Ionic Types

Simon Dobson[1] and Brian Matthews[2]

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

Abstract. We are interested in the class of systems for which the satisfaction of code dependencies is a dynamic process rather than one which is determined purely at load-time. Examples include dynamic delegation, mobile code and agent systems. Such 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 various forms of dynamic object completion including certain aspects of security checking, delegation and method update, contrast them with other approaches to the same problems, and sketch a possible ionic extension to Java.

1 Introduction

Programming languages exhibit a constant tension between static and dynamic checking. The former attempts to catch problems at compile-time, but inhibits many useful dynamic decisions; the latter provides more run-time flexibility at a cost of run-time type errors. This tension is exhibited by many programming approaches of current interest including dynamic adaptation, code mobility and component-based systems. Designers must balance the need for dynamic flexibility against a desire for static correctness checks.

Such systems are representative of a broad class in which the satisfaction of code dependencies is a dynamic process rather than one fixed at compile- or load-time. This dynamism manifests itself in various guises. In a delegation-based system the object which actually satisfies a method may be re-assigned as computation progreesses. For a mobile code system, it occurs when a piece of code migrates into a foreign environment and is tacitly re-linked with that environment's functions for display, file access *etc.* Under certain security regimes the environment may refuse to provide some expected functions, causing execution to fail at run-time. The important point is that the application depends on the availability of the functions and could *never* execute successfully under that security regime, so many of the dynamic security checks – although required in other, more complex cases – might better be performed statically. Intuitively it seems preferable for code whose requirements can never be accommodated to fail before execution.

The unifying theme is that an object's response to a method can be changed as computation progresses. These changes are not generally arbitrary, but have a well-defined structure and require that some dependencies are maintained. If we capture these structures and dependencies uniformly within the type framework we can model their behaviours more accurately and obtain better predictions of an application's long-term behaviour.

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 implementing the extracted functions. Ionic types provide a direct and type-safe model for examining the dynamic satisfaction of code dependencies in its various guises, providing new perspectives on phenomena including inheritance, mobile agents, delegation and method update.

In this preliminary study 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, and show how the ideas apply to both objects and classes. We discuss some of the issues in static *versus* dynamic checking of ions, and then describe the use of ions in a number of scenarios involving dynamic changes in behaviour. We contrast our work with that of others and briefly sketch a possible embedding of ions into Java, before concluding with some directions for the future.

2 Necessary formalism

We present ions using the system $Ob_{1<:}$, 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
$[l_i v_i : A_i^{i\in 1..n}]$	object type	$a.l \Leftarrow \varsigma(x : A)b$	member update
$A \rightarrow B$	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 $O \equiv [val : Int, get : Unit \rightarrow Int]$ 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. Note also that the usual interpretation of the calculus is purely functional, so assigning to a member generates a new object rather than an update-in-place.

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

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

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

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.

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

[1] A variation of $Ob_{1<:}$ provides contravariant methods as well. These are not needed for our purposes, and we do not explicitly address them: however, there seems no *a priori* reason why they cannot be accomodated.

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

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

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

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.

First we need to formalise some intuitions about the construction of objects. In many cases an object is composed of sets of methods, each implementing a particular aspect or feature of the object's overall functionality. These features are often largely independent but weakly interacting. In some cases it is desirable to adapt the object by replacing one feature without disturbing the others, or extracting one or more features for use elsewhere. It is these manipulations that ionic types seek to model.

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 partition this label set into $\{L_J\}_{J \subseteq \{1..n\}}$, each of which induces an object type $O_J \equiv [l_j : A_j^{j \in J}]$. Each O_J defines a feature which combine to form the final definition of O. 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 $A \triangleright B <: B$.

An object is composed of methods, which in general depend on the existence of other methods. An ion is created by removing some of these methods, giving a "proto-object" which may only be used again when these methods have been replaced. An application constructs an ion by specifying those methods of an object which should be retained in the ion; the type rules track the "missing" methods to guarantee that they are replaced by compatible implementations when the ion is de-ionised back to an object (figure 4).

Let two label sets L_I and L_D induce types I and D covering 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

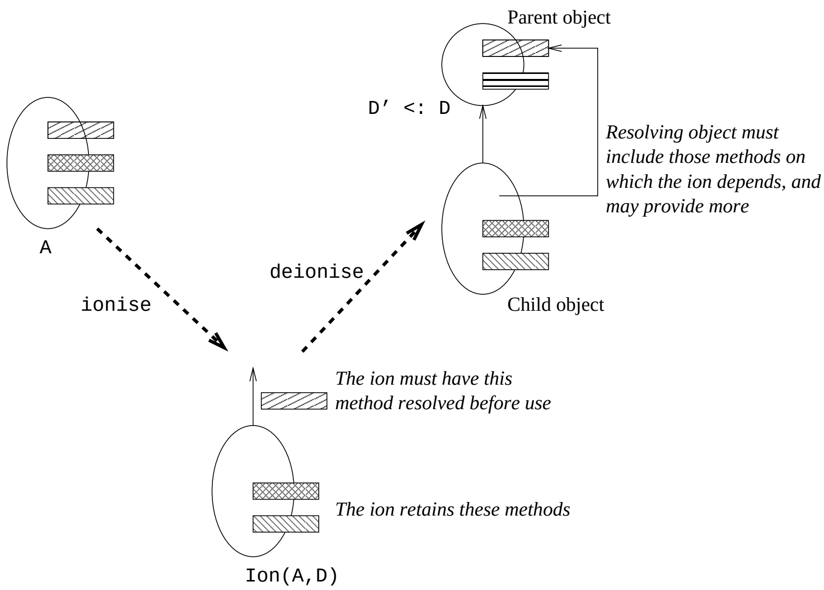

Fig. 4. Ion basics

type $Ion(A, D)$ (read as "ion of A at 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, I as its *ionisation* type, and D as its *dependency type*.

The type rules for ions are shown in figure 5. 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 *at least* the same 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, delegating the methods in the dependency type to this object and yielding an object which includes the actual type of the object against which the ion is resolved.

There is a small subtlety 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. It is also possible that the two method types are incompatible, since A and D' may be sub-types

[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)

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

(Val Deionise)

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

Fig. 5. Type rules for ions

of D down two different sub-typing paths. Since both i and d may contain a definition of l_j we must make three decisions:

1. If a method defined in i invokes l_j, which implementation is used?;
2. If a method defined in d invokes l_j, which implementation is used?; and
3. If a client of $deionise(i, d)$ invokes l_j, what implementation is used?

This problem has arisen in other guises, and most of the combinations have been tried (see section 7). In the current application, we want to use ions to model transparent delegation of functionality to other objects in order to support mobility and other applications. Specifically we want to be able to deionise several ions against the same parent object whilst leaving the parent's behaviour unchanged. This contrasts with most delegation-based languages in which the additional functionality may change the behaviour of the parent object.

We therefore adopt the following model:

1. If the method is defined only in i, invocations from clients of $deionise(i, d)$ and from methods defined in i use the implementation from i – by definition there will be no invocations from d;
2. If the method is defined in i and d, and the two are type-incompatible, a dynamic type error occurs;
3. Otherwise, all invocations use the implementation from d.

It should be noted that point 2 is the only place where ionic types require dynamic checks.

These rules are formalised in figure 6. (The side conditions reflect the interactions between ions and classes dealt with in the next section.) The object d is left completely unaffected by deionisation against it, meaning that several child objects may safely be deionised against a single parent without sensitivity to order. However, some of the ion's functionality may not be propagated into the final deionised object. This is the critical difference between ions and delegation-based systems: they prevent overriding using the type system. It means that an ion can only add functionality to another object, not affect functionality already present. This is essential for modeling mobility using ions: if code from the network is allowed to override system-provided functionality, it opens a gaping security hole. These issues are examined further in section 5.

(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)

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

Fig. 6. Semantics for ions

A further consequence of the rule (Val Deionise) in figure 5 is that the final type of a child object resulting from deionisation reflects the actual type of its parent: the final object is a valid sub-type of the ionisation type, the actual type of the parent object, and (by subsumption) the dependency type. This means that an application receiving an ion may both add the ion's functionality to an object and ensure that the object's extra methods (over and above those in the ion's dependency type) remain accessible. Assuming no name clashes, these extra methods are invisible to the method bodies derived from the ion.

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].

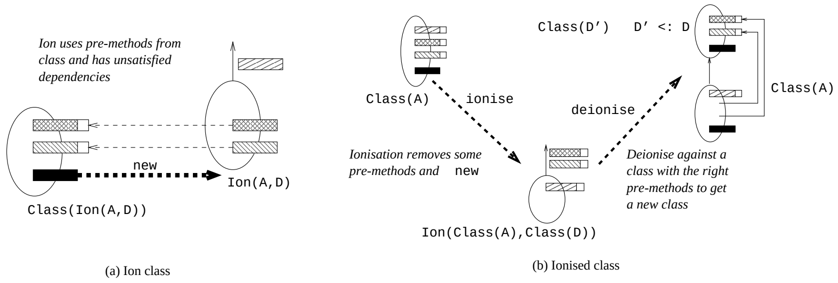

Fig. 7. Ion and ionised classes

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 (figure 7a). 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 $Class(Ion(A, D)) \equiv [new^+ : () \to Ion(A, D), l_i^+ : A \to A_i^{i \in L_I}]$ – 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. Ionising a class removes some of its pre-methods and *new*: the resulting object is then deionised against a class providing the abstracted pre-methods (figure 7b). Deionisation synthesises a *new* method reflecting the new class' structure (figure 8). 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 as expected.

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

[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.

(Eq Class Ionise)

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

(Eq Class Deionise)

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

Fig. 8. Ionisation rules for class types

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`, `protected` and `final` modifiers. Such modifiers define a single overriding policy for all the code in a system – and when code increasingly derives from multiple sources with different degrees of trust 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 (for example) 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 four common programming structures.

5.1 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 adaptation of key algorithms. Since mainstream languages do not support method update, such features are usually addressed using (unchecked) design patterns. Ions provide a direct solution, with the additional benefit of being able to perform multiple updates simultaneously.

The static, inheritance-based approach to method update defines a sub-class with the new method implementations, adding the new implementations from the bottom of the inheritance hierarchy. The dynamic case using ions does the opposite: the object is first ionised to remove the unwanted implementations, and is then deionised against an object providing the new implementations. Since deionisation must satisfy all abstracted dependencies, one may force a set of simultaneous updates simply by controlling the initial ionisation.

This definition of method update retains the old object, dynamically generates a copy with the new functionality, and does not disturb any other objects – a *run-time, selective, preserving* adaptation to use the terminology of [13]. The same approach may be applied to classes, building an ionised class without the unwanted functionality and deionising against a class providing the new implementations. The preservation of the old definition may be undesirable in some circumstances, as it may be impossible to transparently replace the old definition with the new throughout a complex program.

5.2 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 typically depend on the agent's source. Agents are usually represented as objects, but their environmental dependence means that they are more correctly viewed as ions.

When an agent migrates, the host will not typically transfer the entire program: instead it will transfer that part of the program which provides the agent's novel functionality, assuming that the common parts of the run-time system will be made available by the receiving host. This assumption is flawed, in the sense that a host may decide not to make some sensitive functions available to some incoming agents. This in turn means that, if the agent depends on those functions, it cannot successfully execute. However, this dependence remains implicit within the agent rather than being made explicit in its type, making it difficult to determine *a priori* whether the agent has an unsatisfied dependency or not.

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); it is also security-sensitive, in that untrusted code may not be allowed to access the screen. This means that a host may wish not to make *printint* available to all agents.

Suppose we now define an object type $O \equiv Env \triangleright [i : int, inc : int \rightarrow (), print : () \rightarrow ()]$ representing an agent running in an environment and maintaining a counter which can be incremented and printed on the screen. The *print* method of O uses the *printint* method of Env to display the counter on the screen before moving to the next host.

To migrate o from one site to another we ionise it at Env to capture its environmental dependency, transport it using the appropriate wire protocol, and then deionise it against the appropriate local environment (figure 9).

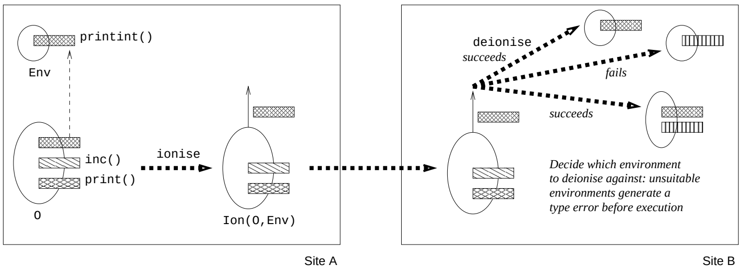

Fig. 9. Ions and migrating agents

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* method. Deionising o against this environment will fail with a type error without execution.

An application receiving an agent may, in addition to running it, wish to provide monitoring "hooks" to track its activities. Ionic types offer a simple way of providing these hooks. The application adds the monitoring methods to the environment (for example by defining an environment $Env' \equiv [printint : int \rightarrow (), numberofintsprinted : () \rightarrow int]$ with a hook to monitor output behaviour). Deionising against this extended environment allows the extra method to show through (a consequence of (Val Deionise) in figure 5), allowing the receiving application to access both the incoming code and its own monitoring and control code.

We may contrast the behaviour of ions against the fully dynamic approach where environmental access is provided by system-wide objects such as Java's use of `java.lang.System`. An incoming object's dependence on these objects is poorly captured, so operations which are known to be generally disallowed (such as file access) 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"), 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.

5.3 Applets

Java applets are a common form of mobile code. The main difference between an applet and an agent is that, while agent systems move objects, applets move 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 manually 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 if the method is called. This simple form of security is basically a type-checking problem, since a call to such a method will *always* fail.

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. This prevents partial completions whose consequences may range from annoying to catestrophic.

This example illustrates the use of ionic types to capture, within the type system, what are effectively load-time link errors. It provides a useful extra "sanity check" for all dynamically loaded classes, not just applets.

5.4 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. Delegation therefore involves two objects rather than one. In most mainstream object-oriented languages delegation is treated as a design pattern rather than as a basic language construct, and must be constructed *ad hoc* by each application.

The simplest case is where a child object delegates to a parent without affecting its behaviour – adding methods but not overriding any. 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 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.

Ions are not a complete replacement for general delegation, however. A general delegation system allows the child object to override methods in the parent. If there are multiple children, the parent must select the method definition from the child on which the invocation originated – the so-called "self problem"[14]. Ions explicitly outlaw this form of behavioural modification – the child's methods cannot override the parent's – so ion-based delegation is only applicable when *extending* the parent's functionality.

Delegation is particularly important in distributed systems, where an object may wish to extend the functionality of another object on a remote server.

Distributed object systems such as CORBA or RMI generate "stub" objects to act as local representatives of remote services. Both inheritance and delegation are problematic in these situations, as the client may attempt to override some of the server's functionality without the server being notified. This means that, while calls to the client will use the overridden method, embedded calls within the server will use the original implementation.

Ions may be used to provide "wrapper" objects around the stub without introducing these problems. The client constructs an ion with the added functionality and deionises it against the stub. The semantics of ions means that the server's behaviour remains unchanged, so the resulting object behaves exactly as expected if the object were local instead of being represented by a stub. This makes it trivially easy for a local object to delegate some of its functionality to a remote object with no "surprises".

6 Ionic types in Java

How might one add ionic types to 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[9].

Java's designers have consistently maintained the significance of the names given to classes and interfaces, 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.

Consider a Java declaration `class A implements D { ... }` together with an instance `a`. We may ionise `a` at the `D` interface boundary to construct an ion `i` having an extended Java type `ion A at D` corresponding informally to our $Ob_{1<:}$ type $Ion(A, D)$ and represented as a clone 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 it cannot be expressed in a Java program. However, we *can* guarantee that it is a legal instance of `A`, `B` or `D`, which is sufficient to allow it to be used.

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

```
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. 10. Ionising Java objects

form an object (figure 11, 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 11, 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. 11. 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

The ability to add functionality uniformly to the bottom of a class hierarchy – usually referred to as *mixin-based* inheritance – has been studied extensively, *e.g.* [4][10]. Mixins arose within a dynamic typing regime, and have been extended with static checks. Ionised classes offer essentially the same functionality as mixins, and are statically type-safe in environments where all classes are elaborated at compile-time. The more general ionic types offer more dynamism with a small and controlled loss of type safety.

The problem of name capture and overriding discussed in section 3 can be resolved in several different ways. Delegation-based languages generally allow the child object to override methods in the parent, changing its behaviour. (An elegant solution to name capture along different sub-typing paths in this context is given by Kniesel[13].) The Beta language[15] prohibits simple overriding but allows a child method to be called within the parent method. Neither approach effectively addresses untrusted downloaded code or transparent delegation to remote objects.

Many groups are working towards the design and implementation of mobile agent systems – a good review may be found in [12]. Most of this work has focussed on mechanisms for migration (*e.g.* [11]) 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[17] 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.

The ambient calculus of Cardelli and Gordon[6] 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[7] and a novel system of "mobility types"[5] 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 a possible area for future work.

Work on design patterns offers a great many useful approaches to designing complex systems. It is interesting to note that many patterns simply provide mechanisms for controlling behavioural updates and indirections – issues which may in many cases be addressed type-safely using ions. This addresses a major criticism of design patterns, that they add layers of complexity to applications without providing additional layers of checking.

8 Conclusion

We have described a minimal model of ionic types and shown how they can model some otherwise problematic issues in systems with dynamic code completion. We showed some applications of the technique to mobile code, simple security checking, delegation and method update. The uniform treatment provides suitable type guarantees to many aspects usually addressed using design patterns or fully dynamic type-checking. 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 no longer sufficiently powerful to represent all the desirable behaviours. The main contribution of this work is to provide a simple model for structured behavioural change which accurately reflects the capabilities, limitations and threats observed in the emerging generation of systems and which allows static and dynamic modifications to be treated uniformly.

Type checking evolved as a way of avoiding as many run-time "sanity checks" as possible. As application domains become more complex it becomes vital to encompass additional properties within the type system, avoiding as far as possible an explosion in avoidable run-time checks (and, by implication, avoidable run-time failures). 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 static exclusions necessary for a policy, highlighting those methods which need context-sensitive handling. This can suggest different design approaches, 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[8] 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[16] which are able to handle dynamic class loading.

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. Martín 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. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOO'90*, 1990.
5. Luca Cardelli, Giorgio Ghelli, and Andrew Gordon. Mobility types for mobile ambients. In *Proceedings of the International Conference on Algebraic and Logic Programming*, 1999.

6. Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of software science and computational structures*, volume 1378. Springer Verlag, 1998.
7. 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.
8. Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis, and John Fuller. Vanilla: an open language framework. In Krzysztof Czarnecki and Ulrich Eisenecker, editors, *Generative and component-based software engineering*. Springer-Verlag, 1999.
9. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 5(1):3–24, 1999.
10. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of POPL'98*, pages 171–183. ACM Press, 1998.
11. 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.
12. Neeran Karnik and Anand Tripathi. Design issues in mobile-agent programming. *IEEE Concurrency*, 6(3), 1998.
13. Günther Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP'99*, pages 351–366. Springer-Verlag, 1999.
14. Henry Lieberman. Using prototypical objects to implement shared behaviour in object-oriented systems. In *Proceedings of OOPSLA'86*, 1986.
15. Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. Addison-Wesley, 1993.
16. Gilles Muller and Ulrik Pagh Schultz. Harissa: a hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, 1999.
17. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993.