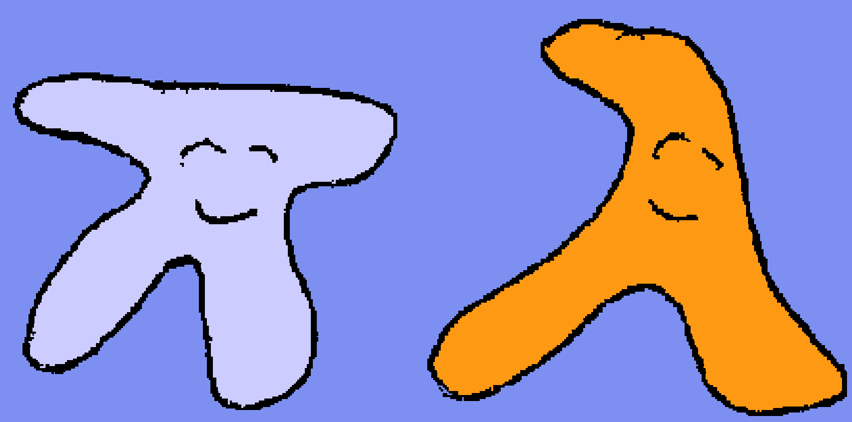
Imperial College London

Explicit Code Mobility in Java RMI

Alexander Ahern and Nobuko Yoshida

Department of Computing, Imperial College London.



THE DJ CALCULUS

DJ is a Java-like core language with primitives for distributed programming and explicit code mobility. These primitives offer the programmer fine-grained control of type-safe code distribution, which is crucial for improving the performance and safety of distributed object-oriented applications.

DJ IN CONTEXT

•FJ [Igarashi et al]
•MJ [Bierman et al]

Foundational calculi describing the semantics of fragments of Java.

•Manifestations of Java Dynamic Linking [Drossopoulou & Eisenbach]

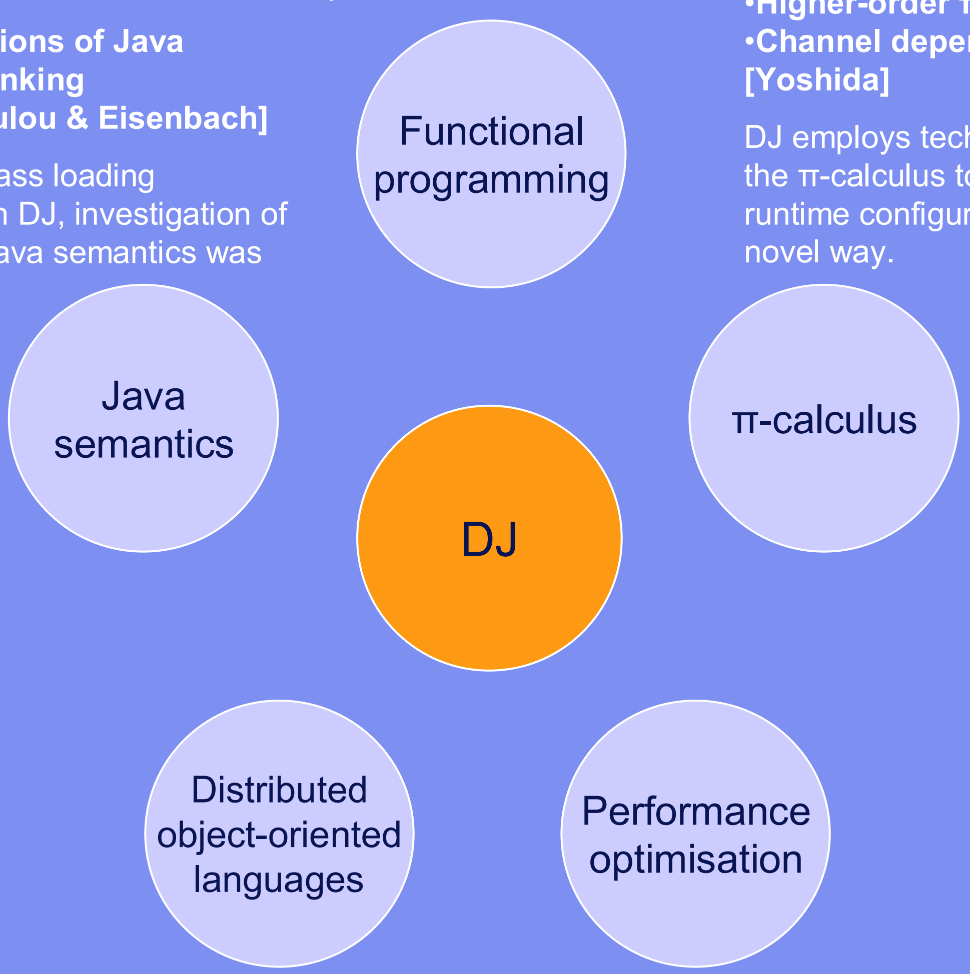
To model class loading accurately in DJ, investigation of the actual Java semantics was required.

•ML, Haskell, Scheme

DJ implements first class functions using code "freezing" and "defrosting" primitives.

•A calculus of mobile processes [Milner, Parrow & Walker]
•Asynchronous π [Honda & Tokoro]
•Higher-order π [Sangiorgi]
•Channel dependent types [Yoshida]

DJ employs techniques from the π-calculus to model runtime configurations in a novel way.



•Emerald [Hutchinson et al]
•Obliq [Cardelli]

These object-based languages support transparent network programming through object mobility. We can encode this kind of mobility directly in DJ.

•Batched Futures [Liskov]
•RMI call aggregation [Yeung & Kelly]

By bundling together several calls to remote sites, we can improve the performance of distributed applications. Using explicit code mobility, we can model this kind of optimisation faithfully in DJ.

EXPLICIT CODE MOBILITY

DJ provides two primitives for *code freezing*, which is analogous to closure creation in a functional language. We provide the *freeze* command to allow a programmer to delay evaluation of an expression, extending the syntax of Java with two new constructs:

```
e ::= … | freeze[t](T x) { e } | defrost(e, e)
```

Creation

$$\mathtt{freeze[eager]}(T\ x)\{e\} \longrightarrow_l \lambda(T\ x).(\nu\vec{u})(l, e, \sigma, \mathtt{CT})$$

Mode of freezing

Parameter to this frozen expression

Fresh names for the identifiers appearing free in this closure

The piece of code that is frozen for later use

The name (IP address) of the location that created this closure

Environment (variables/objects) the closure depends upon

Optional set of classes

Use

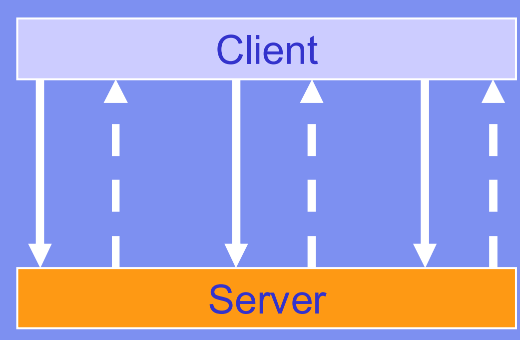
$$\mathtt{defrost}(v, \lambda(T\ x).(\nu\vec{u})(l, e, \sigma, \mathtt{CT}))$$

When we defrost a frozen expression, it is evaluated much like a method call. The formal parameter is substituted for its actual value, and the expression executed.

CODE MOBILITY IN ACTION

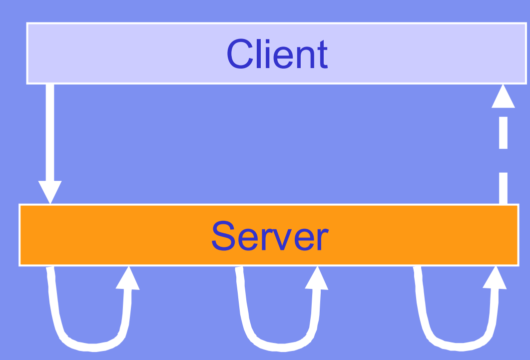
In optimisations for sequential languages, we can aim to improve execution times by removing redundancy and ensuring our programs exploit features of the underlying hardware architecture. In distributed programs these are still valid concerns, but other significant optimisations exist, in particular how latency and bandwidth overheads can be reduced. One typical example of this sort, centring on Java RMI is *aggregation*:

```
int m1(RemoteObject r, int a) {
  int x = r.f(a);
  int y = r.g(a, x);
  int z = r.h(a, y);
  return z;
}
```



This program performs three remote method calls to the same remote object **r**. The return values from each call are unused locally, and are merely passed back to the server in the next call. Hence these three calls can be *aggregated* into a single call, reducing the network latency penalty by a factor of three. We can implement this aggregation using our primitives:

```
// Client
int m1(RemoteObject r, int a) {
  thunk<int> t = freeze {
    int x = r.f(a);
    int y = r.g(a, x);
    int z = r.h(a, y);
    return z;
  };
  return r.run(t);
}
// Server
int run(thunk<int> x) {
  return defrost(x);
}
```



These calls are now *local* to the server

IMPLEMENTATION

DJ can be implemented in terms of source-to-source compilation, taking a program augmented with freeze and defrost and converting this into plain Java source.

To allow eager class downloading, the class loader used in normal RMI programs must be replaced. The new class loader must support the bundles of classes that are sent with our frozen code.

FUTURE WORK

•Code generation and meta-programming in DJ.

•Application to mobile computing platforms.

•Security considerations.

REFERENCES

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

A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. Proceedings of the 20th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005).