ \text{(Type Exp Serial Override)}$$

Figure 11: Type rules for expressions $\Gamma \vdash E : \sigma$, where $\tau = \mathsf{serial}\ [l_1 : \tau_1, \ldots, l_n : \tau_n]$

$$\frac{\Gamma \vdash C : \Gamma}{\vdash L[\![C]\!] : \mathsf{network}} \ \text{(Type Network Located)} \qquad \frac{\vdash N : \mathsf{network}}{\vdash \nu p\,.\,N : \mathsf{network}} \ \text{(Type Config Scope)}$$

$$\frac{}{\vdash 0 : \mathsf{network}} \ \text{(Type Network Empty)} \qquad \frac{\vdash N_1 : \mathsf{network} \quad \vdash N_2 : \mathsf{network}}{\vdash (N_1 \parallel N_2) : \mathsf{network}} \ \text{(Type Network Concur)}$$

Figure 12: Type rules for networks $\vdash N : \mathsf{network}$

We provide serializable objects with a semantics as non-imperative objects (see Abadi and Cardelli [1] for details of imperative vs. non-imperative objects). A type is *serializable* iff it is a base type, or a serializable object type. The rule for remote spawn is now:

$$\frac{\Gamma, \Gamma' \vdash V : \mathsf{location} \quad \Gamma \vdash E : \tau}{\Gamma, \Gamma' \vdash \mathsf{at}\ V\ \mathsf{spawn}\ E : *} [\Gamma \text{ only contains serializable types}]$$

Unfortunately, this still does not work, because objects are essentially closures, and so have their own context. If we create an object whose context is mutable, we can create an object whose type is serializable but which still contains local pointers:

```
let local = new [val = V];
let wrapper = serial [val = local.val];
at somewhere spawn (
   let x = wrapper.val; E
)
```

We need to ensure that when we create an object which might be sent across the network that not only is the object immutable, but also the context each of its methods carries with it is immutable.

Final attempt The problem is the type rule for creating new objects, which allows serializable objects to carry pointers to non-serializable objects in their closure. To stop this, we add a side-condition to the formation rule for serializable objects:

$$\frac{\Gamma \vdash O : [\vec{l} : \vec{\tau}]}{\Gamma, \Gamma' \vdash \mathsf{serial}\ O : \mathsf{serial}\ [\vec{l} : \vec{\tau}]} [\Gamma \text{ only contains serializable types}]$$

This system allows us to define recursive distributed code, and preserves subject reduction.

This notion of serializable type is based on Java's object persistency API, but with one major change: Java allows imperative objects to be serialized, which results in some implicit cloning. It is difficult to tell from the source of a Java program using RMI when an object is being cloned. In this paper, we avoid this problem by only allowing non-imperative objects to be serialized.

3.2 Syntax

We assume a grammar of locations K and L, and that locations are base values.

Extend the grammar of values with serializable objects:

$$\begin{array}{rcl} V & ::= & \cdots \\ & | & \mathsf{serial}\ O \end{array}$$

Extend the grammar of expressions with remote spawning and finding the current host:

$$\begin{array}{rcl} E & ::= & \cdots \\ & | & \mathsf{at}\ V\ \mathsf{spawn}\ E \\ & | & \mathsf{localhost} \end{array}$$

Add a grammar of *networks*:

$$\begin{array}{rcl} N & ::= & L[\![C]\!] \\ & | & 0 \\ & | & N_1 \,||\, N_2 \\ & | & \nu p \,.\, N \end{array}$$

We insist that in any network $N_1 \,||\, N_2$ that the locations of N_1 and N_2 are disjoint.

3.3 Reduction

Extend the reduction precongruence with the rules in Figure 7 and 8.

3.4 Type system

Add a new base type for locations $\mathsf{location}$, with $L : \mathsf{location}$, and add a type for serializable objects (for distinct l_i):

$$\begin{array}{rcl} \tau & ::= & \cdots \\ & | & \mathsf{serial}\ [l_1 : \tau_1, \ldots, l_n : \tau_n] \end{array}$$

A type is *serializable* if it is a base type or $\mathsf{serial}\ \iota$. A context $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ is serializable if all of the σ_i are serializable. Let Σ range over serializable contexts.

The type rules for the distributed mutable object calculus are given in Figures 9–12.

3.5 Subject reduction

Proposition 2 *If* $C \to C'$ *and* $\Gamma \vdash C : \Gamma$ *then* $\Gamma \vdash C' : \Gamma$.

4 Located object calculus

The distributed mutable object calculus is safe, but is not very useful, since it does not allow distributed systems to communicate across the network. A thread can start at one location and migrate to another, but it cannot access any of the existing heap at its new location, since mutable objects are not serializable. Newly arrived threads can create new heap, but they cannot access any previously existing heap. This is especially unfortunate, since threads can only communicate via shared mutable objects, not by any other mechanism.

In this section we propose adding *located types* to a distributed object system, which correspond to *well-known* objects which can be accessed via remote threads. They correspond to stub-skeleton pairs in systems such as RPC or Java RMI.

Located objects are heap-allocated objects with a home location. They can only be accessed directly from their home location, but they are serializable, so we can program agents which migrate to the home of the object, and then access it.

For simplicity, we could have made all mutable objects located, but we believe this is not realistic: in practice creating a stub-skeleton pair for an object is a fairly heavyweight operation, and creating a pair for every object would add a tremendous overhead to a language. Another possibility would be to create stub-skeleton pairs for objects 'on the fly': whenever a heap-allocated object is serialized, we make it a located object 'behind the scenes'. This removes the requirement to have a stub-skeleton pair for every object, but makes it possible for a local object to become localized at any time. We prefer to use a static approach, where we can analyze a system statically to determine which objects may leave the current location. Also, for security reasons, we may wish to have objects which cannot be accessed except through well-known interfaces: with located objects we can enforce this security requirement statically.

4.1 Syntax

Extend the language of values with located objects:

$$\begin{array}{rcl} V & ::= & \cdots \\ & | & (q : \tau\ \mathsf{at}\ L) \end{array}$$

Extend the language of expressions with the ability to create new located objects, and the ability to goto the home location of a located object and read it:

$$\begin{array}{rcl} E & ::= & \cdots \\ & | & \mathsf{located}\ (V : \tau) \\ & | & \mathsf{goto}\ \mathsf{let}\ x = V; E \end{array}$$

4.2 Example

We can call a remote procedure or method:

$$\mathsf{goto}\ V\ \vec{V} \quad = \quad \mathsf{goto}\ \mathsf{let}\ x = V; x\ \vec{V}$$

$$L[\![C \parallel \mathsf{let}\ x = \mathsf{located}\ (O:\tau);E]\!] \quad \mapsto \quad \nu q\,.\,L[\![C \parallel q \mapsto O \parallel \mathsf{let}\ x = (q:\tau\ \mathsf{at}\ L);E]\!] \qquad \text{(Redn Located New)}$$

$$L_1[\![C_1 \parallel \mathsf{let}\ x = (\mathsf{goto}\ \mathsf{let}\ y = (q:\tau\ \mathsf{at}\ L_2);E);F]\!] \parallel L_2[\![C_2]\!] \quad \mapsto \quad L_1[\![C_1 \parallel \mathsf{let}\ x = *;F]\!] \parallel L_2[\![C_2 \parallel \mathsf{let}\ y = q;E]\!] \qquad \text{(Redn Located Access)}$$

Figure 13: Reduction rules (where q does not occur free in C)

$$\frac{}{\Gamma \vdash (q:\tau\ \mathsf{at}\ L) : \mathsf{located}\ \tau}\ \text{(Type Located Object)} \qquad \frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathsf{located}\ (V:\tau) : \mathsf{located}\ \tau}\ \text{(Type Located New)}$$

Figure 14: Type rules

$$\mathsf{goto}\ \mathsf{let}\ x = V.l\,\vec{V};E \quad = \quad \mathsf{goto}\ \mathsf{let}\ y = V;\mathsf{let}\ x = y.l\,\vec{V};E$$

We implement 'distributed call-cc', with type:

$$\frac{\Sigma, k : \mathsf{located}\ (\tau \to \bot) \vdash E : \bot}{\Sigma \vdash \mathsf{remote\ callcc}\ k\,.\,E : \tau}[\tau \text{ is serializable}]$$

by performing regular call-cc, and registering the return function:

$$\begin{aligned} &\mathsf{remote\ callcc}\ k\,.\,E \\ &\quad = \quad \mathsf{callcc}\ k'\,.\,(\mathsf{let}\ k = \mathsf{located}\ (k' : \tau \to \bot);E) \end{aligned}$$

We can plug together remote calling and distributed call-cc to define remote method invocation, which is typed:

$$\frac{\Gamma \vdash V : \mathsf{located}\ [\ldots, l : \vec{\tau} \to \tau] \quad \Gamma \vdash \vec{V} : \vec{\tau}}{\Gamma \vdash \mathsf{remote}\ V.l\,\vec{V} : \tau}[\vec{\tau}, \tau \text{ are serializable}]$$

and defined:

$$\begin{aligned} \mathsf{remote}\ V.l\,\vec{V} \quad = \quad & \mathsf{remote\ callcc}\ k\,.\,(\\ & \quad \mathsf{goto}\ \mathsf{let}\ x = V.l\,\vec{V}; \\ & \quad \mathsf{goto}\ k\ x \\ &) \end{aligned}$$

With remote method invocation, we can, for example, code up much of Sewell's distributed π-calculus, by extending Gordon and Hankin's coding of the asynchronous π-calculus with global channels and process migration. In the following, let a, b and c range over global channels:

$$\begin{aligned} \nu a\,.\,P \quad &= \quad \mathsf{let}\ a = \mathsf{located}\ (\mathsf{buff1.build} : \mathsf{chan}\ \tau);P \\ \mathsf{migrate}\ x\,.\,P \quad &= \quad \mathsf{at}\ x\ \mathsf{spawn}\ P;\mathsf{die} \\ a\langle V \rangle \quad &= \quad \mathsf{goto}\ a.\mathsf{write}\ V \\ a(x)\,.\,P \quad &= \quad \mathsf{let}\ x = \mathsf{remote}\ a.\mathsf{read};P \\ !a(x)\,.\,P \quad &= \quad \mathsf{new}\ [\\ &\qquad\quad \mathsf{run} = (\\ &\qquad\qquad \mathsf{let}\ x = \mathsf{remote}\ a.\mathsf{read}; \\ &\qquad\qquad \mathsf{spawn}\ P; \\ &\qquad\qquad \mathsf{this.run} \\ &\qquad\quad) \\ &\qquad].\mathsf{run} \end{aligned}$$

We can also code up most of Sewell's types as object types:

$$\begin{aligned} \updownarrow_{LL} \tau \quad &= \quad \mathsf{chan}\ \tau \\ \updownarrow_{GG} \tau \quad &= \quad \mathsf{located\ chan}\ \tau \\ \updownarrow_{L-} \tau \quad &= \quad \mathsf{rochan}\ \tau \\ \updownarrow_{G-} \tau \quad &= \quad \mathsf{located\ rochan}\ \tau \\ \updownarrow_{-L} \tau \quad &= \quad \mathsf{wochan}\ \tau \\ \updownarrow_{-G} \tau \quad &= \quad \mathsf{located\ wochan}\ \tau \end{aligned}$$

However, we do not have a translation of Sewell's $\updownarrow_{LG} \tau$ or $\updownarrow_{GL} \tau$. Also, Sewell uses subsumption to transform global channels to local channels, and allows that subsumption to take place anywhere on the network. In the located object calculus, global channels can only be used locally at their home location, so we cannot use subsumption to convert a global channel into a local one. Sewell's fine grained semantics for $a(y)\,.\,P$ in distributed π gives the same reductions as this translation: a new continuation channel k is created, we migrate to the home of a, read locally on a, migrate back to the home of k and send the result on k. Our syntax makes this semantics explicit rather than implicit.

4.3 Reduction

Extend the reduction precongruence with rules in Figure 13.

4.4 Type system

Extend the type judgments with rules in Figure 14.

4.5 Subject reduction

Subject reduction does not hold! The problem is that sites only know local type information, not global type information, so they may have incorrect knowledge about the type of a located object. The following configuration type-checks:

$$L[\![q \mapsto [\,]]\!] \parallel K[\![\mathsf{goto}\ \mathsf{let}\ x = (q : \mathsf{chan}\ *\ \mathsf{at}\ L);x.\mathsf{write}\ *]\!]$$

but after reduction can become:

$$L[\![q \mapsto [\,] \parallel q.\mathsf{write}\ *]\!] \parallel K[\![*]\!]$$

which does not type-check, since [] does not have a field write. This problem is especially severe in the presence of intruders, since malicious code could try to subvert the type system to gain access to system privileges. We need to find some way to enforce a constraint, which is that in (Redn Located Access) that at L_2, q has type τ.

We shall write $C \vdash q : \tau$ whenever $\Gamma \vdash C : \Gamma$ and $\Gamma : \tau$.

In a network N containing $L[\![C]\!]$, we say C' *respects* L iff for every $(q : \tau \mathsf{\ at\ } L)$ in C' we have $C \vdash q : \tau$.

In a network N containing $L_1[\![C_1]\!]$, we say L_1 respects L_2 iff C_1 respects L_2.

A network N is *locally respectful* iff for every location L in N, L respects L.

A network N is *globally respectful* iff for every locations L_1 and L_2 in N, L_1 respects L_2.

There are (at least) two possible ways to force the subject reduction property of (Red Located Access): either add a side-condition that q has the appropriate type to (Redn Located Access), or ensure that the configuration is locally respectful.

Unfortunately, being locally respectful is not preserved by reduction: the culprit is remote spawning. There are again two possible fixes: either add a side-condition that E is well-registered at L_2 in (Redn Remote Spawn) and (Redn Located Access), or require C to be globally well-registered.

Global respectfulness can be implemented using digital signatures: each location is equipped with a public/private key pair, and we implement $(q : \tau \mathsf{\ at\ } L)$ by having L sign the object with its private key. Any site can then implement (Type Located Object) by checking the signature.

There is a trade-off here: digital signatures are expensive and require each location to have a public/private key pair, but they allow type-checking to be done anywhere on the network, and require no extra run-time checks. Resolving this engineering trade-off is beyond the scope of this paper.

Proposition 3 *If $N \to N'$ and $\vdash N : \mathsf{network}$ then:*

1. *If N is globally respectful then N' is globally respectful.*

2. *If N is locally respectful, and we add the side condition that E is respectful at L_2 in (Redn Remote Spawn) and (Redn Located Access), then N' is locally-well registered.*

3. *If N' is locally respectful then $\vdash N' : \mathsf{network}$.*

4. *If we add the condition that $C_2 \vdash q : \tau$ in (Redn Located Access) then $\vdash N' : \mathsf{network}$.*

5 Subject reduction in the presence of intruders

We shall now give a brief account of type safety in the presence of intruders, who are trying deliberately to subvert the type system.

Following Riely and Hennessy [12], we split the locations into *good* locations G and *bad* locations B. We assume a notion of trust between locations: the only requirement on the trust relation is that good locations do not trust bad locations.

We add a side-condition to the rules for remote spawning:

1. Add the side-condition that either L_2 trusts L_1 or $\vdash E : \tau$ to (Redn Remote Spawn).

2. Add the side-condition that either L_2 trusts L_1 or $\vdash E[q/y] : \tau$ to (Redn Located Access).

Code from a trusted location is allowed to migrate without any test, but code from an untrusted location is type-checked before it is allowed to run. In this model of trust, communication is cheap between sites in trusted space, and all the run-time testing is done when code enters trusted space from untrusted space.

Note that these side-conditions are much simpler than the equivalent rule in Riely and Hennessy, because their model allows for a dynamic model of trust, and allows for sites to learn about located objects at other sites.

A system is *partially typed* if we allow code at bad locations to violate the type discipline. Bad locations can then try to infect the network with badly typed code, and gain access to privileged resources. The additional type rule for bad locations is:

$$\frac{}{\vdash B[\![C]\!] : \mathsf{network}} \quad \text{(Type Network Bad)}$$

We weaken the requirements for respectfulness to allow bad locations to lie about their resource types. A configuration C is *locally trustworthy* iff for every G in C, G respects G in C. A configuration C is *globally trustworthy* iff for every G and L in C, L respects G in C. Note that we require bad locations to respect good locations, which is why some form of authentication mechanism such as digital signatures would be required in an implementation.

Proposition 4 *If $N \to N'$ and $\vdash N : \mathsf{network}$ then:*

1. *If N is globally trustworthy then N' is globally trustworthy.*

2. *If N is locally trustworthy, and we add the side condition that E respects L_2 in (Redn Remote Spawn) and (Redn Located Access), then N' is locally trustworthy.*

3. *If N' is locally trustworthy then $\vdash N' : \mathsf{network}$.*

4. *If we add the condition that $C_2 \vdash q : \tau$ in (Redn Located Access) then $\vdash N' : \mathsf{network}$.*

References

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

[2] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1999. To appear.

[3] C. Fournet, G. Gonthier, J.J. Levy, L. Marganget and D. Remy. A calculus of mobile agents. In *Proc. CONCUR 96*, pages 406–421. Springer-Verlag, 1996.

[4] L. Cardelli and A.D. Gordon. Mobile ambients. In *Proc. FOSSaCS 98*, LNCS, 1998.

[5] A.D. Gordon and P.D. Hankin. A concurrent object calculus: Reduction and typing. In *Proc. HLCL 98*, 1998‘.

[6] Hans Hüttel and Josva Kleist. Objects as mobile processes. In *Proc. Mathematical Foundations of Programming Semantics*, 1996.

[7] A.S.A. Jeffrey and I. Wakeman. The safetynet project. Available electronically from http://www.cogs.susx.ac.uk/projects/safetynet/, 1998.

[8] C. Jones. A pi-calculus semantics for an object-based design notation. In *Proc. Concur 93*, volume 715 of *LNCS*, pages 158–172. Springer-Verlag, 1993.

[9] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, 1998.

[10] B.C. Pierce and D.N. Turner. Concurrent objects in a process calculus. In *Proc. TPPP 94*, volume 907 of *LNCS*, pages 187–215. Springer-Verlag, 1994.

[11] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proc. POPL 98*. ACM Press, 1998.

[12] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. POPL 99*. ACM Press, 1999.

[13] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. ICALP 98*, number 1443 in LNCS, pages 695–706. Springer-Verlag, 1998.

[14] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *Proc. Workshop on Internet Programming Languages*, 1999.

[15] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proc. Concur 99*, 1999.