Programming in Paragon

Bart van DELFT, Niklas BROBERG, and David SANDS
Chalmers University of Technology

Introduction

This tutorial introduces *Paragon*, a programming language which allows programmers to express, as an integral part of the code, security concerns about the data that is manipulated. The Paragon compiler will only allow a program to be run if it is guaranteed to respect the security policy declared for its data. In this sense Paragon promises that well-typed programs are secure by construction.

But what security policies might we want for our data? Access control mechanisms are perhaps one obvious way to control security, and Java (on which Paragon is built) includes APIs to express fine-grained access to security-sensitive resources. But access controls, while useful, are often a poor tool to express the end-to-end security requirements that we actually desire from applications.

For example, consider an "app" which sends you special offers from your local florists in advance of the birthdays of your friends. To function the app needs access to at least your calendar (to retrieve birthdays), the network (to retrieve the latest special offers from florists) and your geolocation (to determine which florists are located nearby). But any app with these access permissions can freely send the whole of your calendar or your location to anywhere on the net. What we want is to grant access (since it is necessary) but limit the *information flows*. In this case we want to limit the information flows from the calendar to the network despite granting access to both.

Paragon allows the programmer to express such concerns directly in the program, by labelling data with policies describing where, and under what conditions, the data is permitted to flow. Moreover, the Paragon compiler checks, at compile time, that the intended information flows are never violated. This helps programmers check their own code for information flow errors, and can be used to ensure that third-party code respects the policy of the data to which it is given access.

For whom is this tutorial written? Several academic publications on Paragon and its policy specification language *Paralocks* have been written [3,4]. To better understand the technicalities behind the language design, and the historical context of this work the reader is referred to those articles. This document, however, is written for the programmer, not the researcher; the programmer, as the end user of the programming language, should be able to write information-flow secure programs without having to learn all of the theories, enforcement mechanisms and meta-properties underlying Paragon, nor indeed the field of information flow research itself. Since Paragon builds on Java, we assume that the reader is reasonably conversant with the Java programming language.

Overview To demonstrate exactly where and why current programming languages lack the enforcement we want, we start with a section containing a collection of Java programs that demonstrate how easily introduced bugs can violate the desired security policy of an application. In Section 2 we encounter our first examples of small Paragon programs; we see how we can attach security policies to data and let the Paragon compiler determine if the program violates these policies. We then look into the definitions of the policies themselves (Section 3), and see how we can roll our own policies. We delve further in the policy specification language in Section 4 and see how we can define more dynamic policies by specifying the conditions under which information is allowed to flow. Section 5 rises above the world of small example code fragments to discuss the features of Paragon that allow for modular programming. For the final deployment and distribution of Paragon programs, Sections 6 and 7 discuss how we can adopt Java's library oriented programming style, as well as provide hints on challenges found in practical applications.

Digital version On the Paragon project's home page an up-to-date version of this tutorial can be found in digital form [1]. What is more, this online tutorial allows you to *compile* as well as to *edit* the example code listed in this document. We encourage you to use this interactive version of the tutorial, as it will give you a better understanding on the Paragon programming language.

1. A Programmer's Perspective on Information-Flow Control

This section motivates what merits an information-flow aware language such as Paragon can offer over conventional programming languages. In particular, we discuss unintentional errors a programmer might make that violate the information flow policy of a program, and could have been prevented by programming in Paragon. If necessary Paragon can also be used to reject programs that intentionally violate the information flow policy of a system; an application that we discuss in Section 6.

Consider the following Java code fragment. The fragment is an excerpt of a blogging web site where visitors can leave their name and a comment. These comments will be displayed to other visitors, so in order to prevent cross-site scripting (XSS) attacks[1] the comment is escaped before it is stored in the database. Implicitly the programmer is trying to enforce the policy "all user input needs to be escaped before it is displayed on the web site".

```
public void postComment(HttpRequest userInput) {
    String name    = userInput.getParameter("name");
    String comment = userInput.getParameter("comment");

    String esc_comment = escapeHTML(comment);

    storeComment(name, esc_comment);
}
```

The Java compiler raises no errors and the code is deployed, now allowing visitors to comment on the various blog posts. Unfortunately XSS attacks quickly start to infiltrate

[1] In a cross-site scripting attack a malicious visitor of the web site tries to e.g. inject HTML or JavaScript code in web pages viewed by other visitors to steal their personal information.

the blog's comments. Despite the programmer's intentions, she did not keep careful track of all information flows and missed that the name of the commenter ends up unescaped in the database. Since the information flow policy is implicit, and not explicitly present in the program, it is impossible for the Java compiler to warn about any violation of this policy.

The programmer corrects the code to the following, now escaping the variable `name` as well:

```
public void postComment(HttpRequest userInput) {
    String name    = userInput.getParameter("name");
    String comment = userInput.getParameter("comment");

    String esc_name    = escapeHTML(name);
    String esc_comment = escapeHTML(comment);

    storeComment(name, esc_comment);
}
```

The updated code fragment has no noticable effect; the XSS attacks persist. The programmer's fix was incomplete, since it is the unescaped user input from the variable `name` that ends up in the database, rather then the intended escaped information stored in `esc_name`. Again, the Java compiler detects no issues with this program (apart from some unused variable) because the information flow policy cannot be made explicit.

The programmer updates the code fragment again, now sending the escaped name into the database:

```
storeComment(esc_name, esc_comment);
```

Even this seemingly final version does not necessarily guarantee the absence of security vulnerabilities in the system. The (potentially different) programmer of the `storeComment` method might be assuming that the arguments provided are already escaped to be free from SQL injection attacks[2]. The Java programming language, or any other conventional programming language, does not give us the opportunity to provide any context information that might allow the programmer or the compiler to realise that we are violating the desired information flow policy.

There are more ways, and more subtle ways, in which a programmer can accidentally introduce undesired information flows. Consider the following code, from another Java-driven web site. As is common for login checkers, the web site presents you with the same error message regardless whether the login name exists or not – this prevents an attacker from learning whether the account he tries to access actually exists.

```
public void login(String username, String password) {
    String correctPass = Database.lookupPass(username);
    if (correctPass == null)
        error("Username/password is incorrect");
    else if (!correctPass.equals(password))
        error("Username/password is incorrect.");
    else  // ... perform login ...
}
```

[2] If a string is inserted unescaped in a database query, it might alter the structure and effect of the query itself.

Accidentally, the programmer returns two different error messages after all. One of them has a period at the end and the other does not. Even without access to the source code an attacker would quite readily understand the difference between the two error messages. Again, the programming language does not allow us to specify what information flows we want to prohibit, making it impossible to detect the policy violation automatically.

As a final example, consider this code fragment coming from a social networking web site:

```
public void sendMessages(User from, User to) {
    to.receive(from.getMessages());
}
```

There is no way to judge the correctness of this code fragment without being aware of the information flow policy of the application where this code appears in. For example, it might be the case that only users that marked each other as friends can share messages, and the code should have read:

```
public void sendMessages(User from, User to) {
    if (friends(from, to))
        to.receive(from.getMessages());
}
```

The various examples in this section each display an undesired information flow which could easily arise by small programmer's mistakes. None of the bugs can be captured during compilation, since there is no explicit information flow policy to comply with in the first place.

What the following sections show, is that the Paragon language allows programmers to make the information flow policy an explicit part of the program. The Paragon compiler understands the policy annotations and checks that implementations comply with them. Therefore programmers' mistakes such as those listed in this section can be detected already at compilation time, preventing them from showing up only when the system is being tested, or worse, when the system is already deployed and the bug is being exploited.

2. Static Lattices and Information-Flow Varieties

Paragon allows us to specify a program's intended information flows, and the compiler verifies that the intended flows will never be violated. This is achieved by *labelling* data with *policies* which describe to where, and under what conditions, the labelled information may flow.

To understand how this works in Paragon we must understand how to specify *where* information may flow, how to *construct* policies, and how *conditional information flow* is specified.

In this section we focus on the first of these three concepts: *where* information may flow. So for now, policies will be just abstract entities and we will not concern ourselves with how they are created.

Comparing Information Flow Policies Paragon policies are a generalisation of the classic concept of security in the presence of multiple levels of security clearance. To make things concrete we will consider the simplest example, confidentiality policies in which data is labelled according to two confidentiality levels: `high` confidentiality (a.k.a. secret) and `low` confidentiality (a.k.a. public). Here `high` and `low` are Paragon objects of a special type named **`policy`**. But `high` and `low` themselves are not special built-in policies, and we will see how to define them (in class `HighLow`) in the next section. The key property of `high` and `low` is the relation between them. Data labelled with `high` should be handled more restrictively than data labelled with `low`. Data labelled `high` – secrets – may not flow to containers labelled `low`. But the reverse is permitted – `low` data may flow to containers labelled `high`. For this reason we say that `high` is more restrictive than `low`.

In addition to the relation between policies which describes when one policy is more restrictive than another, there are a few other basic operators and policies:

- Given two policies p and q, $p \sqcup q$ is the most liberal policy which is at least as restrictive as both p and q. This means that when we combine data labelled with p together with data labelled with q, $p \sqcup q$ is a description of the policy of the resulting data.
- Given two policies p and q, $p \sqcap q$ is the most restrictive policy which nevertheless allows everything that both p and q allow. For example, if a methods modifies some data labelled with p, and some data labelled with q, then calling the method will result in information flow to level $p \sqcap q$ (and higher).
- Of all policies, there is a most restrictive policy (top) and a least restrictive policy (bottom).

In Paragon programs, the policy operators $\sqcap$ and $\sqcup$ can be written $\star$ and + respectively.

Labelling Data and Information Flow Basics How do we use policies in programs? Data containers of various kinds (objects, files, communication channels) are classified according to the intended information that they contain. For example, the listing below declares a class with two boolean fields which are each declared with a specific policy; `mySecret` is intended for storing high confidentiality data and is labelled **`?`**`high`, indicating that reading (querying) the variable will reveal high confidentiality data:

```
import static HighLow.*; // Importing definitions of high and low

public class MyClass {
    ?high boolean mySecret;
    ?low  boolean myPublic;
}
```

The Paragon compiler ensures that all code will respect the policies on these fields. Thus the following code fragment will be accepted by the compiler:

```
    mySecret = myPublic;
```

It takes data labelled with `low` and places it in a location with a more restrictive label, namely `high`. However the compiler will flag the following assignment as an information flow violation:

```
    myPublic = mySecret;
```

It is instructive to see how we might attempt to fool the compiler by writing equivalent code. Suppose we try to hide the value of `mySecret` in a method `getSecret()` which simply returns the value of `mySecret`. This will not help: in Paragon, methods must declare the policy of their result, so we would have to write

```
public static ?high boolean getSecret(){
    return mySecret;
}
```

And the violation is detected by the compiler when we try to assign the result of the method to the `low` location:

```
    myPublic = getSecret();
```

Now consider the following alternative:

```
    if (mySecret) { myPublic = true; } else { myPublic = false; }
```

This is also caught by the compiler, which sees that a publicly visible action (assigning to a low variable) is performed in a secret "context" – i.e. at a program point that is reached or not depending on secret information. As a final attempt, let's try to combine the previous two obfuscations to trick the compiler. Could we hide the assignments inside an innocent looking method?

```
void setPublicTrue(){
   myPublic = true;
}
...
if (mySecret) { setPublicTrue(); } else { setPublicFalse(); }
```

Here we attempt to hide the publicly visible side effect inside a function, hoping that the compiler does not notice that we call this function in a high context. However, Paragon requires (for this very reason) that methods declare their information flow side effects – their so-called *write effect*. We must annotate not only the parameters and return type of a method, but also the policy of the lowest level to which the method might write. The write effect of a method tells us: if we run the method, what is the lowest level at which we might we notice an effect? In this case, Paragon will only accept the method declaration if we declare that it writes to a low variable thus:

```
!low void setPublicTrue(){
   myPublic = true;
}
```

With this annotation on the method the compiler easily rules out the attempted information flow via the secret context.

To fully cover all the ways in which information might flow through the program, Paragon also requires annotations on other components of the language, such as the arguments of a method or the exceptions it may throw. We come back to these in Section 5.

There are, however, limits to what kinds of information flows the compiler checks. It is possible to leak information by exploiting variation in the timing or termination behaviour of code. Such covert channels are beyond the scope of what Paragon attempts to achieve.

Combining Policies As a second example of a simple information flow policy, consider the problem of ensuring that data originating from an untrusted source (for example a user input) does not influence data which ought to be trustworthy (e.g. a query string to be sent to a database). This is sometimes referred to as an *integrity policy*.

To model this intent we can introduce another two policies, named `trusted` and `untrusted`. Once again we will not reveal how they are defined just yet, suffice to say that information labelled `trusted` may freely flow to containers labelled `untrusted`, but not vice-versa. Note that these policies behave just like `low` and `high`, respectively. In fact, if we only wanted to model integrity requirements then we could do so by reusing the `HighLow` module. However, by defining them as different abstract policies they can be combined freely to form policies which deal with confidentiality and integrity issues simultaneously. For example, suppose I have two secrets, my diary and my password. Both are secret, but my diary might well quote text from an untrusted source, whereas untrusted data should never have an influence on the contents of my password:

```
?(high ⊓ untrusted) String myDiary;
?(high ⊓ trusted)   String myPassword;

public !(high ⊓ untrusted) void savePassword(){
   myDiary += myPassword;
}
```

Method `savePassword()` writes a copy of the password into the diary. Note that if we accidentally assigned the diary contents to the password the compiler would reject the program, because the diary contents is untrusted, and as such cannot flow to a trusted container.

The ability to modularly combine policies in this way depends on the policies being orthogonal – the property that they do not interfere with each other. To see what this really means we need to see exactly how they are built.

3. Defining Static Policies

So far we have treated policies as abstract entities which are related according to some permitted information flows, and which can be combined using the operations $\sqcap$, and $\sqcup$. In this section we will open the box and see how such static policies can be defined.

The basic building block for static policies (by static here we mean policies that do not vary over time) is the notion of an *actor*. An actor is nothing more than an object which is used to model an observer of information. Static policies are built by simply specifying to which actors information may flow. (Dynamic policies, which will be described later, will further add conditions which describe the *circumstances* under which the information may flow.) Let us begin with the *HighLow* lattice, and consider one of several possible ways of defining it. The approach is to construct two actors, `highObserver` and `lowObserver`, as representations of users with high and low

security clearance, respectively. The `high` policy is defined by specifying that only `highObserver` may view high-labelled data. The `low` policy, on the other hand, is viewable by both `highObserver` and `lowObserver`. Hence the high policy is more restrictive than the low policy. The syntax necessary to achieve this is as follows:

```
public class HighLow {
    private static final Object lowObserver;
    private static final Object highObserver;

    public static final policy high = { highObserver : };
    public static final policy low  = { lowObserver :
                                      ; highObserver : };
}
```

Analogously, we could create a class `TrustedUntrusted` in which we use the actors `untrustedObserver` and `trustedObserver` respectively.

In the previous section we mentioned the existence of a "bottom" policy, more liberal than any other, and a "top" policy, being the most restrictive of all policies. These are not predefined policy constants, but are definable directly. The most restrictive policy is simply the empty set of actors: `{ : }`. Any policy which permits information to flow to some actor is by definition more liberal than this policy. At the other extreme, the most liberal policy says that data may flow to any actor

```
  public static final policy top    = { : };
  public static final policy bottom = { Object a : };
```

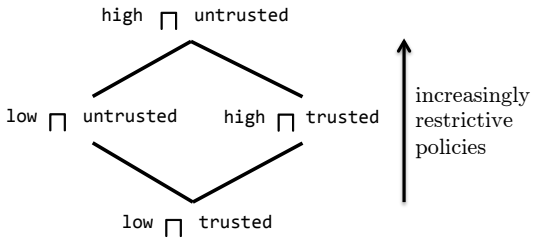
Note that this policy introduces us to a feature that we will see more often – a bound actor variable (in this case `a`). The policy says: for any actor `a`, information may flow to that actor.

Using bottom and top we can illustrate two alternative encodings of the HighLow policies:

```
public class HighLowB {

    private static final Object highObserver;

    public static final policy high = { highObserver : };
    public static final policy low  = bottom;
}

public class HighLowT {
    private static final Object lowObserver;

    public static final policy high = top;
    public static final policy low  = { lowObserver : };
}
```

In both cases the result creates two policies which are ordered in the intended way. Thus for the purposes of tracking information flow according to a high-low classification they

serve equally well. However there are some differences. The first version, `HighLow`, exports policies which are completely fresh – they cannot be reconstructed outside of the class. This means that they can be freely combined with other policies without "interference" between the policies. This means in particular that if we combine `HighLow` policies with `TrustedUntrusted` policies, however the latter are encoded, we will get the collection of policies we expect, as depicted in the Hasse diagram below:



This would not be the case if we used either of the other two encodings. `HighLowB.low` ⊓ `q` is equivalent to `HighLowB.low` for any policy q. On the other hand `HighLowT.high` ⊓ `q` is equivalent to `q` . For the other combinations the result will depend crucially on the encoding of the policy `q`.

4. Dynamic Lattices and Controlling Locks

So far we have seen how to define static policies. A static policy describes information flows that do not depend on the state of the system, or the context in which the flow occurs. Although useful, in practice the completely static prescription of information flow is too inflexible. For example:

- Declassification: confidential information becomes public whenever a login attempt is made: "incorrect password" tells us something about the secret, but sufficiently little that we would like to ignore it.
- Information Purchase: The permitted flows of a digital product depend on who has paid for them.
- Sanitisation: Untrusted data can flow to trusted sinks providing that it has been sanitised.
- Trust relations: Whether you can see my photos depends on whether you are my friend in a social network.

The key to Paragon's flexibility is the ability to make each potential information flow described in a policy *conditional on the state of the system*.

4.1. Locks

To make information flows conditional on the state of the system, we need a policy-level representation of the state. This interface is provided by a new type of object called a *lock*. A lock is special kind of boolean value which is used by the programmer to model security relevant events. Suppose that we have a system involving a potential buyer of

information, modelled by an object `customer`. Information which the customer may freely learn can be labelled with the policy `{customer:}`. To model information which must be purchased, for example a software activation key, we declare a lock `Paid`. The lock will be used to flag when payment has been made:

```
public class KeySeller {
    public static lock Paid;

    ?{customer: }     String customerData
    ?{customer: Paid} String softwareKey
}
```

Here the policy `{customer: Paid}` dictates that the software key can flow to the customer only when the `Paid` lock is *open* (the Paragon terminology for the paid lock being true).

A lock is a built-in data type of Paragon that can be used in limited ways. One of the important ways that we use a lock is to *open* and *close* the lock to signal changes in the state of the system. For example, suppose that we have a `confirmPayment` method which is called once payment is complete. This would be a suitable place to insert the code

```
    open(Paid)
```

Now consider the assignment in the following code:

```
    if (paymentSuccess) {
        confirmPayment()
        customerData = softwareKey
    }
```

Is this a legal information flow? The policy on the expression `softwareKey` is more restrictive than the policy on the target `customerData`, so it seems that it violates the intended information flow. However, the *effective policy* of the `softwareKey` at this program point is the policy obtained by deleting all locks which are known to be open at that point. Here the lock `Paid` is open, and thus the effective policy of `softwareKey` here is `{customer: }` and so the policies are indeed compatible. This all relies on the Paragon compiler to compute (an under-approximation of) the locks that are open at any assignment point in the program.

Runtime Testing of Locks However smart the compiler might be at working out which locks might be open, sometimes it is simply impossible to know the status of a lock at compile time. For example, suppose that `Paid` lock is opened after a successful payment:

```
public void processPayment() {
    // customer pays for item
    if (paymentSuccessful) { open Paid; } else { ... }
}
```

After calling `processPayment()` we cannot be sure, statically, that the `Paid` lock is open. To resolve this uncertainty we can add a run-time test on a lock, treating it as a boolean value:

```
    processPayment();
    if (Paid) { customerData = softwareKey; } else { ... }
```

Now, of course, the compiler has no problem determining that the assignment conforms to the information flow policy.

4.2. Parameterised Locks

Suppose now that instead of a single customer we have any number of individual customers, created from a class `Customer`. In this case we would need a Paid lock for each of these. This is accomplished by parameterising the locks over objects thus:

```
public class KeySeller {
    public static lock Paid(Customer);

    ?{Customer x : Paid x} String softwareKey
}
```

Thus the `Paid` lock is now a family of locks, and the policy says that the `softwareKey` can flow to any customer `x` who has paid.

In general, parameterised locks with one parameter (unary locks) are a good tool to model information flow to principals that depends on a dynamic *role*. Binary (two parameter) locks, on the other hand, are useful for modelling relationships between objects which may influence permitted flows. The following policy for work documents in an organisation reflects this, and also serves to illustrate that policies may have multiple clauses:

```
policy workData =
    { Manager m :
    ; (Manager m) Employee e : GivesPermissions(m, e)
    ; (Manager m) Employee e : IsBoss(m), WorksFor(e, m) };
```

This policy allows (clause 1) information to flow to any manager, (clause 2) any employee who has been given permission by a manager, and (clause 3) any employee who works immediately under the manager who is the overall boss. Spelling out the third clause in natural language, it says:

> For all managers *m*, information may flow to any employee *e* providing that *m* is a boss, and *e* works for *m*.

Comparing Policies As before, data with policy p can be assigned to a container with policy q providing that q does not permit any flows that are not already allowed by p. I.e. q is more restrictive than p. So for example suppose that `alice` is a `Manager`, and `bob` is an `Employee`. Then the following three policies are all more restrictive than `workData`:

```
policy workData2 =
    { Manager m :
    ; (Manager m) Employee e : IsBoss(m), WorksFor(e, m) };
```

```
policy aliceSecretProject = // only alice and the boss
    { alice :
    ; (Manager m) : IsBoss(m) };

policy bobAlice = // alice and maybe bob
    { alice :
    ; bob :  GivesPermissions(alice, bob) };
```

We could freely assign a value with policy `workData` to a container declared with any of these policies. On the other hand, the following policies are incompatible with `workData`, as they each allow some flows that `workData` does not necessarily allow.

```
policy managerAndBob =
   { Manager m :
   ; bob : };

policy underManagement =
   { Manager m :
   ; (Manager m) Employee e : WorksFor(e, m)  };
```

As before, when comparing policies the compiler will also take into account the locks which are open at the point of comparison, the *effective policy*. One way of understanding the effective policy when we have parameterised locks is to think of the locks which are open as allowing us to add further clauses to the policy. Consider, for example, `workData` in a context where `GivesPermissions(alice, bob)` is open. This means that, by the second clause of the policy, information may flow to `bob`. Thus the effective policy is

```
workData ⊓ { bob : }
```

Thus in this context we could assign `workData` to a container with policy `managerAndBob`.

4.3. Lock Properties

As we have seen, the status of locks influences the meaning of a policy. This means that we have to open and close appropriate locks at the right times. For some security policies this might result in a lot of seemingly boilerplate code. For instance, consider a company which has a strict hierarchy among its employees. Each employee has the authority to read all the documents that can be read by the employees below him in the hierarchy. The policy on a document of employee Elton might therefore be:

```
{ elton : ; Employee e : ReadsFor(e, elton) }
```

This means that the `ReadsFor` lock needs to properly represent the hierarchical state, including its transitive nature. Thus a method that allows an employee a to start reading documents of b already becomes quite complicated:

```
void allowReading(Employee a, Employee b) {
  open ReadsFor(a, b);
  for (Employee e : employees) {
    if (ReadsFor(b, e))  open ReadsFor(a, e);
    if (ReadsFor(e, a))  open ReadsFor(e, b);
  }
}
```

If a read permission gets revoked, *a* can no longer read for *b*, it becomes hard to even correctly modify the lock state:

```
void disallowReading(Employee a, Employee b) {
  close ReadsFor(a, b);
  for (Employee e : employees) {
    if (ReadsFor(b, e))  close ReadsFor(a, e);
  }
}
```

Is it correct to also disallow employee *a* from reading of any of the indirectly obtained permissions? Perhaps one could argue that this depends on whether those read permissions were not also provided explicitly to *a*.

Clearly, this all results in fairly complicated code, only to successfully maintain the correct lock state. At the same time it also leads to quite a large lock state that consumes memory at run time. To better address these situations where certain locks are *implicitly* opened depending on some explicitly opened locks, Paragon provides *lock properties*.

A lock property specifies under which circumstances a lock is implicitly opened. A property is defined at the declaration point of the lock on which it is a property, e.g.:

```
lock ReadsFor(Employee, Employee)
    { (Employee a) ReadsFor(a,a) :
    ; (Employee a, b, c) ReadsFor(a,c) : ReadsFor(a,b),
    ReadsFor(b,c) };
```

The clauses forming the lock properties are similar to the policy clauses we saw earlier. The first property states that each employee is allowed to read on his own behalf. The second property, in natural language, reads:

> Any employee *a* may read for any employee *c*, provided that there exists some employee *b* such that *a* reads for *b* and *b* reads for *c*.

With these properties, we now only need to maintain the essential links that form the hierarchy among the employees, while Paragon ensures that the implicit lock state is correctly maintained – that is, we can do without the for-loops in the previous code fragments.

Some forms of lock properties are rather common, in particular on binary locks. Paragon provides shorthands for three typical relational properties: reflexivity, transitivity, and symmetry. The two first are exactly the ones we have used above: that an actor is related to itself, and that we can form chains of relations, respectively. Symmetry specifies that if some actor *a* is related to some actor *b*, then *b* is also related to *a*. An example is the following relation:

```
symmetric reflexive transitive lock CoWorkers(Employee,Employee);
```

Finally, a lock property could involve other locks than the lock it is defined on. For example, each senior employee who is a manager is automatically also a member of the company's board:

```
lock BoardMember(Employee)
    { (Employee e) BoardMember(e) : Manager(e), Senior(e) };
```

5. Modularity

So far we have only looked at information flows in short code fragments. To enable information-flow control on realistic applications we need to track information flows across multiple methods, fields, and classes. Paragon does this in a way that enables each method to be checked independently of the others – much in the same manner that Java checks the types of methods.

In order to check information flows in a compositional way, each of these components (methods, fields, and classes) specifies additional annotations in its signature regarding its information flow policy – which the compiler needs to check. When a component is referenced elsewhere in the application we can reason about information flow using only its signature without needing to consult its actual implementation. Some of these annotations we already encountered in Section 2. In this section we present a full overview of all such annotations introduced in the Paragon language.

5.1. Read Effects

A *read effect* specifies the information flow policy of a particular location, such as the fields used to demonstrate various information flows in Section 2. If a read effect annotation on a field is omitted, the compiler defaults to the bottom (least restrictive) policy `{Object x:}`, except for locks themselves which are defaulted to the top policy `{:}`. The different default for locks is motivated by write effects, which are discussed in Section 5.2.

Similarly local variables within a method can be given a read effect:

```
public class Clazz {
    ?high boolean mySecret;
    ?low  boolean myPublic;
    void myMethod() {
        ?low boolean localBoolean = myPublic;
        mySecret = localBoolean;
    }
}
```

However, it is never necessary to explicitly specify the read effect on a local variable since Paragon will infer these effects for you, if possible, based on the other annotations found on fields, method's return values etc. If Paragon is not able to infer the read effects, this means that some information flow policy is violated via the use of local variables, such as in this fragment:

```
public class Clazz {
    ?high boolean mySecret;
    ?low  boolean myPublic;
    void myMethod() {
        boolean localBoolean = mySecret;
        myPublic = localBoolean;
    }
}
```

The arguments of a method and, as we saw previously, its return type, are annotated with a read effect:

```
?high int specialAdd(?low int lowInt, ?high int highInt) {
    return lowInt + highInt;
}
```

When checking the information flows within this method's body, we treat the argument `lowInt` as a location which stores information with policy `low`, and information in `highInt` has policy `high`. The read effect of any returned value should be at most as restrictive as the read effect on the method's return type.

In order to rely on these assumptions, the compiler checks that in every call to this method the arguments are not more restrictive then the annotated policies. Similarly, the result of the method should be stored in a location which has a policy at least as restrictive as the read effect on the method's return argument, i.e. `high`.

```
highInt = specialAdd(lowInt,  lowInt);   // Valid
highInt = specialAdd(lowInt,  highInt);  // Valid
highInt = specialAdd(highInt, lowInt);   // Invalid first
    argument
lowInt  = specialAdd(lowInt,  lowInt);   // Invalid assignment
```

The read effect annotations prevent unintended relabeling of information via method calls. When a read effect annotation is not present on a method argument, we assume that the method is polymorphic in that argument. The read effect annotations of the method can refer to its polymorphic policy using the **`policyof`** keyword:

```
?(policyof(otherInt)*low) int specialAdd(?low int lowInt, int
    otherInt) {
    return lowInt + otherInt;
}
```

With this definition the following uses of the method are both valid:

```
highInt = specialAdd(lowInt, highInt);
lowInt  = specialAdd(lowInt, lowInt);
```

When the read effect on the return type of the method is omitted, it defaults to the join of the read effects of all the arguments (polymorphic or not). Note that this default

is not necessarily restrictive enough, for example if the method uses more restrictive information to contribute to its result than is provided in its arguments:

```
public class Clazz {
    private ?high int highInt = 10;

    boolean greaterThan(int val) {
        return highInt > val;
    }
}
```

The compiler would yield the error when checking the return statement in the `greaterThan` method: the policy on the returned expression (**`policyof`**`(val)*high`) is more restrictive than what the method's signature expresses (**`policyof`**`(val)`).

5.2. Write Effects

The read effects used in the signatures of fields and methods prevent explicit information flows in a modular fashion. Write effects achieve the same for implicit information flows, i.e. flows via secret contexts.

All observable effects of statements that are visible outside a method's scope, such as performing assignments to fields or writing to an output stream, are referred to as *side effects*. Any entity that can observe the side effects of a method can potentially determine whether that method has been called. Therefore, the decision to call a method should not depend on information with a higher security level than the security level of the method's side effects – a flow that is exploited in the following class:

```
public class Clazz {
    public ?low int data = 0;

    void setData(?low int val) {
        this.data = val;
    }

    void myMethod(?high boolean secret) {
        if (secret) {    // Decision made on high data
            setData(1);  // Low side effect
        }
    }
}
```

If we inline the method call to `setData`, that is, replace it with the method's body **`this`**`.data = 1;`, we see that this is indeed the same implicit flow as we saw earlier in Section 2. To detect these implicit flows while preserving our modular approach, i.e. without inlining the method's body, Paragon requires that methods are annotated with their *write effect*.

The write effect of a method is a lower bound on its side effects. For every method, the compiler verifies that the write effect of any side-effect in its body, including calls to other methods, is indeed at least as restrictive as this lower bound. In our example, the

method `setData` has a statement with a side effect on security level `low` which has to be reflected in its signature. The write effect is written as `!pol` :

```
public class Clazz {
    public ?low int data = 0;

    !low void setData(?low int val) {
        this.data = val;
    }

    !low void myMethod(?high boolean secret) {
        setData(0);
        if (secret) {
            setData(1); // Invalid side effect detected
        }
    }
}
```

Since `myMethod` calls `setData` it inherits its side-effects and needs to be annotated accordingly. Due to the annotation on `setData` the Paragon compiler is now able to derive that the call in the branch depending on `secret` breaks the information flow policy via an implicit flow.

In the absence of a write effect the compiler assumes the top policy `{:}` as an upper bound on the side effects. This means that the method has either no side effects, or only side effects on containers with exactly the policy `{:}`.

One interesting observation is that opening and closing locks are side effects as well. As indicated in Section 5.1 the read effect of locks defaults to the top policy `{:}`, which means that the side effect of opening and closing a lock can be largely ignored by the programmer. The only reason to change the default policy on a lock is when the state of the lock may flow to different locations, i.e. via lock querying (see Section 4.1).

5.3. Lock-State Modifications

The locks which are open at any time will be referred to as the *lock state*. Irrespective of side effects, the lock state influences the effective information flow policy at any point in the code. To be able to determine how the effective policy changes between method calls, a method needs to have annotations that describe how it modifies the lock state.

Within one method it is relatively straightforward to determine how the lock state changes and therefore how the effective policy is affected. When another method is called, everything we know about the current lock state might be changed. The simplest approximation would therefore be to assume that all the locks we knew to be open before the method call are closed after the method call. This is a safe approximation because by assuming locks to be closed locks we strictly reduce the number of information flows which are deemed acceptable.

But such a pessimistic assumption would cause the compiler to reject many reasonable programs, so instead Paragon *requires* the programmer to annotate each method with *all* the locks it *potentially* closes. We write these lock state modification annotations as `-Lock`:

```
-LoggedIn -Administrator public void logout() {
    close LoggedIn;
    if (isAdministrator(user))
        close Administrator;
}
```

In this example the annotation says at *at most* locks `LoggedIn` and `Administrator` will be closed on return from this method. And neither of these annotations could safely be omitted from the method declaration. By implication, any other lock which was open before the method call will still be open after the call.

Conversely, we *allow* the programmer to annotate each method with *any* of the locks it *definitely* opens. These annotations are written as `+Lock`:

```
+LoggedIn public void login() {
    open LoggedIn;
    if (isAdministrator(this.user))
        open Administrator;
    open MayEdit;
}
```

Here, the programmer chose to not add the lock state modifier `+MayEdit` to the method's signature, although it would have been valid to do so. Adding `+Administrator` to the signature would not have been allowed, since this lock is not guaranteed to have been opened after the method returns. These annotations allow the compiler to update its approximation of the current lock state as the result of a method call, by only inspecting the method's signature.

As a final annotation, a method might explicitly state that it can only be called in a context where a particular lock is guaranteed to be open. This annotation is written `~Lock`:

```
~Administrator public void deletePage() {
    ...
}
```

Here we say that the method *expects* lock `Administrator` to be open. Thus when checking the body of this method the compiler assumes the lock `Administrator` to be open. In order to rely on this assumption, the compiler must also be able to determine that `Administrator` is open at every call-site of `deletePage` method, for example:

```
public void foo() {
    login();
    if (Administrator) // runtime check required
        deletePage();
    logout();
}
```

5.4. *Exceptions*

An alternative way of returning from a method to its caller is by throwing an exception. Since the effects of a method might differ depending on whether it returned normally or not, we annotate each thrown exception with its own write effect and lock state modifiers. In addition the exception itself might carry information, for example in its message, and can therefore be given a read effect as well.

```
!high +A -B public void divideBy(?low int lowInt)
        throws +A !low ?low Exception {
    open A;
    if (lowInt == 0)
        throw new Exception("Argument cannot be 0");
    this.highInt = this.highInt / lowInt;
    close B;
}
```

In the case that method `divideBy` returns normally, it guarantees to open lock `A`, close lock `B` and has a write effect `high` because it changes the value in the field `highInt`. In the case that the method returns with an exception, it has also guarantees to have opened lock `A` but has not closed lock `B` that annotation is left out.

The write effect annotation on the exception denotes what information might be learned from the fact that the exception is thrown. Therefore, code following a call to this method, up to and including the catch clause for this exception, may only perform side-effects whose write effects are of the specified policy or above. At the same time the annotation constrains the contexts inside the method's body where the exception could occur, enforcing that those contexts are no more restrictive than `low`.

Every exception forms an implicit information flow channel and therefore needs to be handled explicitly in every Paragon program. That is, every exception needs to be either caught or part of the method's **`throws`** annotation. This includes runtime exceptions such as `NullpointerException` and `ArithmeticException`. The method `divideBy` could thus also be written as:

```
!high +A -B public void divideBy(?low int lowInt)
        throws +A !{:} ?low ArithmeticException {
    open A;
    this.highInt = this.highInt / lowInt;
    close B;
}
```

5.5. *Classes and `.pi`-files*

Like Java applications, a typical Paragon application is a collection of various classes defined in separate `.para` files, collected in one or more packages. If the current class file under compilation refers to other classes, e.g. by extending them or using object instances of those other classes, the Paragon compiler does not need to have access to original source code of those classes. Instead, after the successful compilation of each class, a `.pi` file (or: Paragon Interface file) is created that preserves the policy-relevant

signatures of fields, methods and constructors. The Paragon compiler looks for these `.pi` files and assumes that the information flow policies therein are correct. This modular approach also allows the programmer to tie Paragon with Java programs, as is described in more detail in Section 7.

The current edition of the Paragon compiler does not support inner classes or cyclic dependencies between classes – for any referenced class the `.pi` file is assumed to exist. The Paragon compiler comes with a collection of Paragon Interface files for a subset of the standard java packages `java.io`, `java.lang`, `java.security`, `java.util`, `javax.crypto`, `javax.mail` and `javax.servlet` .

5.6. Generics and Policy-Agnostic Libraries

Certain classes might be agnostic to the actual information flow policy and simply provide functionality agnostic to the security level of the data that they handle. Most of the standard Java classes fall into this category, such as `List` or `InputStream`. To write these classes we build on the notion of generics, as introduced in Java 5. That is, it is possible to add **`policy`** type arguments to the definition of a class and use them as regular policy variables. Typically this occurs for classes that already have standard generic arguments, such as this simplified `LinkedList`:

```
public class LinkedList<G, policy p> implements List<G, p> {

    private ?p G data;
    private LinkedList<G, p> next;

    public LinkedList(?p G data) {
        this.data = data;
    }

    public ?(p*policyof(index)) G get(int index) {
        if (index == 0)
            return this.data;
        return this.next.get(index - 1);
    }

    public void append(?p G data) {
        if (this.next == null)
            this.next = new LinkedList<G,p>(data);
        else
            this.next.append(data);
    }

}
```

The (Java) type argument `G` is the type of the objects stored in the list, and each element in the list is given the policy `p`. The policy can be used exactly like any concrete policy, as is done for example in the read effect of the `get` method. We can now use this data structure to store elements of any desirable policy:

```
public void foo(?low Object a, ?high Object b) {

    LinkedList<Object, low> myList = new LinkedList<>();
    myList.append(a);
    myList.append(b); // Policy violation detected
    ?low Object x = myList.get(0);

}
```

Like in Java, it is also possible to provide type arguments to a method, for example in this static method to obtain an instance with a particular policy:

```
public class MyObject<policy p> {

   // Private constructor
   private MyObject() { ... }

   public static <policy q> MyObject<q> getInstance() {
       MyObject<q> result = new MyObject<q>();
       ...
       return result;
   }
}
```

Using policy type arguments it is possible to write libraries that are independent of the actual policy enforced. This places the information flow policy entirely on the level of the user code. On the other end of the spectrum it is possible to encapsulate an information flow policy entirely within the library code, forcing the user code to comply with the policy of the library. This is the perspective we explore in Section 6.

6. Abstraction and Encapsulation

Throughout Section 2 we used a library as a layer of abstraction between the security levels `high`, `low` and their actual definitions. In this section we profit even more from the synergy between Paragon policies and Java encapsulation, by encapsulating locks and even complete information flow paradigms as libraries.

6.1. Encapsulating Locks – a Sanitisation Policy

Locks enable conditional information flow policies. Opening locks make the effective information flow policy more liberal. But in general we don't always want the programmer to have full control over the opening and closing of locks. For example, we might want to open a lock briefly to allow the untrusted output of a sanitisation function to be moved to a trusted container. But we only want this to happen when we use the sanitisation operation. The key here is the appropriate use of encapsulation.

The following class exports a sanitisation function, stripping possible HTML tags from a string – like the `escapeHTML` function used in Section 1. It also exports a policy

`unescaped` which guards information flows using the *private* lock `Escaping`. As a result, information protected under the `unescaped` policy can only remove this policy by applying the `escapeHTML` method on that information. The method uses a policy argument `p` that allows the library's policy to be used in conjunction with any other policy.

```
public class Escaper {

    private lock Escaping;
    public static final policy unescaped = {Object x : Escaping};

    public static <policy p> ?p String
            escapeHTML(?(p*unescaped) String text) {
        open Escaping {
            // Perform the escaping
            return result;
        }
    }
}
```

External code can use the `unescaped` policy to label its untrusted sources of information. Using proper annotation on the arguments of the `postComment` and `storeComment` methods Paragon detects the information flow bug from Section 1 (assuming that the `getParameter` method returns a `String` with the same policy as its `HttpRequest` object):

```
public void storeComment(?low String name, ?low String comment) { ... }

public void postComment(?(low*Escaper.unescaped) HttpRequest userInput) {
    String name    = userInput.getParameter("name");
    String comment = userInput.getParameter("comment");

    ?low String esc_comment = Escaper.<low>escapeHTML(comment);

    storeComment(name, esc_comment); // Invalid flow detected
}
```

The escape policy easily combines with the external code's policy on the user input (`low`). The same pattern can be used for other so-called 'trusted declassifier' libraries, such as encryption, signing and tainting.

6.2. Encapsulating Information Flow Policies – a Framework for Third-party Code

In Paragon it is possible to completely encapsulate an application's information flow policy and force third-party code to follow that policy. As an example we use a very simplified framework for smart-phone applications. Third-party programmers can write applications in Paragon, addressing the resources of the phone via the framework's API. Our small case study's framework provides API operations to get the phone's GPS location, read and write access to the file system, and read and post access to the internet. The framework's information flow policy annotations dictate how these resources may

be used. Information on the file system is considered to be of the security level `high` whereas information on the internet is `low`. The GPS location can be declassified to `low` but only by asking the user's permission to do so.

```
public class Framework {

    private lock DeclassifyGPS;

    public final policy low  = { Object any : };
    public final policy high = { : };
    public final policy gps  = { Object any : DeclassifyGPS};

    ?low  String   readWebPage    (?low String page);
    ?high String   readFile       (?high String fileName);
    ?gps  Location getGPSLocation ();

    // Method that asks user to declassify location
    ?low Location declLocation(?gps Location loc)
            throws PermissionException;

    void writeFile(?high String fileName, ?high String data);
    void sendWeb(?low String url, ?low String data);
}

public interface App {
    public abstract void run(Framework framework);
}
```

An application for the phone is required to implement the `App` interface. The Paragon compiler enforces the policies specified by the framework, and detects any attempt by the third-party code to violate the information flow policy:

```
public class MyApp implements App {

  public void run(Framework fw) {
      String news = fw.readWebPage("thenews.com")
      fw.writeFile("latest.xml", news);
      String agenda = fw.readFile("agenda.xml");
      fw.sendWeb("collecting.com", agenda); // Invalid flow
      Location loc = fw.getGPSLocation();
      fw.sendWeb("collecting.com", loc.toString());
                                            // Invalid flow
      try {
          Location dloc = fw.declLocation(loc);
          fw.sendWeb("collecting.com", dloc.toString());
      } catch (PermissionException) {
          // User denied declassification
      }
  }

}
```

Note that, in this simple setting, the Paragon code in `MyApp` contains only standard Java syntax. If we are willing to accept that the third-party programmer needs to write some Paragon specific syntax we can encapsulate more complicated policies, such as different permissions for different applications, or policies that can be modified by the user of the phone.

7. Real world practicalities

So far we only discussed Paragon in the context of small or simplified examples. When starting to program large scale, real world applications in Paragon some additional programming paradigms become useful.

7.1. Linking Java and Paragon using `.pi`-files

In the Paragon distribution, Paragon Interface (`.pi`) files are used to summarise the information flows present in a collection of standard java packages. Although there is the risk that these information flow assertions are incorrect and introduce security bugs, they remove the need for rewriting the large (and partly native) Java code base entirely in Paragon.

The same approach can be taken to reuse existing Java code and libraries for a Paragon application, whenever the programmer is confident that this code is free of policy violations. For example, the JPMail application discussed in Section 7.4 uses `.pi`-files to avoid reimplementation of e-mail protocols and encryption standards in Paragon. This modularity allows the programmer to limit Paragon policy annotations to the parts of the application where information flows are deemed relevant and that need strong guarantees that the policy is not violated.

Since Paragon trusts that the information flow assertions in `.pi`-files are correct, it is possible to exploit this trust during the development of an application. In particular, it allows us to write a (`.pi`, `.java`) file pair for printing debugging messages that potentially violate the information flow policy.

```
// Debug.pi:
public native class Debug {
    public !{:} void out(?{:} String message);
}

// Debug.java:
public class Debug {
    private static final boolean DEBUG = true;
    public void out(String message) {
        if (DEBUG) {
            System.out.println("DEBUG: " + message);
        }
    }
}
```

The **`native`** keyword in the `.pi`-file indicates that this interface is written by hand and not the result of a Paragon compilation. The policies in the `.pi`-file assert that mes-

sages with any information flow policy can be provided as an argument, and that the method has no side effects. Therefore, the debug method can be called at any point in the application even though it violates the information flow policy. Naturally, the debugging module should not be functional during the application's deployment phase.

7.2. Input-Output Channels

Any real world application demands more input and output capabilities than just program arguments and return values. It is important to realise that all communication channels, such as writing to files or reading from an input stream, introduce additional implicit flows. An observer of the channel does not only learn *what* information is read or sent on the channel, but also *that* the channel is being used.

To correctly model this information flow each channel operation must come with an appropriate write effect. The write effect should be at least as permissive as the read effect of the data sent or received on the channel, since at least the observers of that data learn that the channel is used. There might also exist external observers who learn that the channel is used but not the content sent on it, for example when observing encrypted network traffic. Therefore the write effect might be of a strictly lower security level than the level of the information itself.

The standard Paragon Interface files supplied for streams assume that both the read and the write effect of channel operations have the same information flow policy:

```
public native class PrintStream<policy p> {
    !p public void print   (?p String s);
    !p public void println (?p String s);
    !p public void println ();
    !p public void flush   ();
    ...
}
public native class BufferedReader<policy p> {
    public BufferedReader(?p InputStreamReader<p> isr) { }

    !p public ?p String readLine() throws !p IOException;
    ...
}
```

7.3. Linking Paragon Actors with External Entities

Many real world applications with information flow concerns need a way of representing external entities in the application's policy. For example, a Unix application needs to specify information flow policies for the various users of the system, and a web server needs the same for the users of its web applications. A similar situation can be found on a smart phone that has information flow policies for the various applications installed on the phone. In these cases it appears natural to associate a separate actor with each of these external entities.

In Paragon actors are represented as objects, implying that each external entity should be represented by exactly one object pointer. Some care needs to be taken as to

how these actors are created. For example, the following method opens a lock for an actor that exists only in the scope of the method:

```
public void foo() {
    UnixUser alice = new UnixUser("alice");
    open SomeLock(alice);
}
```

The opening of the lock has no effect, since any other `new UnixUser("alice")` instance is referring to a different actor. In such a situation a variant of the singleton pattern can be used to guarantee that we always refer to the same particular instance of `UnixUser` for each actual user:

```
public class UnixUser {
    private static final HashMap<String, UnixUser> users =
            new HashMap<>();

    private UnixUser(String name) { ... }

    public static final UnixUser getUser(String username) {
        UnixUser u = users.get(username);
        if (u == null) {
            u = new UnixUser(username);
            users.put(username, u);
        }
        return u;
    }
}
```

7.4. JPMail

One of the larger current applications in Paragon implements a functional e-mail client based on JPMail [5]. In JPMail the user can specify a mail-policy file, partly dictating the information flow policies that the mail client has to enforce.

JPMail ensures that an e-mail is only sent if its body has been encrypted under an algorithm that is trusted by the receiver of the e-mail. Which encryption algorithms are trusted by what JPMail users is specified in the mail-policy file. In addition JPMail needs to enforce more static policies, e.g. preventing the login credentials from flowing anywhere else than to the e-mail server.

The JPMail example incorporates the various pragmatics discussed in this section, as well as most of the other features of the Paragon langauge. The source code of the application can be found on the Paragon website [1].

8. Further resources

More information about Paragon and its policy specification language can be found in the following resources:

Paragon project web site

All tutorials and publications related to Paragon are been collected on the Paragon project web page [1]. Here you can also find the interactive version of this tutorial that allows you to run the Paragon compiler via your web browser.

Paragon

For more information on the Paragon programming language, and for citing the project, please see [4].

Paralocks

For more information on the earlier versions of the policy specification language, see [3,6]. The version of the language used in Paragon can be found in the appendix of the technical report version of [4].

Issue reporting

If you find any issues in the Paragon compiler, we would be very grateful if you report them at our issue tracker, which can be found at [2].

References

[1] Paragon. Website, 2013. `http://www.cse.chalmers.se/research/group/paragon`.

[2] Paragon issue tracker. Website, 2013. `http://code.google.com/p/paragon-java/issues`.

[3] N. Broberg and D. Sands. Paralocks – Role-Based Information Flow Control and Beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.

[4] N. Broberg, B. van Delft, and D. Sands. Paragon for Practical Programming with Information-Flow Control. In *Asian Symposium on Programming Languages and Systems (APLAS) 2013*, volume 8301, pages 217–232. Springer, 2013.

[5] B. Hicks, K. Ahmadizadeh, and P. D. McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *ACSAC*. IEEE Computer Society, 2006.

[6] B. van Delft, N. Broberg, and D. Sands. A Datalog Semantics for Paralocks. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM)*, pages 305–320. Springer, 2012.

A. Overview Paragon Syntax

This appendix lists the additional syntax introduced by Paragon on top of Java.
TODO: check correctness of syntax here :-)

A.1. Locks

`lock` `MyLock;`
Defines 0-ary lock – may only appear as field of a class and has implicit modifiers **`final`** and **`static`**.

`lock` `MyLock { MyLock : OtherLock };`
Defines lock with a property stating that when `OtherLock` is open `MyLock` is also open.

`lock` `MyLock(File, Object);`
Defines lock of arity 2. Arguments to this lock should be of type `File` and `Object` respectively.

```
lock ActsFor(Actor, Actor)
    { ActsFor(alice, bob) :
    ; (Actor a) ActsFor(a, alice) : SomeLock(a)
    ; (Actor a, b, c) ActsFor(a, b) : ActsFor(a,c), ActsFor(c,b)
    };
```

Example of a lock with multiple lock properties. The lock is always opened for the actor pair `(alice,bob)`. Both `alice` and `bob` need to refer to final, non-null instances of type `Actor`. Any actor acts for `alice` provided that `SomeLock` is opened for that actor. Finally, the last clause makes the lock transitive. Note that `c` is existentially quantified, whereas the others are universally quantified.

`reflexive lock` `Rel(Object, Object);`
Adds the lock property `(Object a)Rel(a,a):`
Only applicable on binary locks.

`symmetric lock` `Rel(Object, Object);`
Adds the lock property `(Object a b)Rel(a,b): Rel(b,a)`
Only applicable on binary locks.

`transitive lock` `Rel(Object, Object);`
Adds the lock property `(Object a b c)Rel(a,b): Rel(a,c), Rel(b,c)`
Only applicable on binary locks.

`readonly lock` `MyLock;`
This lock can be queried outside its defining class, as well as used in method annotations outside its defining class, but can only be opened or closed within its defining class. The **`readonly`** modifier replaces the standard access modifier **`public`**, **`private`** or **`protected`**.

`open` `MyLock(alice);`
Statement which opens the lock `MyLock` for the final, non-null object `alice`.

`close` `MyLock(alice);`
Statement which closes the lock `MyLock` for the final, non-null object `alice`.

`if` `(MyLock(bob)){ s1 }` **`else`** `{ s2 }`
Queries the lock `MyLock` and executes branch `s1` if the lock is open, `s2` otherwise.

A.2. Lock Annotations

`+MyLock void m() { ... }`
: Specifies that method `m`, if terminated normally, guarantees to have opened `MyLock`.

`void m() throws +MyLock SQLException { ... }`
: Specifies that method `m`, if terminated with an `SQLException`, guarantees to have opened `MyLock`.

`-MyLock void m() { ... }`
: Specifies that method `m`, if terminated normally, might have closed `MyLock`.

`void m() throws -MyLock SQLException { ... }`
: Specifies that method `m`, if terminated with an `SQLException`, might have closed `MyLock`.

`~MyLock void m() { ... }`
: Enforces that method `m` can only be called in a context where `MyLock` is known to be open.

A.3. Policies

`policy bottom = { Object o : };`
: Defines the most permissive policy: information can be observed by anybody.

`policy top = { : };`
: Defines the most restrictive policy: information can be observed by nobody.

`policy pol = { alice : }`
: Specifies that information can only flow to `alice`, where `alice` is a final, non-null instance in scope.

`policy pol = { alice : HasPaid(alice)}`
: Information can flow to `alice` provided that she has paid, i.e. when the `HasPaid` lock is open for `alice`.

`policy pol = { Customer c : HasPaid(c), Release }`
: Information can flow to any instance of type `Customer` provided that customer has paid and the lock `Release` is open.

```
policy pol =
    { Manager m :
    ; (Manager m) Employee e : GivesPermissions(m, e)
    ; (Manager m) Employee e : IsBoss(m), WorksFor(m, e)
    };
```
: Policy with multiple clauses. Information can flow to any manager, to any employee if there is some manager that has given this employee permissions, and to any employee that works directly under the boss.

`policy pol = polA * polB;`
: Defines the policy to be the most permissive policy that is at least as restrictive as `polA` as well as `polB`. Also referred to as the *join* or $\sqcup$ of the two policies.

`policy pol = polA + polB;`
: Defines the policy to be the most restrictive policy that is at least as permissive as `polA` as well as `polB`. Also referred to as the *meet* or $\sqcap$ of the two policies.

```
public static typemethod policy owner(Actor a) {
    return { a : Owner(a) };
}
```
: A type method is evaluated at compile time and therefore has to be static, deterministic and side-effect free.

`policy pol = owner(alice);`
: A policy definition using a type method.

`?pol`

Declares a read effect. The policy has to be final. If the policy is absent it is defaulted to the following:

Read effect on	Default policy
local variable	policy inferred by compiler
field	`{ Object o : }`
lock	`{ : }`
method argument	parametric in argument
method return type	join of all arguments
thrown exception	join of all arguments

`!pol`

Declares a write effect. The policy has to be final. If the policy is absent it is defaulted to the following:

Write effect on	Default policy
method return type	`{ : }`
method exception	`{ : }`

```
class Clazz<policy p>
```

Class definition with a policy as type argument. The provided policy has to be final.

```
class Clazz<actor Employee>
```

Class definition that requires a type argument of type `Employee` to serve as an actor (i.e. the provided argument has to be final and not-null).

```
Clazz c = new Clazz<pol>()
```

Calling the constructor of the class, providing a final policy as type argument.

```
<policy p> void m()
```

Method signature that requires a policy as type argument. The provided policy has to be final.

```
inst.<pol>m();
```

Calling an instance method with a policy as type argument.

```
Clazz.<pol>m();
```

Calling a static method with a policy as type argument.

```
?r Clazz<p>[]<q> myArray = new Clazz<p>[10];
```

Creates an array storing instances of `Clazz<p>`, i.e. `Clazz` with policy argument `p`. The elements of the array have the policy `q`. The array itself has policy `r`.

```
?(policyof(arg)*q) int m(int arg) { ... }
```

Method is polymorphic in the policy of its argument; returned value has the same policy joined with policy `q`.