Ionic types

Simon Dobson[1] and Brian Matthews[2]

[1] Department of Computer Science, Trinity College, Dublin, Ireland
simon.dobson@cs.tcd.ie
[2] Information Technology Department, CLRC Rutherford Appleton Laboratory, Oxfordshire, UK
b.m.matthews@rl.ac.uk

Abstract. Mobile code systems allow objects to be moved between environments, potentially changing the behaviour of the object through covert dynamic re-linking of its underlying base classes. As such they are members of a wider class of systems for which the satisfaction of code dependencies is a dynamic process rather than one which is determined purely at load-time. These systems exhibit properties which are not well-captured by current typing models. We describe a system of *ionic* object types which capture these effects and allow them to be analysed within a standard object type framework. We show how ionic types improve the modeling of mobile code fragments, including certain aspects of security checking, and sketch a possible ionic extension to Java. As a side benefit we derive a simple and type-safe representation of method delegation
`Subject areas`: type systems, object calculi, mobility, agent systems, delegation.

1 Introduction

Mobile code systems are an exciting development. They allow system designers a new freedom to control the distribution of functionality within a distributed and evolving environment, essential for addressing the performance and reliability concerns of systems distributed on Internet scales[10]. With these new freedoms come security concerns from executing foreign code within a site, including denial-of-service attacks, data corruption and espionage.

One of the benefits of Java is its type-safety: a Java application cannot access memory arbitrarily and therefore cannot cause the sorts of erroneous (or malicious) pointer errors common in C++ code. However, this sort of type safety is only one aspect of the security needed in a mobile code environment: one must also restrict the accesses which untrusted code makes to system resources such as disks, memory and the network. Essentially this involves restricting the environment in which a piece of downloaded code executes, curtailing or removing completely those operations which are deemed to have security implications. Environmental restriction is currently addressed using operating system level ideas such as "sandboxes" or security domains. Their disadvantage is that they can cause incoming code to fail at run-time in a manner not predicted by the type system. Intuitively it seems preferable for code whose requirements cannot be accommodated to fail *before* execution.

The usual view of an application is as a piece of user code depending on a run-time environment providing access to system resources. One important aspect of mobility is that the underlying run-time system and environment may be changed by migration, since different host sites may offer (or not offer!) different implementations of operations with different security constraints or properties. Mobility brings with it an implicit phase of dynamic re-linking of an application whenever it migrates. If we capture this re-linking within the type framework we can model its behaviour at a formal, and thus more predictable and analysable, level.

We have been exploring the notion of *ionic types* which allow functionality to be deliberately extracted from an object while retaining the necessary dependencies. The resulting *ion* may then be resolved against another object providing the necessary functions. Ionic types provide a direct and type-safe model for examining dynamic object completion as encountered in migratory applications, and as a side benefit provide perspectives on constructs such as delegation and method update.

In this paper we present a system of ionic types expressed within the object calculus of Abadi and Cardelli[2]. We introduce ions as variants on standard objects by providing types and operations for removing and resolving functionality. We show how the ideas can be applied to classes. We describe the use of ions in mobile code systems and briefly discuss some other application areas. We contrast ions with other related work and briefly sketch possible implementation strategies before concluding with some directions for the future.

2 Necessary formalism

We present ions through a minimal version of $Ob_{1<:}$, one of the simplest members of the family of object calculi developed by Abadi and Cardelli – readers familiar with the system may want to skip this section. It should be noted that ions have no essential dependence on the features of this system and can easily be generalised to other calculi.

A,B ::=	types	a,b ::=	terms
X	type variable	x	variable
K	ground type	$[l_i = \varsigma(self : A)a_i^{i\in 1..n}]$	object literal
Top	the biggest type	$a.l$	member access
$[l_i \upsilon_i : A_i^{i\in 1..n}]$	object type	$a.l \Leftarrow \varsigma(x : A)b$	member update
$A \to B$	function type	$\lambda(x : A)b$	function literal
		$b(a)$	function application

Fig. 1. Syntax of the $Ob_{1<:}$ calculus

The object calculus is formed using the types and terms from figure 1. An object type is represented by a sequence of labelled members, all labels distinct. An object literal maps labels to values, where a value may include a reference to the object itself. These self-referring members are the *methods* in the object, and are constructed using the ς (sigma) binder. A method term such as $\varsigma(self : O)b$ binds *self* within b to the object on which the method is called. More complex methods may be built using a λ-term for the body of the method. The "dot" operator is used to access the members of an object, and will implicitly bind *self* in any methods accessed. For example the object type $O \equiv [val : Int, get : Unit \to Int]$ defines the type of an object with two members, an integer val and a function get taking no arguments and returning an integer. One possible object with this type is $o \equiv [val = 1, get = \varsigma(self : O)\lambda()self.val + 1]$, where the get member is a method with self reference *self* and a body consisting of a function returning the value of val plus one.

Note that the use of methods does not affect the type signatures of members, since ς-binding is performed implicitly by the dot operator. This has a number of implications, most notably that, using the example above, $o.get$ (with no application) generates an ordinary function whose free self references are bound to the object from which it was accessed.

(Sub Object)
$$\frac{\Gamma \vdash \upsilon_i B_i <: \upsilon_i' B_i'}{\Gamma \vdash [l_i \upsilon_i : B_i^{i\in 1..n+m}] <: [l_i \upsilon_i' : B_i'^{i\in 1..n}]} \ \forall i \in 1..n,\ \upsilon_i \in \{^0, ^+\}$$

(Sub invariant)
$$\frac{\Gamma \vdash B}{\Gamma \vdash {}^0B <: {}^0B}$$

(Sub covariant)
$$\frac{\Gamma \vdash B <: B'}{\Gamma \vdash \upsilon B <: {}^+B'} \ \upsilon \in \{^0, ^+\}$$

Fig. 2. $Ob_{1<:}$ object sub-typing

Object sub-typing is generally invariant: an object type A is a sub-type of another object type B iff A has at least the members of B with the same types. However, methods may be decorated with $^+$ to mark them as covariant[1]. The ordinary form of method declaration, with no annotation, is considered a short-hand for an invariance decoration ^{0}. (Figure 2, where the υ_i represent decorations.) For our work we use covariance solely for encoding classes.

(Type Class)

$$\frac{\Gamma \vdash A}{\Gamma \vdash Class(A)} \text{ object type } A$$

(Sub Class)

$$\frac{\Gamma \vdash Class(A) \quad \Gamma \vdash Class(B) \quad \Gamma \vdash A <: B}{\Gamma \vdash Class(A) <: Class(B)}$$

Fig. 3. $Ob_{1<:}$ classes and class sub-typing

Classes are not regarded as primitive but are encoded as objects: a class for an object type is simply an object which builds instances of that type with common definitions for the members, and there may be several classes generating objects of a given object type. For an object type $A \equiv [l_i : A_i^{i \in 1..n}]$ the corresponding class type $Class(A) \equiv [new^+ : A, l_i^+ : A \rightarrow A_i^{i \in 1..n}]$ consists of a method *new* and encodings for the methods as *pre-methods* – functions from a receiving object type A to the method body. The covariance decorations ensure $A <: B \Rightarrow Class(A) <: Class(B)$ (figure 3). Each member of $Class(A)$ has the form $[new = \varsigma(z : Class(A))[l_i = \varsigma(self : A)z.l_i(s)^{i \in 1..n}], l_i = \lambda(s : A)a_i^{i \in 1..n}]$, where *new* constructs an object where each member invokes the appropriate pre-method on the class. The *new* method models the mechanical process of constructing an object from a class rather than being a "constructor" in the usual sense of object-oriented languages.

3 Ionic objects

In this section we develop a minimal type model for ions.

Let $A \equiv [l_i : A_i^{i \in 1..n}]$ be an object type having members labeled from $L_A \equiv \{l_i^{i \in 1..n}\}$. We may divide this label set into a number of disjoint sets $\{L_J\}_{J \subseteq \{1..n\}}$, each of which induces an object type $O_J \equiv [l_j : A_j^{j \in J}]$. Generally we say that the O_J *cover* A (or equivalently that the L_J cover L_A), since $A <: O_J$ for all the O_J with no extra methods being introduced. We also introduce a type extension operator. If $A \equiv [l_i : A_i^{i \in 1..n}]$ and $B \equiv [l_j : B_j^{j \in n+1..m}]$ are object types, we define $A \triangleright B \equiv [l_i : A_i^{i \in 1..n}, l_j : B_j^{j \in n+1..m}]$ as extending A with the members of B. Note that in this definition the two types share no common labels, so both $A \triangleright B <: A$ and $A \triangleright B <: B$.

Let two label sets L_I and L_D, induce types I and D which cover A. Suppose we have an object $a : A$, and we wish to capture the members in L_I. We cannot simply remove those members in L_D as the bodies of the L_I members will generally depend on the L_D for their definition. We therefore capture the dependency which the members in L_I have on the members in L_D by defining a type $Ion(A, D)$ (read as "ion of A depending on D"). Because all labels are unique, A and D uniquely specify I, and also $A <: D$. The members of such types – the *ions* – represent objects which have some of their members abstracted and which may be replaced by other suitably-typed members. If L_D has no members – so $D \equiv []$, the empty object type usually referred to as *root* – then the ion has no dependencies and is essentially an ordinary object[2]. We refer to A as the *underlying type* of the ion, to I as its *ionisation* type, and to D as its *dependency type*.

The type rules for ions are shown in figure 4. Ionic types are covariant in their members and contravariant in their dependencies: for an ion i' to be substitutable for an ion i it must provide

[1] A variation of $Ob_{1<:}$ provides contravariant methods as well. These are not needed for our purposes.

[2] One could therefore define an object calculus completely in terms of ions, or equate $Ion(A, [])$ with A.

(Type Ion)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D}{\Gamma \vdash Ion(A, D)} \text{ object type } A \text{ covered by } I, D$$

(Sub Ion)

$$\frac{\Gamma \vdash Ion(A, D) \quad \Gamma \vdash Ion(A', D') \quad \Gamma \vdash A' <: A \quad \Gamma \vdash D <: D'}{\Gamma \vdash Ion(A', D') <: Ion(A, D)}$$

(Val Ionise)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D \quad \Gamma \vdash Ion(A, D) \quad \Gamma \vdash a : A}{\Gamma \vdash ionise(a, I) : Ion(A, D)} \text{ object type } A \text{ covered by } I, D$$

(Val Deionise)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D \quad \Gamma \vdash i : Ion(A, D) \quad \Gamma \vdash D' <: D \quad \Gamma \vdash d : D'}{\Gamma \vdash deionise(i, d) : I \triangleright D'} \text{ object type } A \text{ covered by } I, D$$

Fig. 4. Type rules for ions

at least the same new members with *no more* dependencies. The $ionise()$ operation generates an ion from an object given a description of the methods to retain. The $deionise()$ operation resolves these dependencies against another suitably typed object, yielding an object which includes the type of the actual type of the object against which the ion is resolved. This means that if $d : D'$ can be determined statically, and if D' includes a new member $l_k : D_k$ then we can determine statically that $deionise(i, d)$ also includes l_k. This means that clients of the deionised ion may use the new method, even though the methods derived from the ion itself cannot.

There is one subtlety here concerning method overriding. Suppose that we have $i : Ion(A, D)$ and $d : D' <: D$ where D' also includes a member $l_j : A_j^{j \in L_I}$. It is possible that both A and D' may include a member l_j, so $deionise(i, d)$ must select which method body to use in the deionised object. If the method types are incompatible (possible, since A and D' may be sub-types of D down two different sub-typing paths) there is little choice other than to cause a dynamic type error; if the types are compatible, there are several options:

1. Disallow duplicate methods in d and i.
2. Select the method body from i.
3. Select the method body from d.

The first option is over-restrictive. Since the exact type of d cannot be determined statically, this would lead to a proliferation of unexpected dynamic type errors. The second option is attractive in that it allows a member in i to override a member inherited from d, which seems in keeping with traditional object-oriented practice. In the context of mobile code however, this behaviour introduces a gaping security hole whereby a piece of remote code can override an environmental function.

(Eq Ionise)

$$\frac{\Gamma \vdash a \leftrightarrow [l_j = a_j^{l_j \in L_I \cup L_D}] : A}{\Gamma \vdash ionise(a, I) \leftrightarrow [l_j = a_j^{j \in L_I}] : Ion(A, D)} \; L_I, L_D \text{ cover } L_A, \; A, I \text{ not class types}$$

(Eq Deionise)

$$\frac{\Gamma \vdash c \leftrightarrow [l_j = c_j^{l_j \in L_I}] : Ion(A, D) \quad \Gamma \vdash d \leftrightarrow [l_k = d_k^{l_k \in L_D \cup L_X}] : D'}{\Gamma \vdash deionise(c, d) \leftrightarrow [l_j = c_j^{l_j \in L_I - L_X}, l_k = d_k^{l_k \in L_D \cup L_X}] : I \triangleright D'} \; A, D \text{ not class types}$$

Fig. 5. Semantics for ions

An ion will therefore always use the method bodies provided by the object against which it is deionised – option three, and reflected in figure 5. (The side conditions reflect the interactions between ions and classes dealt with in the next section.) The downside of this is that some of the ion's functionality may not be propagated. This is the critical difference between ions and ordinary sub-types and sub-classes: they prevent overriding in a structured manner, but with the disadvantage that some methods in the ions may not show through in the final object. An implication is that d is and remains self-contained and the behaviour of its methods cannot be affected by the addition of methods from i. We shall explore these issues in more detail below.

4 Ions and classes

As mentioned in section two, $Ob_{1<:}$ does not treat classes as primitive. Rather, a class is an object which produces objects of a given underlying object type which share a common implementation of their methods. If X is an object type then $Class(X)$ is the object type of classes generating objects of type X. In this section we explore the interactions of the class and ion constructions.

The first observation is that the class construction may be applied to ionic types in two distinct ways. The first builds a family of types $Class(Ion(A, D))$, representing classes which create ions rather than objects. The second builds a family of types $Ion(Class(A), Class(D))$, representing classes with some functionality extracted. We refer to the former as *ion classes* and the latter as *ionised classes*[3].

An ion class constructs ions in the same way as a class constructs objects: it generates instances of an object type which share a common implementation. Each object created by a class is identical to all others in terms of initial state but independent in terms of future state transitions; similarly the ions created by an ion class are initially identical but may be deionised independently to generate objects which will be functionally distinct despite their common heritage.

An ion class may be encoded in a manner very similar to the usual class encoding. The pre-methods specify the defined parts of the ion as expected, but the constructor returns an ion including some dependencies. Suppose $A \equiv [l_i : A_i^{i \in 1..n}]$, L_D and L_I cover L_A, and $D \equiv [l_i : A_i^{l_i \in L_D}]$. Then $Class(Ion(A, D)) \equiv [new^+ : () \rightarrow Ion(A, D), l_i^+ : A \rightarrow A_i^{i \in L_I}]$ – the pre-methods map an object of the underlying type to the provided body.

Ionised classes are a little more subtle, requiring explicit handling of *new* methods. Ionisation always removes the *new* method, while deionisation synthesise a *new* method reflecting the new class' structure (figure 6). The ionised class therefore cannot be used to construct objects until this functionality has been provided by deionisation, resulting in an ordinary class. Although the rule (Eq Class Deionise) looks complex, it basically just performs sequential composition of the *new* methods with duplicates being overridden by methods from d.

(Eq Class Ionise)

$$\frac{\Gamma \vdash a \leftrightarrow [new = \dots, l_i = \lambda(s : A)a_i^{l_i \in L_A}] : Class(A)}{\Gamma \vdash ionise(a, I) \leftrightarrow [l_i = \lambda(s : A)a_i^{l_i \in L_I}] : Ion(Class(A), Class(D))} \; L_I, L_D \; cover \; L_A$$

(Eq Class Deionise)

$$\frac{\Gamma \vdash c \leftrightarrow [l_i = c_i^{l_i \in L_I}] : Ion(Class(A), Class(D)) \quad \Gamma \vdash d \leftrightarrow [new = \dots, l_j = d_j^{l_j \in L_{D'}}] : Class(D') \quad \Gamma \vdash D' <: D}{\Gamma \vdash deionise(c, d) \leftrightarrow \begin{bmatrix} new = \varsigma(z : Class(I \triangleright D'))[l_i = \varsigma(self : I \triangleright D')z.l_i(self)^{l_i \in L_I \cup L_{D'}}], \\ l_i = \lambda(s : I \triangleright D')c_i^{l_i \in L_I - L_{D'}}, \\ l_i = \lambda(s : I \triangleright D')d_i^{l_i \in L_{D'}} \end{bmatrix} : Class(I \triangleright D')}$$

Fig. 6. Ionisation rules for class types

[3] There are actually two other possibilities, $Ion(Class(A), D)$ and $Ion(A, Class(D))$, which do not have obvious interpretations for arbitrary types A and D.

It is interesting to note the difference between an ionised class and an abstract class. In an abstract class some of the class' methods are left to be defined by the sub-classes; in an ionised class some of the methods are left to be defined by the dependency type, which is effectively an abstracted superclass. This inversion of the normal extension approach means that a designer may apply *the same* extended functionality to *different* leaves in the inheritance hierarchy, whereas the normal inheritance approach would require such shared functionality to be applied to a common ancestor of the leaves – which is typically not possible in systems which are dynamically extended. Ionised classes therefore provide an important foil to the normal inheritance mechanism.

The traditional approach to controlling overriding has been through visibility modifiers such as Java's public and protected. Such modifiers define a single overriding policy for all the code in a system – and when code derives from multiple sources this seems to be a very blunt implement with which to tackle an increasingly subtle problem. With ionised classes, rather than extending a single class under a single policy, we may apply the *same* extension to *different* bases. This adds a layer of flexibility when the extensions are imported from remote sources, since the base may be tailored to suit the source.

5 Applications

In this section we show how ions and ionised classes can be used to model and improve three structures commonly encountered in distributed and mobile code systems.

5.1 Agents

An agent is an object instance which moves between sites according to some itinery. The agent may retain state gathered at each site and possibly use it to determine its future path. At each site the agent requires access to some part of the host site's environment, and the capabilities which each site grants will depend on the agent's source. Typically agents are represented simply as objects, but their environmental dependence means that they are more correctly viewed as ions.

As an example, suppose we have an environment containing an operation which prints an integer on the screen, $Env \equiv [printint : int \rightarrow ()]$. This operation depends critically on location in terms of resource acquisition (different places use different displays) and security (untrusted code may not be allowed to access the screen).

Suppose we define an object type $O \equiv Env \triangleright [i : int, inc : int \rightarrow (), print : () \rightarrow ()]$ representing an object with a counter which can be printed on the screen, and an object $o : O$. O extends the environment Env as it requires the *printint* method to implement the low-level part of its *print* operation.

To migrate o from one site to another we ionise it to capture its environmental dependency, transport using the appropriate wire protocol, and then deionise it against the appropriate local environment (figure 7).

If the destination site decides that the incoming o should not be allowed to access the screen, it may supply an environment without the *printint* operation. Deionising against this environment will fail with a type error without execution. We may contrast this behaviour against the fully dynamic approach where environmental access is provided by system-wide objects. The incoming code's dependence on these objects is poorly captured, so operations which are *known* to be generally disallowed (such as file access by applets) are only caught at run-time. Furthermore some of the objects may be unavailable even though they remain notionally defined (for example Java's `System.out` console stream, which can be a "black hole" under Microsoft Windows), leading to unexpected behaviour.

By capturing explicitly the dependence which an object has on routines which potentially change across environments we provide a hook for improved analysis and control. One may, for example, define the minimal dependence which an agent has on its environment as the dependency type of the ion, and then analyse precisely the impact of different security regimes.

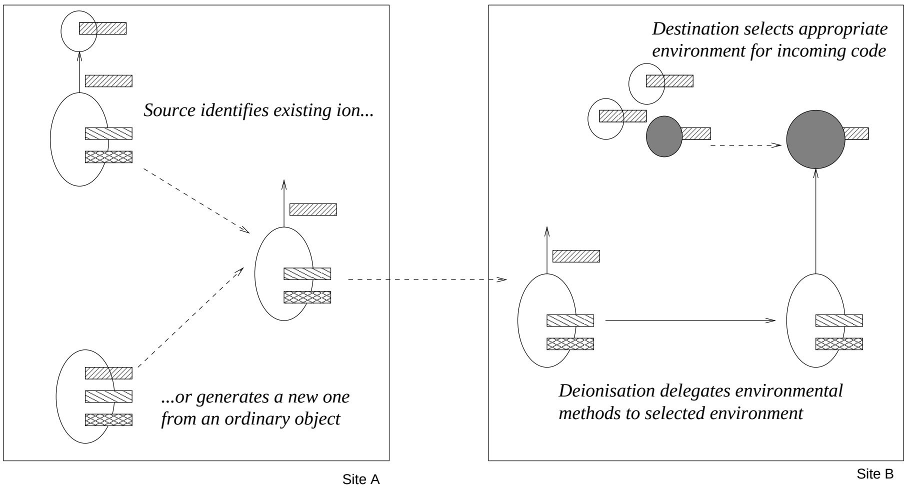

Fig. 7. Ions and migration

5.2 Applets

Java applets are a common form of mobile code. They allow a web page to include a link to a Java applet class, which is downloaded by the browser and executed when the page is displayed. The main difference between an applet and an agent is that, while agent systems move *objects*, applets are *classes* which are instantiated afresh at each site.

Applets generally run in an environment with very restricted disk and network access. These restrictions are enforced by run-time checks inserted into all "sensitive" methods, allowing a browser to intercept and disallow calls which might violate security. For many systems the restrictions will simply disallow all accesses to some methods by applets, throwing a security exception. This form of security is basically a type-checking problem, since a call to such a method will *always* fail at run-time.

We may model this situation using an ionised class. Rather than treat the applet as a fully-fledged class – implying that its execution environment is the same as that in which it was defined – we instead ionise the applet class before transporting it to the client-side. The ionised applet class is then deionised against an environment including exactly those methods which the applet is allowed to call. If the applet requires more methods then this environment will not completely satisfy its dependency type and the deionisation operation will fail *before* the applet is executed.

5.3 Delegation

Delegation is often seen as an alternative to inheritance when building object-oriented programs. In inheritance, the methods of an object are composed by adding new members to an already-existing class, which is then instantiated. In delegation, some of the methods of one object are passed-off to be handled by another object. The basic differences are that delegation involves two objects rather than the one object generated by inheritance, and a single object may be delegated to by several delegates so that they share data (figure 8).

Delegation is particularly important in distributed systems, where an object may wish to extend the functionality of an object on another node. This cannot be captured using normal inheritance. Similarly, in most object-oriented languages delegation is treated as a design pattern rather than as a basic language construct, and must be constructed largely *ad hoc* by each application. However, we may implement delegation directly and simply using ions. We define an ion i whose underlying type include all the available methods and whose dependency type contains the signatures of the methods to be delegated. We then deionise against the object d to which those methods should be delegated. The fact that d is self-contained in $deionise(i, d)$ – so the methods of d remain

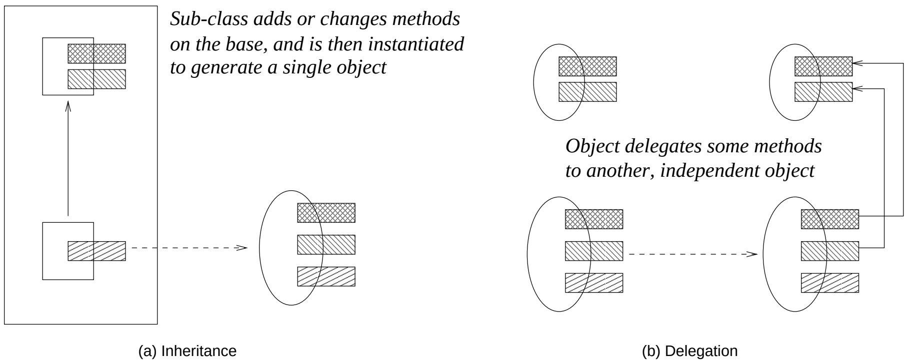

Fig. 8. Inheritance *versus* delegation

unaffected by the addition of i – means that it is perfectly safe to delegate several ions (potentially with different underlying types or extended implementations) to the same object[4].

The ion's underlying type defines the methods which the final object will provide, independent of their source. Delegation does not lead to a loss of type safety, so the ionic types allow a designer to use both delegation and inheritance – the two main paradigms for object extension – as appropriate with the same type safety guarantees.

5.4 Method update

Object calculi (including $Ob_{1<:}$) frequently allow the methods of an object to be updated as computation progresses. In a formal system this allows value and method update to be handled uniformly; in practice it can also be a useful capability, for example in providing dynamic adapotation of key algorithms. Since mainstream languages do not support method update, such features are usualy addressed using (unchecked) design patterns. Ions provide a direct solution, with the additional benefit of providing additional structure to perform multiple updates simultaneously

...bit more TBD

...method re-factoring?

6 Relating ions to Java

How might one implement an ion system in Java? There are several possible solutions, and in this section we sketch perhaps the simplest approach which extends Java minimally (in terms of syntax, semantics and "feel") to provide ionic constructs. The presentation is informal, our intention being to indicate the feasibility of providing ions in Java rather than to fully formalise and extend Java's type system – a task already well performed by others[8].

Java's designers have consistently maintained the significance of class and interface names as well as their structure. This suggests a model of ions where we use interfaces to indicate allowed ionisation boundaries. This respects Java's insistence that all class and objects types are named, which is a little more restrictive than the arbitrary ionisation allowed by our presentation so far. It turns out that this decision also has a slight impact on the definition of deionisation. Furthermore the insistence on all objects being generated from (named) classes makes most of the typings statically determinable.

Consider a Java declaration `class A implements D { ... }` together with an instance `a`. We may ionise `a` at the interface boundary to construct an object `i` having an extended Java type `ion A at D` corresponding informally to our $Ob_{1<:}$ type $Ion(A, D)$ and represented as a copy

[4] If the method bodies from i overrode the bodies from d then the behaviour of a system with multiple delegation to the same object would be non-deterministic, dependent on the deionisation order.

of `a` with the methods from `D` abstracted. We may then deionise `i` against an object `b` of class `B` implementing interface `D` in the expected way. Operationally this delegates the `D`-declared methods of `i` to `b`. The definition of `class A` may include other interfaces as well as `D`, which should be preserved by ionisation.

What is the type of the deionised object? According to our rules it extends `A` with the methods of `B` – but this type is novel, synthetic and without a name, meaning that is cannot be expressed in a Java program. However, we *can* guarantee that it is a legal sub-type of `A`, `B` or `D`. The first assignment "crystallises" the type and constrains future accesses.

```
class A implements D { ... }
class B implements D { ... }
A a = new A();   B b = new B();
ion A at D i = ionise a at D;
A aprime = deionise i against b;
```

Fig. 9. Ionising Java objects

An ion class is declared by providing the definition of its methods assuming that it implements the methods of the interface along which it is ionised. Creating an instance of this class constructs an ion, which may then be deionised to form an object (figure 10, top). Ionised classes are a bit more subtle, as they effectively allow run-time manipulation of the class hierarchy. We introduce a new Java type `ion class A at D` representing a class `A` with methods declared in `D` abstracted, creating such objects using `ionise class A at D` and allowing them to be passed around and resolved (figure 10, bottom). Visibility modifiers behave as expected: a `private` modifier on the base prevents an ion depending on that method, and so forth. Composition of constructors is similarly well-behaved.

```
class ion A at D { ... }
ion A at D a = new A();

ion class A at D Aprime = ionise class A at D;
A a = new (deionise Aprime against class B)();
```

Fig. 10. Ion and ionised classes in Java

How do these constructs support Java-based mobile code systems? We need the ionic operations to preserve special interfaces such as `java.io.Serializable`, so that the ion is serialisable if the object is. Similarly we want ionised classes to be serialisable so that they may be passed between machines. This is possible in Java – outside the type system! – using `Class` objects and the reflection API, so the ionic operations may be regarded simply as giving a typed basis to existing low-level functionality.

This sketch of ions in Java remains incomplete, leaving a number of thorny issues inevitable when trying to integrate a new construct into an existing language – most notably the impact of threads, class loaders, and the handling of interfaces such as `java.io.Externalizable` which could generate subtle errors if applied to ions. We hope we have indicated the ions lie sufficiently close to modern practice to be a viable proposition.

7 Related work

Many groups are working towards the design and implementation of mobile agent systems – a good review may be found in [10]. Most of this work has focussed on mechanisms for migration[9] and on traditional authorisation and verification approaches such as digital signatures, assuming that an agent's environment is in some senses the same across hosts. Treating environmental changes as a phenomenon in their own right provides a different – and in some ways more flexible – view of these issues, allowing them to be treated more uniformly within a type framework.

Another substantial body of work has addressed low-level aspects of mobile systems such as channel usage[12] and encrypted transport protocols[3] using type systems. These efforts may be seen as part of a trend which replaces *ad hoc* checks with richer type checking. Ionic types are simply another aspect of the trend, allowing limited checking of environmental variation due to migration.

The ambient calculus of Cardelli and Gordon[5] is a particularly interesting emerging formalism for mobility. An ambient encapsulates a set of processes and sub-ambients into a package which may migrate through a hierarchy according to some simple rules. Two type frameworks have been developed on top of the original untyped system: a conventional system controlling the types of communications[6] and a novel system of "mobility types"[4] providing type-level control over how an ambient may move about the system. We speculate that communication types are closely related to ions, in the sense that an ion represented as an ambient might add new communications (method calls) if opened within another context. This is an area for further work.

8 Conclusion

We have described a minimal model of ionic types and shown how they can model some otherwise problematic issues in mobile code systems. By capturing the dependencies which an object has on its environment we are able to improve the handling of certain security issues within a type theoretic setting. As a side benefit we also derived a type-safe model of delegation, allowing an important design technique for distributed systems to be provided cleanly with suitable type guarantees. We briefly sketched an incomplete embedding of ions into Java, suggesting that the concepts could easily be reified in a practical programming language.

Existing models of object-oriented systems are designed with a static, monolithic system in mind. With distributed systems, mobile code and policy-controlled access becoming more commonplace, these models are not sufficiently powerful to represent the desirable behaviours. Ions represent a straightforward extension to existing object modeling theory which capture in a type-safe manner the desired behaviour of mobile objects.

As security models become more complex, so does the need to include their behaviour within the type system. New versions of Java, for example, will allow applications to define security domains for incoming code, meaning that the behaviour of a migrating object may be radically affected as it changes domains[1]. One may use ions to model the exclusions necessary for a policy, highlighting those methods which need context-sensitive handling. This can suggest different design approaches for environments, for example separating potentially sensitive functions which manipulate files from their more general counterparts. We are also exploring refinements by which the dependency type of an ionic type can be expanded and reduced piecemeal, to provide finer-grained control over ion dependencies.

A number of implementations of ions are possible, and we have sketched a restricted model close to the spirit of Java. We are exploring other models using the Vanilla language toolkit[7] with a view to determining the extent to which ions provide a widely applicable orthogonal type construction to be integrated into languages for mobile computing. We are also interested in the interaction of ions with native compilation, especially systems such as Harissa[11] which are able to handle dynamic class loading. Intuitively it would appear that ionisation, as basically a type level operation, should have little or no run-time impact, and we would be interested in demonstrating this in practice.

References

1. Secure computing with Java: now and the future. http://www.javasoft.com/marketing/collateral/security.html, 1998.

2. Martín Abadi and Luca Cardelli. *A theory of objects*. Springer Verlag, 1996.
3. Martin Abadi and Andrew Gordon. Secrecy by typing in security protocols. In *Theoretical aspects of computer science*, volume 1243, pages 59–73. Springer-Verlag, 1997.
4. Luca Cardelli, Giorgio Ghelli, and Andrew Gordon. Mobility types for mobile ambients, 1999.
5. Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of software science and computational structures*, volume 1378. Springer Verlag, 1998.
6. Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 79–92, 1999.
7. Simon Dobson. A first taste of Vanilla. Technical Report TCD-CS-1998-20, Department of Computer Science, Trinity College Dublin, 1998.
8. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 5(1):3–24, 1999.
9. Eric Jul, Henry Levy, Neil Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.
10. Neeran Karnik and Anand Tripathi. Design issues in mobile-agent programming. *IEEE Concurrency*, 6(3), 1998.
11. Gilles Muller and Ulrik Pagh Schultz. Harissa: a hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, 1999.
12. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993.