Static support for capability-based programming in Java

Vijay Saraswat
Department of Computer Science and Engineering
Penn State University
University Park, Pa 16802
saraswat@cse.psu.edu

Radha Jagadeesan
School of CTI
DePaul University
Chicago, Il 60604
rjagadeesan@cs.depaul.edu

Abstract

We are developing a secure programming language, M, that can be used for programming networked spaces: dynamically extensible, multi-person, networked, persistent, distributed virtual worlds (such as MUDs and MOOs)[Sar98]. Rather than design M from scratch, we are reusing as much of Java technology as possible to ensure that the resulting language is extremely familiar to the mainstream programmer. M features the use of (logic-variable style) promises and vats (cf the recently proposed Java *isolates*, JSR 121) for (data-flow) concurrency (rather than the notorious Java threads), and develops the idea of object-references as *capabilities* for fine grained authorization and access control (within a collection of worlds running in a single JVM instance).

A central problem in capability programming is *confinement*. A (security) principal A may wish to grant a capability c to a principal B only if s/he can be assured that while B can propagate c to objects in its "private" state, B will not propagate c to any other publically reachable object (such as another principal). The capability model has notoriously been held to be incapable of solving this problem (e.g., [WBDF97]) because putatively B is free to send c to any object it has access to, including other publically reachable objects.

We propose to solve this problem by requiring that M programmers use type annotations to specify statically checkable constraints on the runtime object-reference graph. We identify the notion of a *neighborhood* of an object (corresponding intuitively to the idea of private state of an object). We provide a type annotation `confined` that statically approximates the neighborhood. The type-checking rules guarantee that (1) an object has an incoming `confined` reference iff all its incoming references are `confined`, (2) `confined` values (values stored at a `confined` type) can only propagate through `confined` links, and (3) a `confined` object is reachable from at most one object (the *root* of the neighborhood) that has an incoming non-`confined` link (i.e., is a "publically accessible" object). The rules are rich enough to allow (the static approximation of) the neighborhood of an object to be any kind of graph.

We introduce an additional annotation `contained`, and define type rules which allow `contained` values to be propagated only through `confined` references. Thus an object o that receives an object s through a `contained` argument of a method invocation (on a public reference to o) is prevented from propagating s to any object that is not in o's `confined` state, thus solving the confinement problem.

Finally, we motivate two other problems, the *representation confinement* problem and the *authority verification* problem. We show that by introducing the notion of *portals* – confined objects that may release public references to themselves, while ensuring that contained objects are not accessible through these references – both of these problems can be solved.

1 Introduction

There is a growing interest in the design and analysis of object-oriented capability-based languages [ea98] as a basis for secure, persistent, distributed systems such as virtual worlds [Sar98].

Inspired by Carl Hewitt's actors model of computation, such languages seek to take object languages to an extreme of modularity. In such languages, *objects* are considered atomic, encapsulated bundles of state (*fields*) and action (*methods* or *messages*).[1] A *capability* is a reference to an object. The notion of a *subject* from the security literature (i.e. a source of change in the system, "a program in execution") is identified with that of an object. Objects may create more objects. Each object is associated with a programmer-specified *public signature*: the set of public actions that may be taken on the object.

The only operations available on objects are:

- *Creation*: A new object may be created, typically through a primitive operation (e.g. invocation of a constructor)[2].
- *Storage*: A capability may be stored in the field of another, or in a local variable (temporary storage accessible to actions). Capabilities may be duplicated freely.
- *Retrieval*: A capability may be fetched from a field or a local variable.
- *Transmission*: A capability may be sent as an argument in an action performed on another object, or returned as the result of an action.
- *Execution*: A capability may be used to invoke any method in the public signature of the object referenced by the capability.

As a consequence of these operations, a fundamental property of the capability model is (in Mark S. Miller's words) *connectivity begets connectivity*: the only way that an object p may acquire a reference r to an object o is (a) by creating o (b) by receiving r as an argument in a method (constructor) invocation, (c) by receiving r as the return value of a method invocation. Note what is ruled out:

1. It is not possible for an object to manufacture a reference to another object from a "built-in" type (such as integer or string) – no such operation is provided.
2. No primitive operation is provided that will take one capability and transform it into another unrelated one (e.g. via pointer arithmetic).
3. Other than `this`, there are no "static designators for objects" (cf Java's `System.out`, i.e. code designators for specific objects. Thus "code" cannot name a capability (other than one for the "current object").[3]

In the capability model, *everything* is an object. For instance, system *resources* (such as files, streams, sockets) are encapsulated as (certain kinds of) objects, references to which may be handed out as needed. *Security principals* (entities on whose behalf actions are performed) are also internalized as objects; new security principals may be created on the fly. The intuitive notion of *authority* (i.e. the capacity to perform an action, or the ability of a subject to access a resource [MYS03]) may therefore be identified with possession of an object reference.

As [MYS03] points out, such systems satisfy the following properties:

[1] In the capability model, the emphasis is on modularity and encapsulation, rather than on code-sharing and reuse. Thus some aspects of the traditional object-oriented computation model, such as inheritanc and dispatch polymorphism are not emphasized.

[2] Disposal may be performed by garbage collection; we do not assume any specific primitives for this purpose.

[3] Note that in Java `super` is a designator for the same object as `this`, though it is interpreted differently in the context of method invocation.

- *No Designation without Authority*: Designating a resource always conveys its corresponding authority.
- *Dynamic Subject Creation*: Objects may create more objects – the universe of subjects is open-ended.
- *Subject-Aggregated Authority Management:* Instead of resources determining which subjects may access them (e.g. through an access control list), subjects accumulate authorities (capabilities).
- *No ambient authority:* Subjects must select an authority when performing an access.
- *Composability of Authorities:* Resources are also subjects, and may, in turn, possess authority to use other resources, and to dynamically create resources.
- *Access-controlled delegation channels:* An access relationship betwen two subjects X and Y is required in order for X to pass an authority (for perfoming some actions) to Y.

Capability-based systems have a very rich history, going back at least to work at CMU and Cambridge in the 1960s [WLP75,NW77]. From a programming language point of view, a major source of attraction is the cleanliness of the computation model. The fundamental property of capabilities may be termed the "Possession is the Law" property: Any object with a reference to another may invoke the operations in its public signature. No other "approval" is needed. If you have it, you can use it.

1.1 Problems with capabilities

1.1.1 Capability confinement problem

On the face of it, capability-based systems are well-suited to providing a clean framework for secure computation. However, a fundamental problem (originally described in [Lam73] and usually attributed in the context of capabilities to [Boe84,KL87]) has surfaced. Boebert showed in [Boe84] that in an "unmodified capability system" one of the requirements of the DoD security policy, namely the "$\star$-property" could not hold. The example adapted to our current setting is as follows:

Example 1.1 (Information Leakage) Suppose there exist two objects `low` and `high`. Both of them are instances of classes which implement the `ReadWrite` interface:

```
public interface Read {
  Object read();
}
public interface Write {
  void write(Object o);
}
public interface ReadWrite extends Read, Write {}
```

Suppose `Alice`, an object representing a person with a low security clearance, has been given read and write capabilities on `low` (`lowReader` and `lowWriter`), and write capability on `high` (`highWriter`).

Conversely, `Bob`, an object representing a person with a high security clearance has been given read and write capabilities on `high` (`highReader` and `highWriter`), and read capabilities on `low`(`lowReader`).

For the $\star$-property to hold, it must not be possible for `Bob` to get access to `lowWriter` – for then `Bob` is in a position to leak secrets from `high` to `low`.[4]

In a capability system such as Java however, the following sequence of actions will result in this violation:

[4]Symmetrically, it should not be possible for `Alice` to get access to `highReader`.

```
// Alice
lowWriter.write( lowWriter );
// Bob
Object secret = highReader.read();
Writer trapdoor = (Writer) lowReader.read();
trapdoor.write( secret);
```

Intuitively, `Bob` is able to "read" data at a lower classification; but this data is actually a capability, so `Bob`, by being in possession of it, can exercise it, thus performing a prohibited action. □

[KL87] interpretes this result as showing that "designs in which the access rights are checked upon access but are not modified upon capability copying" cannot enforce the $\star$-property. [WBDF97] states:

> [...] an important issue is *confinement* of privileges [Lam73]. It should not generally be possible for one program to delegate a privilege to another program (that right should also be mediated by the system). This is the fundamental flaw in an unmodified capability system; two programs which can communicate object references can share their capabilities without system mediation. This means that any code which is granted a capability must be trusted to care for it properly. In a mobile code system, the number of such trusted programs is unbounded. Thus, it may be impossible to ever trust a simple capability system. [...]
>
> Fundamentally, extended capability systems must either place restrictions on how capabilities can be used, or must place restrictions on how capabilities can be shared. Some systems, such as ICAP [Gon89], make capabilities aware of "who" called them; they can know who they belong to and become useless to anyone else. The IBM System/38[BTR80]associates optional access control lists with its capabilities, accomplishing the same purpose. Other systems use hardware mechanisms to block the sharing of capabilities [KH84]. For Java, any such techniques would be problematic. To make a capability aware of who is calling it, a certain level of introspection into the call stack must be available. To make a capability object unshareable, you must either remove its class from the name space of potential attackers, or block all communication channels that coul dbe used for an authorized program to leak it (either blocking all inter-program memory-sharing or creating a complex system of capability-sharing groups).

Another concrete manifestation of this problem is provided by the following example.

Example 1.2 (Confining Room) Consider a programmable virtual world server. A person (e.g. Alice) may connect using a client, authenticate, and be associated with an object in the server (e.g., `Alice`) serving as a representation of the person within the world. People may enter (chat) *rooms*. A room maintains a list of all objects in the room. Whenever a person says something in the room, this text is communicated to all objects in the room.

Thus, on entering, a person `p` provides the room with a capability, say `tell(p)` that allows the room to communicate text to the person. We now desire that it should be possible for the compiler to verify, through type-checking, that the room will *not* propagate `tell(p)` to any other player. It should be possible to do this even if the room is "owned" by a player, say `Charlie`, inimical to `Alice`, and the code for the room is defined by `Charlie`. How shall this be done? □

In this paper we present an extremely simple and general approach to this problem by exploiting the idea of *static typing*.

Our approach. Our basic insight is that the local state of an object can be defined intrinsically on purely graph-theoretical terms – as the *neighborhood* $n(o)$ of the object o. Intuitively, the neighborhood, $n(o)$ of an object o is the largest set of objects that can be reached from o and is not pointed to by any

object that is not o or in $n(o)$. We show that a simple, purely local, static type system can be defined which approximates neighborhoods.

We introduce a distinction between *free* objects and *confined* objects. Intuitively, a free object is one whose reference can be passed around freely between objects. All objects with a "public persona" (all "world objects" [Sar98]) such as rooms, exits, chairs, tables, cars, player objects etc. that expect to move freely between world objects should be created as free objects. Objects that are intended to stay confined to the private state of an object should be created as confined objects. Objects created free remain free throughout their life; objects created confined stay confined to the same neighborhood throughout their life. (Though, in Section 5, we shall see how such objects may give out public references, thus becoming *portals*, without violating neighborhood encapsulation.) Confined objects are allowed to propagate only through *confined links*, i.e. links that point to objects in the neighborhood, thus preserving the integrity of neighborhoods.

We will find it extremely convenient to label each object with an object. Free objects o are labeled with o. Confined objects p are labeled with the root of the largest neighborhood containing p. A label may be thought of as the value of a `final` field on the object. Thus labels must be established when the object comes into being and cannot be changed or dropped. The programming language may well wish to offer extra functionality to take advantage of this extra information, e.g. determining if two nodes are in the same neighborhood. See Section 5.

On this basic notion it is then easy to define a notion of *contained* object: a contained object p is a free object that is received by a free object o at a contained type, and can be propagated by o only through its confined links. Thus it will stay confined to $n(o)$.

Example 1.3 (Information leakage, revisited) We analyze Example 1.1 as follows. If it is desired that `Alice` not be in a position to freely propagate `lowWriter`, then `lowWriter` must be given to `Alice` at a contained type. `Alice` will then be forced to keep this reference internal to her private state, and will be unable to communicate it to people like `Bob`. In terms of type-checking, we will see that in the sequence:

```
// Alice
lowWriter.write( lowWriter );
// Bob
Object secret = highReader.read();
Writer trapdoor = (Writer) lowReader.read();
trapdoor.write( secret);
```

the very first statement will fail to typecheck – a contained argument `lowWriter` cannot be passed on a method invoked on a contained reference (`lowWriter`). □

Our solution described above will not allow `Alice` to share `lowWriter` with any free object. A more general notion of containment, `contained(k)` would allow for an object to propagate from one one neighborhood to another only if the second was able to demonstrate possession of the "key" k. (For example, all entities at a given security clearance would posses the same key.) Such an approach is currently under development and hence beyond the scope of this paper.

1.1.2 Representation confinement problem

The solution outlined above allows a free object o to maintain internal state. This internal state may not be accessed by any other public object, except through o.

This situation can be quite limiting, however.

Example 1.4 (Vector iterators, [BLS03]) Consider the code for a class implementing a vector. Such a class would like to keep in its internal state some representation of the elements in the vector, e.g. as a linked list. Clearly the cells in the linked list should be considered confined to the vector. However, it also becomes necessary to provide *iterators* to the outside world. Clearly references to iterators should be able to propagate freely and yet iterators should have access to confined state (the linked list). □

Thus there is a need for *portals*: a portal for an object o is a public object that may access objects in $n(o)$. Like objects in $n(o)$, a portal is labeled with o, and may receive references to objects in $n(o)$ to store in its local state. Unlike objects in $n(o)$, a portal behaves like a free object, and may be communicated to free objects. We will introduce type rules that guarantee that a portal can be a portal for a single neighborhood. While there is some flexibility on what rules to introduce which allow a confined object to become a portal, we choose a very simple rule that allows each object to use a static object designator `this` for itself (as in languages such as `Java`) at a "free type". Thus every confined object may, in principle, become a portal for its neighborhood.

1.1.3 Authority verification problem

Another important reason for portals is that they may be used to solve the *authority verification problem*. This problem is best illustrated with an example.

Example 1.5 (Verifying the source of an utterance) Consider `Alice` has entered a room programmed by `Charlie`. `Alice` has no reason to trust `Charlie`. If `Alice` receives a message object from the room, purporting to be from `Bob`, how is `Alice` to verify that indeed this message was created by `Bob`, and not by `Charlie` masquerading as `Bob`? □

Note that this problem is different from the *authentication* problem. Within the virtual world server, a connection from some client will not be established to a person object until the client authenticates (e.g. through some password mechanism, or digital certificates e.g. SPKI/SDSI). Once such a connection has been established, and people objects in the world such as `Alice` are "animated", the question arises about how such an object can determine that a particular message that it has received was actually created by `Bob` and by no one else.

The naive solution would be to provide a method on the message that returns the "authority" that created the message. The authority would be some well-known object `p(Bob)` that "represents" Bob. However, what is to prevent `Charlie` from launching a "man-in-the-middle" attack? Charlie may intercept a message from Bob, invoke this method, obtain `p(Bob)`, create another message (as an instance of a class that `Charlie` has written) that establishes `p(Bob)` as the authority.

Portals provide a solution to this problem. Consider each message from `Bob` to be a portal on $n(p(Bob))$. A receiver of the message may simply check the label on the message to verify the authority for the message; by the rules for portals discussed above this authority cannot be faked.

1.2 Related work

[CDM01] provides a common framework for studying simulations between different mechanisms for access control (capabilities, access control lists, trust management systems etc). In future work we hope to explore how the model presented in this paper compares with the other models analyzed in their paper.

To our knowledge, no other paper has proposed a solution for this problem using static typing or using the concept of neighborhood of an object. [Inc88] proposes the use of "factories" for capability confinement. (This idea also appears as "constructors" in EROS [SSF99].) A factory may be described as a trusted system initialized with some code (written by an untrusted *builder*) and data which may possess only those capabilities that (a) are explicitly passed to it by the user, or (b) are "read-only" (*sensory* capabilities), or (c) point to other factories (*sub-contractors*), or (d) are "holes", capabilities to trusted system services (e.g. compilers). In the framework of this paper, factories need not be assumed as a primitive – factories are *programmable*. Indeed, *each object* is a factory in this sense. The concept of "holes" is not needed because the object is incapable of passing contained information to any external capability. Indeed, our proposal substantially extends factories: a factory (object) may be passed both (a) capabilities to other services (e.g. `Bob`) as well as (b) capabilities that should be held locally. The type system will guarantee that capabilities of the first type are not permitted to access capabilities of the second type.

There has been considerable related work on *alias types* and *confinement types* [MPH00,CNP01,BV01]. These papers are motivated from the *representation confinement* problem. In order to reason about object-oriented code in a modular fashion, it is necessary that the behavior of an object be captured accurately by its specification. Uses of that object can then be verified to be correct by using the specification of the object, rather than the actual code for the object. However, modular reasoning is not possible if the objects used in the representation (e.g. the cell objects internal to a vector) are allowed to "leak" out of the object. Various approaches, such as ownership types have been devised for this problem.

We are still investigating the relationship of our approach to these approaches. Our preliminary thinking is that our approach using neighborhoods and portals provides a simple and general solution that does not rely on the imposition of an external structure (such as a tree of "owners") on the object reference graph. We believe that the fact that any reference from an object q to objects in $n(o)$ must go through a portal on $n(o)$ will enable modular reasoning on objects. We also believe that object-based mechanisms for protection are substantially more robust than code-based (e.g. package-based or inner-class based) mechanisms because they allow code written by different authors at widely different times to co-exist while respecting safety properties. Indeed, with other researchers we believe that for the sake of robust programming types should be interfaces rather than classes; class identity (i.e. the name of the class) should not be important. Such a view accomodates on-the-fly code construction for example – while still insisting on type-safety to implement security. Finally, the simplicity of the notions presented here speak to their generality – there are no issues with respect to making these work in languages with subtyping or delegation or other forms of code-sharing (unlike some of the proposals mentioned above).

Finally, a very important closely related area of work that appears to be orthogonal to our approach is the work on "bunched implication" logics [OP99,Rey02] of Reynolds, Pym, O'Hearn and their colleagues. These logics allow for modular assertions and reasoning about the heap. In future work we hope to exploit these logics and establish a connection with the present paper.

2 Object reference graph and neighborhoods

We start by considering the structure of the the *object reference graph* (ORG). At any state of the computation, the current set of objects constitute a graph which has as nodes objects and as directed edges references from one object to another. Each such object o is assumed to have an edge labeled `this` pointing to itself, with type $t(o)$.

How can such a graph evolve as computation progresses? Given the general discussion above, there are but three basic operations (completely independent of the user program) identified in [MVPS00]:

***insert(v,x)*:** v inserts a new node x, and points to x. This corresponds to the execution of code that creates a new object, and stores a reference to that newly created object in a field of v.

***give(a, b, c)*:** a "gives c to b" by adding $b \rightarrow c$ to E. This operation can be performed only if $a \rightarrow b$ and $a \rightarrow c$ in E, and $t(b)$ has a method that takes an argument that is assignment compatible with $t(c)$.

***get(a, b, c)*:** a "gets c from b" by adding $a \rightarrow c$ to E. This operation can be performed only if $a \rightarrow b$ and $b \rightarrow c$ in E, and $t(b)$ has a method with a return type assignment compatible with $t(c)$.

A node q is said to be *reachable* from a node p if there is a (directed) path from p to q. If there is an edge from p to q, we say that p *points to* q. The set of all objects reachable from o is called $\star(o)$.

Why is there no delete operation? Let us address a natural question that arises at this point. Note that we have not defined any operation for *dropping* a reference. For instance, any programming language of interest will have a way by which fields can be assigned, thus overwriting the previous edge. We can introduce this notion by adding the rule:

***delete(a, b)*:** a drops a reference to b. (The new graph is the same as the old graph except that a specified edge from a to b is deleted.)

Thus in practice the execution of a program will result in the object reference graph transforming according to these four rules. Nevertheless, in this paper we shall only consider graphs obtained by using the first three transformation rules as was done in [MVPS00]. The reason is that such graphs are much easier to analyze and represent a conservative approximation of reachability in the actual runtime graph. The graphs are much easier to analyze because edges are *cumulative*: an edge once introduced is never forgotten. If we can establish that particular edges do not exist in any sequence of graphs obtained by applying the first three transformation rules, then we are guaranteed that these edges do not exist even if we could use the *delete* rule.

[MVPS00] shows that the *accessibility problem* for ORGs – is there an evolution of the current graph such that a given object o can reach a given object p (given the public signatures of all the objects concerned) – is decidable. In this paper we shall introduce extra statically analyzable structure that can be used to ensure that particular kinds of edges will not arise at runtime.

Neighborhoods Is there any structure to the ORG which can help us with the confinement problem? A natural question to ask here is: how does the "local state" of an object appear in the ORG? Consider a `Vector` class which provides methods to add elements to a sequence and remove elements from a sequence. It may perform its task by dynamically creating a list in which to store these elements. Intuitively, the list cells represent the local state of the vector. They are created by the Vector to perform its task, and their existence should really not be observable by any user of Vector.

Let us now consider the notion of neighborhood formally:

Definition 2.1 (Neighborhood) Given a ORG G, the *neighborhood* of an object o is the largest collection of objects $n(o)$ satisfying:

1. If p points to an object $q \in n(o)$ then $p = o$ or $p \in n(o)$.
2. $n(o) \subseteq \star(o)$

□

We say that a collection of nodes N is a neighborhood if $N = n(o)$ for some o. o is said to be a *root* for the neighborhood. The reader may verify that there is no requirement that nodes in $n(o)$ cannot point to nodes outside $n(o)$; indeed it is fairly common and routine for an object to hold an external capability in its internal state.

The notion of neighborhoods is fairly fundamental and robust. Let us discuss a few of its relevant properties.

First, there is a very simple way to determine the neighborhood of an object o, given an ORG G. Consider $\star(o)$, the set of all objects reachable from o. Throw out all objects in this set that are reachable from nodes not in $\star(o) \cup \{o\}$. The set that remains is $n(o)$.

Note that the root of a neighborhood is not necessarily unique. Why? Say that a neighborhood N is *disconnected* if it has no incoming edge from a node not in N; otherwise say that it is *connected*. If N is disconnected, it must contain all its roots. Conversely, if N does not contain one of its roots, then it must be because the root has an incoming edge from a node not in N, and hence N is connected. In this case it is easy to see that N has a unique root.

Proposition 2.1 *A neighborhood is connected iff it does not contain its root. A connected neighborhood has a single root.*

In the following, we will mostly be concerned with connected neighborhoods.

The following properties of neighborhoods may be established in a simple fashion:

Proposition 2.2 *1. A neighborhood may be empty.*

(An object may have no local state.)

2. *If two neighborhoods intersect, then one is contained in the other.*

   Thus, neighborhoods may be nested.

3. *A neighborhood N is contained in a different neighborhood N' iff it is connected and its root lies in N'.*

4. *Let N contain an object o. Then $n(o)$ is contained in N.*

   We shall call this property the transitivity *of neighborhoods.*

3 `confined` annotations

To allow programs to exploit the notion of neighborhoods, we introduce `confined` structure.

We introduce a new kind of edge in the ORG, a *confined* edge. In the diagrams below, confined edges will be colored green and labeled with "`c`". Additionally, each node is labeled with a node, according to the rules given below. (The edges that were present earlier will be called "free" edges, to distinguish them from confined edges. They will be colored black and labeled with "`f`".)

Within the context of a specific programming language such as Java, confined edges may be introduced by introducing a *type annotation*, `confined`, for reference types. Thus, for every reference type `T`, `confined T` will also be a type (and will be called a *confined* type). The type of fields, method arguments, method returns, local variables, for-loop variables etc may be confined types.

```
public class Vector {
   confined Cell head;

   public void add(Object element) {
     head = new Cell(element, head);
     ...
   }

   public Object get() {
     if (head != null)
       return head.element;

  }
   ...
}
```

Definition 3.1 (Confined edge Transformation Rules) The following transformation rules are provided for confined edges:

Confined creation: Let $a(o)$ be a node. Create a new node $b(o)$ and add an confined edge from $a(o)$ to $b(o)$.

$b(o)$ will be a *confined* node.

Free creation: Let $a(o)$ be a node. Create a new node $b(b)$ and add a normal edge from $a(o)$ to $b(b)$.

$b(b)$ will be a *free* node. Below, we will shorten the designator $b(b)$ to just b.

Free Give: Let $a(o)$ be a node, with a confined or free edge to a node $c(o)$. Let $a(o)$ point to b. Then add a normal edge from $c(o)$ to b.

This corresponds to sending a free object in a method invocation on a confined or free object.

Free Get: Let $a(o)$ be a node, with a confined or free edge to a node $c(o)$. Let $c(o)$ point to b. Then add a normal edge from $a(o)$ to b.

This corresponds to receiving a free object in a method invocation on a confined object.

Confined Give: Let $a(o)$ be a node, with a confined edge to a node $c(o)$. Let $a(o)$ have a confined edge to $b(o)$. Then add a confined edge from $c(o)$ to $b(o)$.

This corresponds to giving a confined object in a method invocation on a confined object.

Confined Get: Let $a(o)$ be a node, with a confined edge to a node $c(o)$. Let $c(o)$ have a confined edge to $b(o)$. Then add a confined edge from $a(o)$ to $b(o)$.

This corresponds to receiving a confined object in a method invocation on a confined object.

□

See Figure 2 for a pictorial representation of these rules. The rules include the "basic rules" defined in Figure 1.

Note what is *not* allowed: a confined edge cannot be used to access a node and return it as the result of a method call on a free reference, or send it in as an argument of a method call on a free reference.

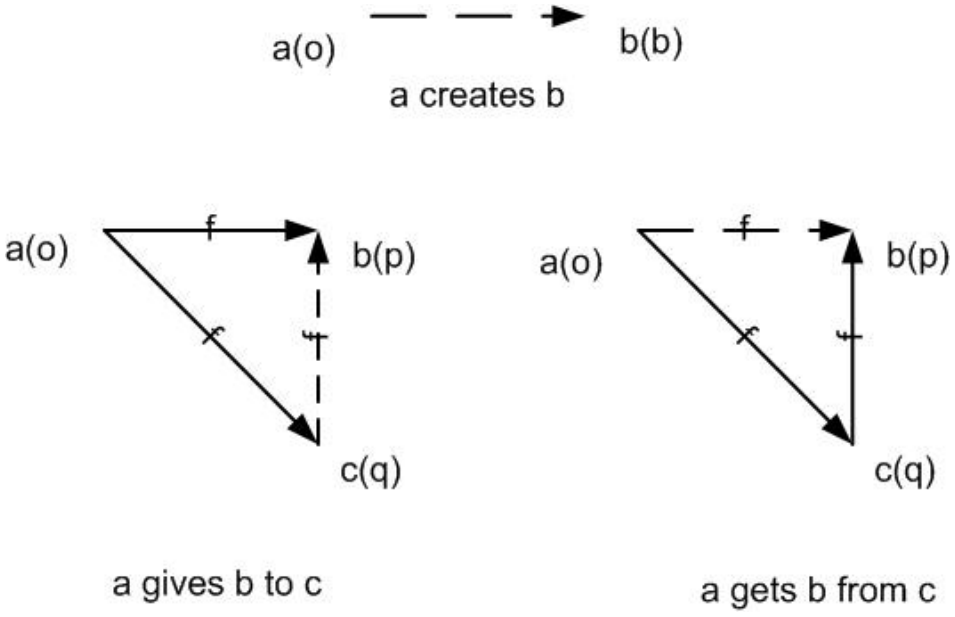

Figure 1: Transformation rules for simple graphs

In the following, let G be the one-point, zero-edge graph whose single node a_0 is labeled with a_0.

Lemma 3.1 *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. Then in any graph G_n if there is a confined edge from a node $a(o)$ to a node $b(p)$, then $o = p$.*

Proof 3.1 By induction on n. There is nothing to show in the base case. There are three cases in the inductive case, one for each rule that introduces a confined edge. The result follows immediately, using the transitivity of equality in the give and get cases. □

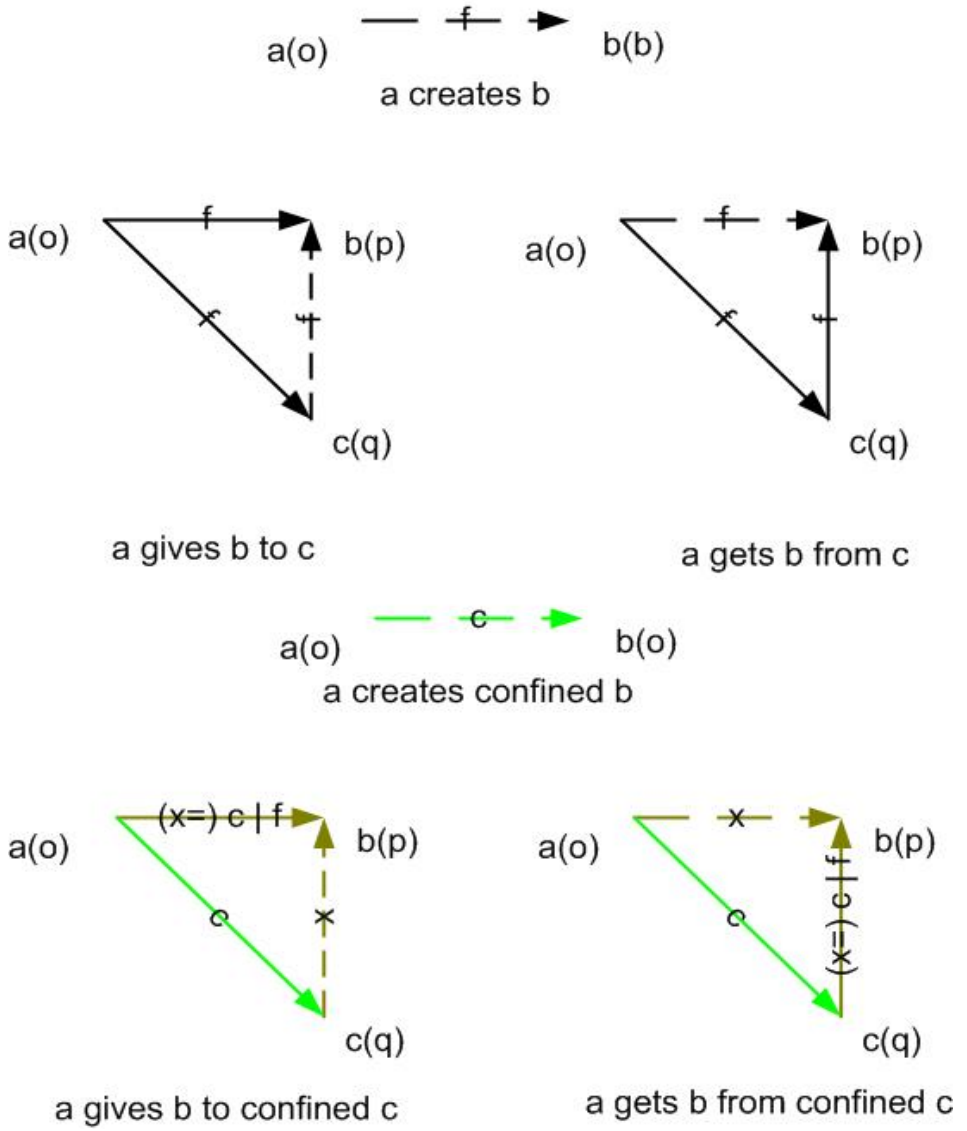

Figure 2: Transformation rules for graphs with confined edges

Lemma 3.2 (Isolation of free and confined nodes) *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. Then for any n, if a node in G_n has an incoming confined edge, then all its incoming edges are confined.*

Proof 3.2 By induction on n. □

Theorem 3.3 (Labels capture neighborhoods) *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. Then for any n, if a node a is labeled with the object o in G_n then $a \in n(o) \cup \{o\}$.*

Proof 3.3 By induction on n.

The base case is obvious – G_0 has a single node a_0 labeled with a_0, and hence the result follows.

Consider the proposition is true for graph G_{n-1}. Let G_n be obtained from G_{n-1} by applying some rule.

The only rules that introduce new nodes are "free creation" and "confined creation". The label of a freely created node is itself, so the condition is satisfied. The label of a newly created confined node b is the label of its creator a. Since b is newly created, there are no other references to it, and hence $b \in n(a)$. We have to show that $b \in n(o) \cup \{o\}$. But by inductive hypothesis, either $a \in n(o)$ or $a = o$. In the former case, by transitivity of neighborhoods $b \in n(o)$. In the latter case, the result follows because $b \in n(a)$.

Consider now that G_n was obtained by applying one of the other four rules. We must show that if $p \in n(o)$ in G_{n-1}, then $p \in n(o)$ in G_n. The only situation we need to be concerned about is if an edge is added to p from outside the neighborhood. But by previous lemmas if p is inside a neighborhood, it cannot have an incoming free edge, and if it has an incoming confined edge then it must be from a node in the same neighborhood. Therefore the step used in getting to G_n cannot add an incoming edge to p from outside its neighborhood. □

Thus a node o is created as either a free node (label equals o) or a confined node (label different from o). Once created as a confined node, it stays a confined node forever.

Do the nodes labeled by o capture precisely $n(o)$? No. Consider for example that o may create a free node p and then not communicate a reference to p to any other node. Now p is in $n(o)$, but is labeled with p, not o. On usual computability grounds, one cannot expect the neighborhood to be captured statically.

4 `contained` annotations

Above we have described type-checking rules that ensure that values statically asserted to lie in the neighborhood of an object will in fact at runtime lie in the neighborhood of the object. Now we show how we can allow a reference to a free object to enter a neighborhood with the static guarantee that the reference will not leave the neighborhood.

We introduce a new kind of edge in the ORG, a *contained* edge. In the diagrams below, confined edges will be colored purple and labeled with "`t`".

Within the context of a specific programming language such as Java, contained edges may be introduced by introducing the type annotation `confined` for reference types. Thus, for every reference type `T`, `contained T` will also be a type (and will be called a *contained* type). The types of fields, method arguments, method returns, local variables, for-loop variables etc may be contained types.

```
public class Cell {
   contained Object car;
   confined Cell cdr;

   public Cell( contained Object car, confined Cell cdr ) {
     this.car = car;
```

```
    this.cdr = cdr;

  }
  public contained Object car() {
    return car;
  }
  public confined Cell cdr() {
    return cdr;
  }
  ...
}
```

We now discuss the transformation rules for contained edges. For the sake of later proofs, we annotate such edges with a set of objects and their labels. Intuitively, the set captures the nodes through which this particular edge has propagated.

Definition 4.1 (Contained edge Transformation Rules) To the rules of the previous section we add:

Contained Give: Let $a(o)$ be a node with a free or contained edge to a node $c(q)$. Let $a(o)$ have a free reference to $b(p)$. Then add a contained edge from $c(q)$ to $b(p)$, annotated with the set $\{c(q)\}$.

This corresponds to sending a free object in a method invocation on a free or contained object, and having the object be received at a contained type.

Contained Get: Let $a(o)$ be a node with a free or contained edge to a node $c(q)$. Let $c(q)$ point to $b(p)$. Then add a contained edge from $a(o)$ to $b(p)$, annotated with the set $\{a(o)\}$.

This corresponds to receiving a value (b) at a contained type as the result of a method invocation on a free or contained object. The receiver must ensure that the value remains contained to its neighborhood.

Give on a Confined Ref: Let $a(o)$ be a node, with a confined edge to a node $c(q)$. Let $a(o)$ have a free, contained or confined edge to $b(p)$. Then add an edge from $c(q)$ to $b(p)$ with the same label. If the edge from $a(o)$ to $b(p)$ is contained and has label s, then the edge from $c(q)$ to $b(p)$ should have label $s \cup \{c(q)\}$.

Get on a Contained Ref: Let $a(o)$ be a node, with a confined edge to a node $c(q)$. Let $c(q)$ have a free, confined or contained edge to $b(p)$. Then add an edge from $a(o)$ to $b(p)$ with the same label. If the edge from $c(q)$ to $b(p)$ is contained and has label s, then the edge from $a(o)$ to $b(p)$ should have label $s \cup \{a(o)\}$.

□

See Figure 3 for a pictorial representation of these rules, together with the previous rules.

Note what is not possible to do: A confined object a cannot be given to a contained reference b – for the neighborhoods of the two are different. Neither can a contained reference b return a confined object a – for this object is confined within the neighborhood of b, not the current object. For similar reasons, a contained object cannot be given another contained object, and a contained object cannot return a contained object.

Theorem 4.1 (Confinement theorem) *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. Then for any graph G_n, if there is a contained edge from a node $a(o)$ to a node $b(q)$ labeled with s, then the label of each object in s is o.*

Futhermore there is no graph transformation that depends on a contained edge from a node $a(o)$ to a node $b(p)$ to introduce an edge into $b(p)$ other than a contained edge.

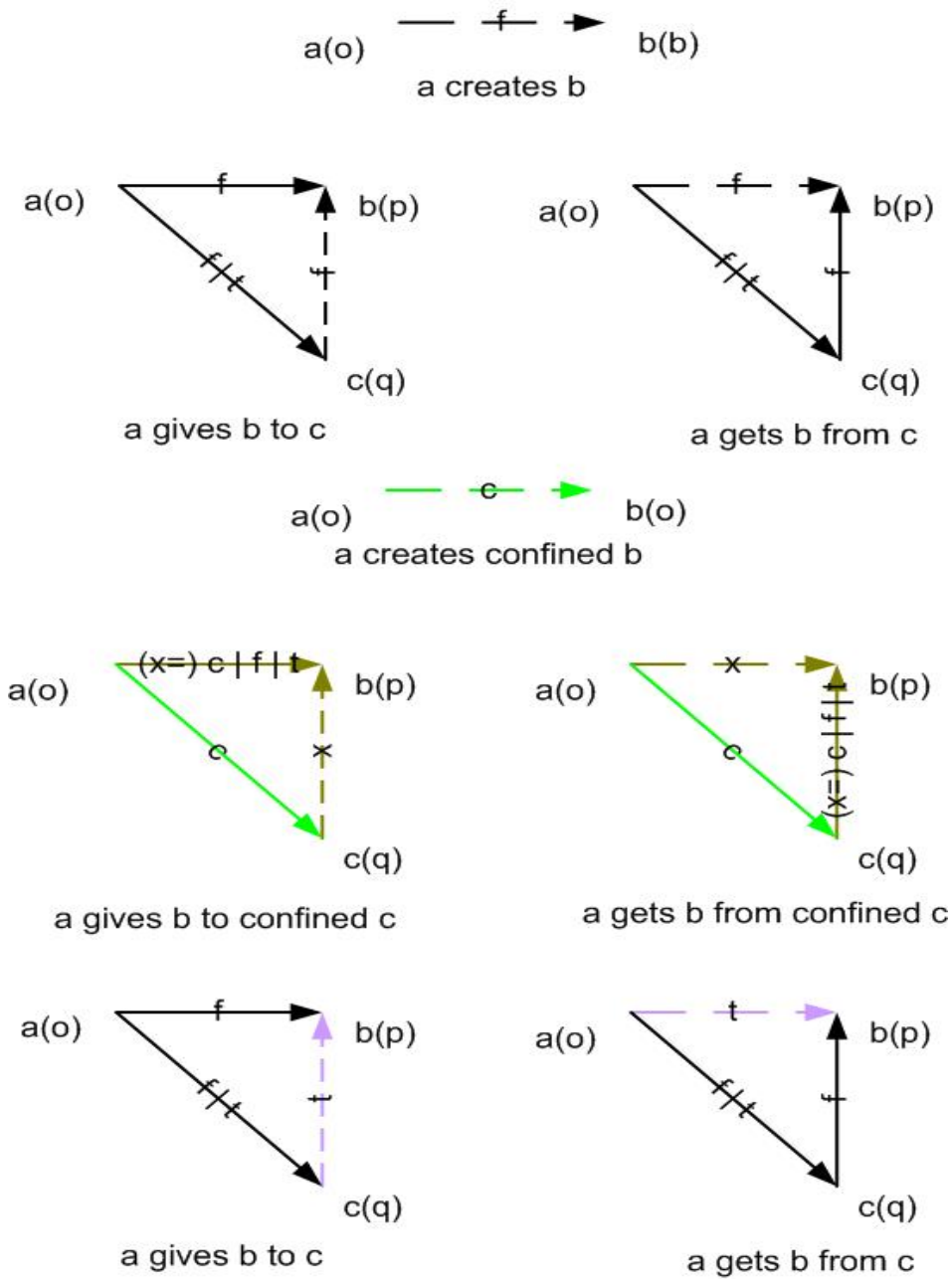

Figure 3: Transformation rules for graphs with contained edges

Proof 4.1 The first assertion is easily proved by induction on n, using the previous theorem. The second assertion follows from inspection of the transformation rules. Note that it implies that if $a(o)$ has a free edge into $c(q)$, and $c(q)$ has a contained edge to $b(p)$, it is not possible for $a(o)$ to use that edge to get an edge into $b(p)$. □

Thus, contained references propagate only within the neighborhood in which they were created. Since `Bob` has a different label from the `room` (because both are different free objects), `Bob` cannot get the reference directly. `Bob` cannot obtain that reference through a free edge into `room` because a contained reference cannot enable any other kind of reference to be generated into its target. Thus `Alice` can be assured that `Bob` will not be able to obtain a reference to `Alice` as a result of the contained reference to `Alice` being provided to the room.

4.1 Room example revisited.

Let us set up a little object structure to make the problem clearer. Let `Alice` and `Bob` be instances of a class `Player`. Let `room` be an instance of the class `Room`. `Player` has the following public method that may be used by others to communicate with it:

```
public void receive( final Message m)
```

Typically, `Message` will have a method (`String toString()`) which returns a text version of the message suitable for display on a screen. `Message` will also have a method (`Authority getAuthority()`) which will return an `Authority` that unimpeachably identifies a public name for the entity responsible for creating the message.[5]

`Room` has a method that allows a player to enter the room; on successful entry, a capability (`SayRights`) to perform certain actions is returned. A player may leave the room at any time, when the player leaves the room, its `SayRight` is revoked.

```
public SayRights Room.enter( final Player p) throws EntranceDenied
public void Room.leave( final Player p)
public void SayRights.say( final Message m)
```

The `say` method behaves thus. The room should throw a `RevokedCapabilityException` if the authority for the message is not in the room. Otherwise it should communicate the message to all the players in the room, using their `receive` method.

How might the room misbehave? The room may fail to deliver a message. We shall not address this problem in this paper – it corresponds to trying to prove some liveness properties of the room code. We are interested in making sure, however, that the room does not *cheat*, that is, communicate the `Player` object to *third parties* (objects which are, intuitively, "not in the rooom"). For, such a third party may then spam the player – sending unwanted communications through the `receive` method.

We may use `confined/contained` as follows. The public signature of the `Room` object is:

```
public SayRights Room.enter( final contained Player p) throws EntranceDenied
public void Room.leave( final contained Player p)
```

Example 4.1 (Room code) Here is the relevant code fragment for the room:

```
confined Collection occupants = new TreeSet();
public SayRights enter(final contained Player p) {
     occupants.add(p); // Collection takes a contained arg here.
}
public void leave(final contained Player p) {
     occupants.remove(p);
```

[5]The programming techniques necessary to define `Authority` in M will be discussed in the next section.

```
  }

   public void say(final Message m) {
     if (! (occupants.contains( (contained Player) m.getAuthority() )))
     throw new RevokedCapabilityException();

     for (confined Iterator i = occupants.iterator(); i.hasNext();) {
       ((contained Player) i.next()).receive( m );
     }
}
```

Here we assume the following signatures:

```
confined Iterator Collection.iterator()
contained Object Iterator.next()
void Collection.add( contained Object r)
void Collection.remove( contained Object r)
```

□

5 Portals

Let us now consider the notion of *multi-neighborhoods*:

Definition 5.1 (Multi-neighborhood) Given a ORG G, the *neighborhood* of a set s of objects is the largest collection of objects $n(s)$ satisfying:

1. If p points to an object $q \in n(s)$ then $p \in s \cup n(s)$.
2. $n(s) \subseteq \star(s)$

□

(Note that $n(\emptyset)$ is forced to be $\emptyset$ by the second condition.)

We say that the set s is the set of *portals* (or *roots*) for the neighborhood $n(s)$. A multi-neighborhood is also sometimes called a *disjunctive* neighborhood because the nodes in the neighborhood are required to be reachable from only one of the roots. Note that the notion of neighborhoods for singleton sets is identical to that for single objects. As before, nodes in $n(s)$ may point to nodes outside $n(s)$.

How shall we exploit multi-neighborhoods? It should be clear that we do *not* want an operation that can combine two neighborhoods to provide a third neighborhood containing the two. For this would mean that neighborhoods could not be used for containment. For instance, `Alice` may enter a room believing that references to `Alice` stay confined in the neighborhood of the room. But if this room later merges with `Bill`, `Bill` will now have access to that reference.

Instead we will explore the idea that a unitary neighborhood may add more portals over time. To this end, we define:

Definition 5.2 (Generated Neighborhoods) Given an ORG G, we say that a neighborhood $n(s)$ *is generated by* an object $o \in s$ if all the objects in s are labeled by o. The objects in s are called *portals*; o is called the *root* of the neighborhood. □

Thus in a generated neighborhood each portal other than the generator can be thought of as a confined node in the neighborhood of the generator that "escaped", i.e., managed to have an incoming free edges from a node outside the neighborhood.

What rule shall we introduce to allow confined nodes to escape? Many solutions are possible. A particularly simple solution is to introduce the *This rule.*[6]

Definition 5.3 (This Transformation Rule) Let $a(o)$ be a node. Add a free edge from $a(o)$ to $a(o)$. □

In a language such as `Java`, the use of the static object designator `this` may be seen as resulting in application of this transformation rule. With the addition of this rule, every confined object has the ability to "leak" itself, e.g. by sending `this` as an argument into a method call on an external capability.

The collected rules are shown in Figure 4.

With this addition to the rule-set, it should be clear that Lemma 3.1 continues to hold. Lemma 3.2 does not hold; the nodes for which it does not hold are, by definition, portals.

Theorem 5.1 (Labels capture neighborhoods and portals) *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. For any n and object $o \in G_n$, let $p(o)$ (the* portals of *o) denote the set of all objects labeled with o that have incoming free edges. Then if a node a is labeled with the object o in G_n then $a \in n(p(o)) \cup p(o) \cup \{o\}$.*

Proof 5.1 By induction on the length of the sequence.

The base case is obvious – G_0 has a single node a_0 labeled with a_0.

Consider the proposition is true for graph G_{n-1}. Let G_n be obtained from G_{n-1} by applying some rule.

The only rules that introduce new nodes are "free creation" and "confined creation". The label of a freely created node is itself, so the condition is satisfied. The label of a newly created confined node b is the label of its creator a. Since b is newly created, there are no other references to it, and hence $b \in n(a)$. We have to show that $b \in n(p(o)) \cup p(o) \cup \{o\}$. But by inductive hypothesis, in G_{n-1} either $a \in n(p(o))$ or $a \in p(o)$ or $a = o$. In the first, by transitivity of neighborhoods $b \in n(p(o))$. In the second case, $b \in n(a)$ and $a \in p(o)$ implies $b \in n(p(o))$. In the third case, $b \in n(o)$, hence $b \in n(p(o))$.

Consider now that G_n was obtained by applying one of the other rules. We must show that if a is in $n(p(o)) \cup p(o) \cup \{o\}$ in G_{n-1}, then so is it in G_n. Clearly, "being o" is a stable property, as is being a portal for o (the application of no rule can change that).

Now assume that a is in $n(p(o))$ in G_{n-1}. It is easy to see that the only way that a might not be in $n(p(o))$ in G_n is if it has an incoming free edge added to it in G_n from a node outside $n(p(o)) \cup p(o) \cup \{o\}$. But then it is in $p(o)$ in G_n, and we are done. □

Theorem 5.2 (Confinement theorem) *Let the sequence $G = G_0, G_1, G_2, \ldots$ describe an evolution of the graph according to the rules above. Then for any graph G_n, if there is a contained edge from a node $a(o)$ to a node $b(q)$ labeled with s, then the label of each object in s is o.*

Futhermore there is no graph transformation that depends on a contained edge from a node $a(o)$ to a node $b(q)$ to introduce an edge into $b(q)$ other than a contained edge.

The proof is unchanged from before.

Thus, contained references propagate only within the neighborhood in which they were created, and its portals. Since `Bob` has a different label from the `room` (because both are different free objects), `Bob` cannot get the reference directly. `Bob` cannot obtain that reference through a free edge into `room` because a contained reference cannot enable any other kind of reference to be generated into its target. Thus `Alice` can be assured that `Bob` will not be able to obtain a reference to `Alice` as a result of the contained reference to `Alice` being provided to the room.

[6] Another rule that could be used is to allow typecasting from a confined edge to a free edge. That is, if there is a confined edge from $a(o)$ to $b(p)$, add a free edge from $a(o)$ to $b(p)$.

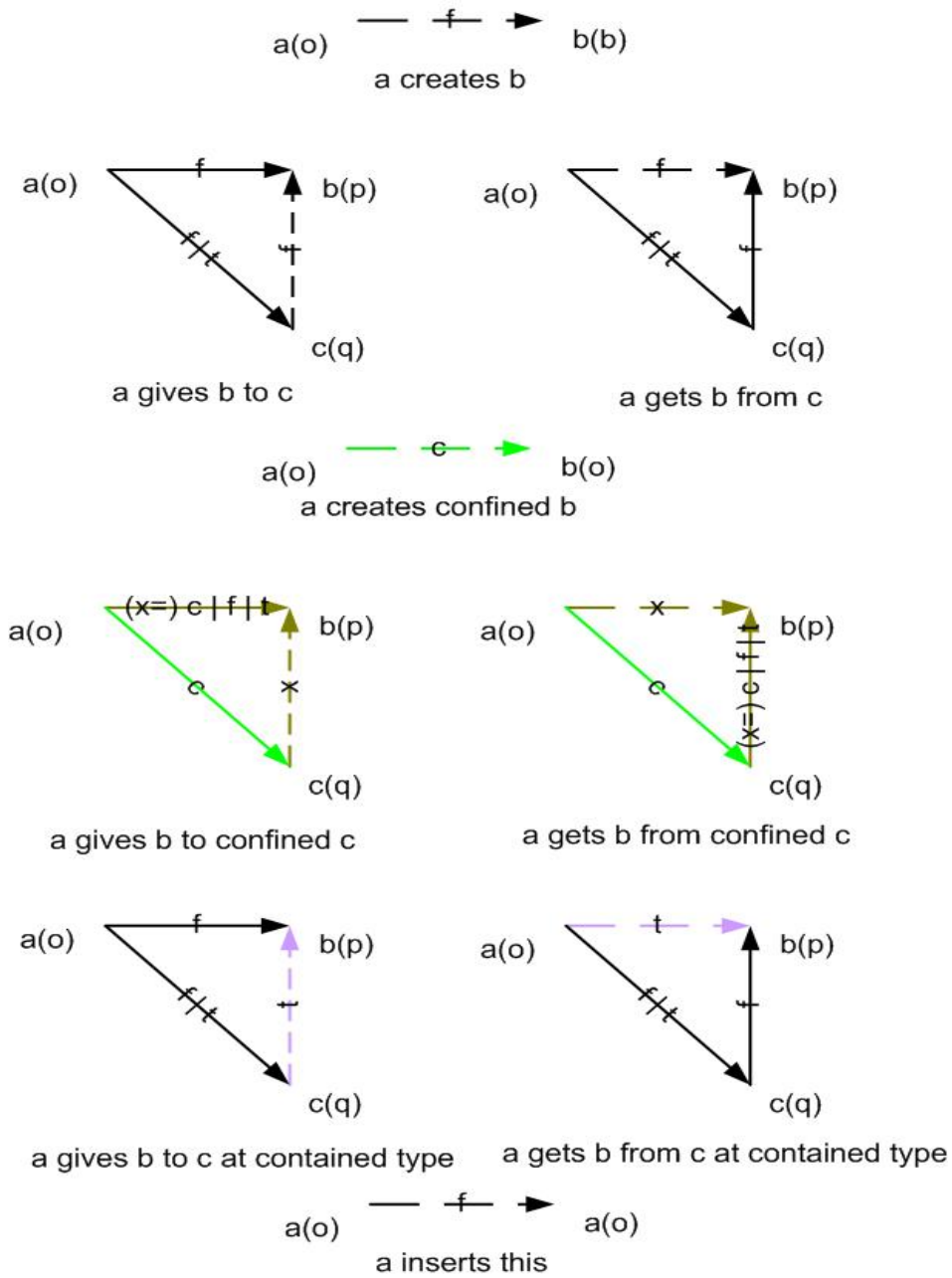

Figure 4: Transformation rules for graphs with contained edges

6 Conclusions and future work

Several issues remain to be explored. The notion of Object Reference Graph seems to provide a convenient, simple, abstract setting within which to explore type annotations devoid of the inessential clutter that arises once a specific syntactic context (e.g. Java) is chosen. It may make sense to augment this notion with a "temporary computational context" (e.g. the current stack frame) in order to model some other aspects of execution.

We believe that the space of useful annotations based on the ORG deserves significant further study. For instance, several other useful annotations may be defined to support patterns of capability programming:

1. The `sole` annotation may be used to specify that this variable contains the sole reference to this object. This is one way that objects created outside a neighborhood could be added to that neighborhood.

2. The `flow` annotation may be used to specify that an object reference passed at that type may not be stored in object fields. Thus it may be stored only in local variables, or passed as a parameter at a `flow` type to other method invocations, or returned at `flow` type from a method invocation.

3. The `opaque` annotation may be used to specify that an object reference may not have any methods invoked on it. The reference may be stored in fields, retrieved from fields and communicated through give/get actions. Storage classes such as vectors and cells often treat their input items in this way.

4. The `other(o)` annotation on a type, where `o` is a final variable of that type, may be used to constrain values passed at that type to be other than `o`. This is of particular importance when it is necessary to specify that a method body does not propagate `this` in a method call, or return `this`. (The corresponding type would be annotated with `other(this)`.)

On a different front, other methods for defining protection domains in Java, such as stack walking [WAF00], should be examined and compared with the scheme proposed in this paper.

Finally, as mentioned in the related work section, the considerable recent literature on the "rep confinement" problem needs to be related to the present work.

Acknowledgements. We wish to thank the many members of the `e-lang` mailing list for their interest in the topic of this paper.

References

[BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL '03. Proceedings of the 30th ACM SIGPLAN-SIGACT on Principles of programming languages*, New York, NY, USA, 2003. ACM Press.

[Boe84] W.E. Boebert. On the inability of an unmodified capability machine to enforce the $\star$-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 291–293, September 1984. See http://zesty.ca/capmyths/boebert.html.

[BTR80] V. Berstis, C. Truxal, and J. Ranweiler. System/38 addressing and authorization. Technical report, IBM, 1980.

[BV01] J. Bokowski and J. Vitek. Confined Types in Java. *Software Practice and Experience*, 31(6), 2001.

[CDM01] A. Chander, D. Dean, and J. Mithell. A state-transition model of trust management and access control. In *Proceedings of the 14th Computer Security Foundations Workshop*, pages 27–43, 2001.

[CNP01] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

[ea98] Mark S. Miller et al. E: Open source distributed capabilities, 1998. http://www.erights.org.

[Gon89] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[Inc88] Key Logic Inc. KeyKOS – A Secure, High-Performance Environment for S/370. In *Proceedings of SHARE 52 I*, pages 3–17, March 1988. `www.cis.upenn.edu/~keyKOS/Key370/Key370.html`.

[KH84] P. Karger and A. Herbert. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, 1984.

[KL87] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2):202–207, February 1987.

[Lam73] Butler Lampson. A note on the confinement problem. *CACM*, 16(10):613–615, October 1973.

[MPH00] P. Muller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. In *ECOOP Workshop on Formal Techniques for* Java *programs*, 2000.

[MVPS00] Rajeev Motwani, Suresh Venkatsubramaniam, Rina Panigrahy, and Vijay Saraswat. On the decidability of the accessibility problem. In *ACM Symposium on the Theory of Computing*, 2000.

[MYS03] Mark S. Miller, Ka Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical report, Combex, Inc., 2003.

[NW77] Roger Needham and R. Walker. The cambridge cap computer and its protection system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 1–10, 1977.

[OP99] P. O'Hearn and D. Pym. The logic of bunched implications. *Bull. of Symbolic Logic*, 5(2):215–243, June 1999.

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS '02. Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE, July 2002.

[Sar98] Vijay Saraswat. Design requirements for network spaces. In *Proceedings of the Virtual Worlds and Simulation Conference*, January 1998.

[SSF99] J. Shapiro, J. Smith, and D. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[WAF00] D. Wallach, A. Appel, and E. Felten. Safkasi: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[WBDF97] Dan Wallach, Dirk Balfanz, Drew Dean, and Edward Felten. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[WLP75] William Wulf, Roy Levine, and Charles Pierson. Overview of the hydra operating system development. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 122–131, 1975.

A Rules for type-checking `confined/contained` extensions to Java

In this section we provide more conventional type rules for `confined` and `contained` type annotations as an extension of the Java type rules. We shall not formally justify these rules with respect to the ORG system presented in this paper, leaving that for future work.

1. The LHS of an assignment is a variable of confined type iff the RHS is an expression of assignment-compatible confined type or an constructor invocation of assignment-compatible type.[7]
2. A method invocation expression `p.m(v1, ..., vn)` is of type `contained T` iff
   (a) `p` is an expression of confined type and `m` is declared to return at a contained type assignment-compatible with `T`, or,
   (b) `p` is an expression of free or contained type and `m` is declared to return at a free type assignment-compatible with `T`.
3. A method (or constructor) invocation expression `p.m(e1, ..., en)` is syntactically correct iff
   (a) `p` is of free or contained type, all the `ei` are of free type, and all of the arguments in the definition of `m` are of contained or free types
   (b) `p` is of confined type, and an `ei` is of contained (confined) (free) type iff the declared class for `p` specifies that the corresponding argument of `m` is of an assignment-compatible contained (confined) (free) type.

Note that Case 3a covers two important cases. Method invocations on an object `p` of a contained type are allowed, as long as all information passed in is free.

Second, method invocation on an object `p` of free type may pass a free value in an argument declared to be of contained type. The receiving object will confine this value to its neighborhood.

The second case allows a free reference to enter a protection domain, under the static guarantee that it cannot leave that domain. And it highlighs why we need to distinguish confined from contained. It is *not ok* for `p` to be a free type and an argument to be defined at a confined type – for the method body for `p` expects to see an object in $n(p)$ in that argument ... and the current object cannot have any reference to an object in $n(p)$ because it holds a free reference to p.

Conditions for the other kinds of statements in Java may be defined in an analogous fashion. For instance,

1. In any class T, the expression `this` is assumed to be of free type T.
2. If a method `m` is declared to return a confined (contained) type, then all `return` expressions in its body must return arguments at an assignment-compatible confined (contained) type. Conversely, if in `return t`, `t` is of a confined (contained) type, then the method must declare that it returns (an assignment compatible) confined (contained) type respectively.
3. A classcast `(confined T) t` (`(contained T) t`) is type-correct statically iff `t` is an expression of a confined (contained) type. The compiler may then replace this classcast by the runtime cast to `T`.
4. An `t instanceof confined T` (`t instanceof contained T`) is type-correct iff `t` is statically declared to be of a confined (contained) type. The expression is treated at runtime as if it were the expression `t instanceof T`.

These rules are easy to check locally, on a per class basis, as an integral part of Java's type-checking algorithm.

[7] An object is created confined iff the corresponding constructor invocation occurs in a confined context.