Towards Capability Policy Specification and Verification

Abstract. The object-capability model is a de-facto industry standard widely adopted for the implementation of security policies for web-based software. Unfortunately, code written using capabilities tends to concentrate on the low-level *mechanism* rather than the high-level *policy*, and the parts implementing the policy are tangled with parts implementing the functionality.
In this paper we argue that the policies intended by programs using object capabilities should be made explicit and written separately from the code implementing them. We also argue that the specification of such capability policies requires concepts that go beyond the features of current specification languages. Moreover, we argue that we need methodologies with which to prove that programs adhere to their capability policies as specified.
We sketch such a capability policy specification language, and propose capability triples, which are a generalisation of Hoare logic triples. Capability triples consist of conditions, code and conclusions, but interestingly they allow the conditions as well as the conclusion to talk about the state before and after execution of the code, they allow the code to be existentially or universally quantified, and their interpretation quantifies over all modules extending the curent module.
Based on the famous mint example [22], we outline its capability policy specifications, and demonstrate how we can reason that it satisfies the capability policies. Interestingly, the reasoning makes use of restrictions imposed by the type system, such as `final` and `private`.

1 Introduction

Capabilities — unforgeable authentication tokens — have been used to provide security and task separation on multi-user machines since the 60s [7], *e.g.* PDP-1, operating systems *e.g.* CAL-TSS [16], and the CAP computer and operating system [47]. In capability-based security resources can only be accessed via capabilities: possessing a capability gives the right to access the resource represented by that capability.

Object capabilities [24] apply the concept of capability to object-oriented programming languages. In an object capability system, an object is a capability for the services the object provides: any part of a program that has a reference to an object can always use all the services of that object. To restrict authority over an object, one can create, instead of "naked" reference, an intermediate object which offers only restricted services on the original object.

Object capabilities afford simpler and more fine-grained protection than privilege levels (as in Unix), static types, ad-hoc dynamic security managers (as in Java or JSand [1]), or state-machine-based event monitoring [2]. On the other hand, object capability systems are only secure as long as trusted capabilities (that is, trusted objects) are never leaked to untrusted code. Object capabilities have been adopted in several programming languages [26, 22, 45] and are increasingly used for the provision of security in web-programming in industry [27, 46, 39].

On a different development strand to object capabilities, and with the aim to restrict access across code, programming languages adopted features like packages and opaque types, `const` or fields, `private` and `protected` members, or `final` classes [48, 40]. More advanced features, such as ownership types [6], restrict access to different parts of the heap. Such features do not introduce new behaviour into the language, but restrict the set of legal programs, hence we call them here *restrictive*; they usually are part of statically typed languages.

The key problem with object capability programming as practiced today is that — because capabilities are just objects — code manipulating capabilities is tangled together with code supporting the functional behaviour of the program. The actual security policies enforced by a program are *implicit*, scattered throughout the program's code. Any part of a program that uses an object may (by oversight, error, or fraud) hand that object to an untrusted part of the program, giving the untrusted code access to all the services provided by that object. This makes it difficult to determine what security properties are guaranteed by a given program, and as a result, programs are difficult to understand, validate, and maintain.

We argue that capability policies should be specified separately from the program implementing them. And that the specification of capability policies requires features that go beyond what is available in current specification languages. Namely, capability policies are *program centred*, *fine grained*, *open* in the sense that they specify aspects of the behaviour of all possible extensions of a program, and have *deny* elements, which require that certain effects may only take place if the originating code or the runtime context satisfy some conditions. In [10] the authors anticipated expressing such policies through extensions of temporal logics.

In this paper we propose that capability policies can be specified through a generalisation of Hoare triples, which we call *capability triples*. Capability triples allow explicit quantification over modules, and over code – and thus reflect the open nature of capability policies. They also allow premises to talk about properties observed in the configuration *before* as well as *after* execution of the code – and thus they reflect the deny elements of the policies. Finally, the triples not only talk about properties of the heap (*e.g.,* the balance of an account), but also about accessibility and tracing properties (*e.g.,* the current stack frame has a path to object `o` which involves only `public` fields) – thus they reflect the program centred elements of capability policies.

To make the meaning of such policies precise, we define some concepts relating to program execution, accessibility, reachability and tracing. We give the definition of their manifestation for a Joe-E/Java subset, but we believe that these concepts have a manifestation in most object-oriented programming languages. We use the Mint example [22] to illustrate our ideas, and gave precise meaning to five out of the six policies proposed informally in that paper. We were surprised by the many different interpretations we found for the policies. We then give a semi-formal argument showing that certain code adheres to the capability policy. In doing so, we make heavy use of restrictive language features; this was surprising for us, since in traditional program verification, but also in verification of refinement properties, restrictive features have played no role.

The rest of the paper is organised as follows: Sect. 2 presents the Mint [22] as an example of object-capability programming, implemented in Joe-E/Java. Based on

that example, sect. 3 distills the characteristics of capability policies. Sect 5 informally explores reasoning about capability policies, and the use restrictive language features. Sect. 6 discusses further useful policies, not listed in [22]. Section 7 surveys related work. Sect. 8 concludes.

2 Object-Capability Example

We use as running example a system for electronic money as proposed in [26]. This example allows for mints with electronic money, purses held within mints, and transfers of funds between purses. The *currency of a mint* is the sum of the balances of all purses created by that mint. Purses trust the mint to which they belong, and programs using the money system trust their purses (and thus the mint). Crucially, separate users of the money system *do not* trust each other.

The standard presentation of the mint example defines six capability policies, which we repeat here, as they were described in [26]:

Pol_1 With two purses of the same mint, one can transfer money between them.
Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.
Pol_3 The mint can only inflate its own currency.
Pol_4 No one can affect the balance of a purse they don't have.
Pol_5 Balances are always non-negative integers.
Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

An immediate consequence of these policies is that the mint capability gives its holder the ability to subvert the currency system by "printing money". This means that while purse capabilities may safely be passed around the system, the mint capability must be carefully protected.

There is also an implicit assumption that no purses are destroyed. This assumption is necessary because destruction of a purse would decrease the currency of a mint, in opposition to **Pol_3**. The implication of this assumption is that there will be no explicit destruction of purses, but also no garbage collection of purses.

Several different implementations have been proposed for the mint. Fig.2 contains an implementation in Joe-E [22], a capability-oriented subset of Java, which restricts static variables and reflection.

In the Joe-E version, the policies are adhered to through the interplay of appropriate actions in the method bodies (*e.g.* the check in line 17), with the use of Java's *restrictive* language features (`private` members are visible to the same class only; `final` fields cannot be changed after initialisation; and `final` classes cannot be extended). The code concerned with the functional behaviour is tangled with the code implementing the policy (*e.g.* in `deposit`, line 19 is concerned with the functionality, while line 17 is concerned with **Pol_2**). The implementation of *one* policy is scattered throughout the code, and may use explicit runtime tests, as well as restrictive elements (*e.g.* **Pol_2** is implemented through a check in line 17, the `private` and `final` annotations, and the initialisations in lines 9 and 13). Note that an apparently innocuous change to this code

```
1  public final class Mint {      }   //  the Mint capability
2
3  public final class Purse {
4      private final Mint mint;
5      private long balance;
6
7      public Purse(Mint mint, long balance) {
8         if (balance<0){throw new IllegalArgtException();};
9         this.mint = mint;  this.balance = balance;
10     }
11
12     public Purse(Purse pts) {
13        mint = pts.mint;  balance = 0;
14     }
15
16     public void deposit(Purse prs, long amnt) {
17        if ( mint!=prs.mint || amnt>prs.balance || amnt+balance
              <0 )
18           { throw new IllegalArgtException(); };
19        prs.balance -= amnt;  balance += amnt;   }
20     }
21 }
```

— such as a public `getMint` accessor that returned a purse's `mint` — would be enough to leak the `mint` to untrusted code, destroying the security of the whole system.

Both the Joe-E and the E versions suffer from tangling of policy with functionality, and scattering of policy implementations [10].

3 Capability Policies

We use the term *capability policy* as a description of how capabilities are intended to be used: which objects are trusted, which are untrusted, and precisely which capabilities can be accessed by which object. A key feature of capability systems is the *principle of least authority* — an object should only be able to access the capabilities (i.e. the other objects) that it needs in order to function correctly: even a trusted object should not have access to all the capabilities (objects) in the system [37, 29, 47]. A range of object capability policies are discernible from the literature [24, 26, 25].

Capability policies generally have the following characteristics:

- They are *program centred*: they talk about properties of uses of programs rather than properties of execution of protocols.
- They are *fine-grained*: they can talk about *individual objects*, while *coarse-grained* policies only talk about large components such as `System` or `DOM`.

- They are *open*. *Open* requirements must be satisfied for any use of the code *extended in any possible manner*, while *closed* requirements need only be satisfied for any use of code itself.
- They have *rely* as well as *deny* elements. Rely elements essentially promise that execution of a *given* code tin a state satisfying a given pre-condition will reach another state which satisfies some post-condition [13]. Deny elements promise that if an execution reaches a certain state, or changes state in a certain way, or accesses some program entity, then the code must satisfy some given properties.
  In other words, rely policies are about *sufficient* conditions, while deny policies are about *necessary* conditions.

None of the terms above are standard; we coined them to delineate our ideas. The mint's policies are capability policies; namely::

- They are program centred, since they referto actual programs.
- They are fine grained, as they refer to the *individual* purses and mints;
- Even though not explicitly stated there, the policy in [26] is expected to be open; any expansion of the code (through dynamic loading, subclassing, mashups *e.t.c.*) should satisfy the requirements.
- They contain rely as well as deny elements:
  - **Pol_1** is a rely requirement, expressible through classic pre- and post- conditions: namely, execution of `deposit` in a state satisfying the pre-condition that the where the two purses belong to the same mint leads to a state satisfying the post-condition that where the money has been transferred.
  - **Pol_2** is a deny requirement; it says that a currency may be changed by some code only if the code contained a function call executed by the mint owning the currency. **Pol_3** is another deny requirement; it says that if the currency should change, then it increases. **Pol_4** is also a deny requirement, preventing objects that cannot access a purse from modifying its value.
  - **Pol_5** can be understood as an object invariant, requiring purses' balances to always be positive, but also as a deny requirement which requires that any code preserves this property.

Open policies are central for JavaScript security, which requires that in any mashup, untrusted code cannot access the trusted security-critical resources of the execution environment (*e.g.* the DOM), nor interfere with the execution of any other component [19, 15]. These works usually implement coarse-grained, fixed-in-advance policies. SecureJS [44] leverages local scoping (a restrictive feature) to prove fine-grained confinement (*e.g.* `decr` is confined), but cannot express the high-level policies (*e.g.* currency cannot be affected). JSand [1] uses Secure ECMAScript and proxies (other restrictive features) to isolate the DOM, and then ensures access to that DOM proxy are mediated by dynamically checking security policies expressed as JavaScript predicates.

Deny policies are related to *deny-guarantee* specifications [9] which can forbid given locations from being modified by the current, or the other threads. Deny policies typically apply throughout program execution, rather than during specific functions, and refer to any properties of the program (*e.g.* the currency), rather than specific locations.

Deny policies are also related to *correspondence assertions* [49, 12], which require principals reaching a certain point in a protocol to be preceded by other principals reaching corresponding points. Recently, correspondence assertions have been adapted to refer to program state, and thus can prove that the code adheres to security, authentication, and privacy policies [3]: functions are annotated by *refinement types* that require that the function is only called if its arguments satisfy the type's conditions.

Deny policies go further than correspondence assertions in the following significant ways:

- They support *implicit* properties, *i.e.* properties that depend on state reachable from more than one object, perhaps quantifying over the complete heap, or even on the history of execution. In our example, the currency is the sum of the balance of all purses from the same mint, and therefore is an implicit property.
- They are *pervasive*, *i.e.* they are not attached to one function, and may be affected by several different methods. For example, the currency may be affected by the creation of purses and the payments.
- They are *persistent*, *i.e.* they allow the comparison of properties of the state at different times in execution. For example, **Pol_3** compares the currency between any two times in execution.

Deny policies could be transformed into equivalent refinement types; however, the transformation would not be trivial, and the resulting policies would not be open (because the refinement types cannot prevent the addition of functions which break the requirements), and less abstract (how would refinement types express that the currency can only grow?).

4 Towards Capability Policy Specification Languages

In this section we sketch a capability policy specification language. Our specification language is a generalisation of Hoare triples. We allow quantification over any code, and the extension of modules by any further modules — thus we deal with the open nature of policies. Moreover, in the precondition we allow the comparison of the state before and after execution — thus we deal with the deny part of policies, and can specify necessary, instead of sufficient, conditions.

We start by introducing the concepts necessary to give precise meaning to policies (sect 4.1), then use these to describe the meaning of the policies (sect 4.2), and then outline a capability policy specification language.

4.1 Concepts for Capability Policy Specifications

Capability policies should be expressible regardless of the particular program implementing them and the particular programming language the program was written in, provided that they (the policies) are expressed in terms of concepts which have their own individual manifestation in the particular programming language. For example, the concept of runtime configuration is manifest in programming languages as diverse as Cobol, Java, or Python, even though its manifestation differs.

In this section we introduce concepts used to define the meaning of policies. We describe their manifestation in a "capability-safe" Java-subset, which we call $\mathcal{L}ng_{\mathtt{cj}}$. We give a precise definition of $\mathcal{L}ng_{\mathtt{cj}}$ and all concepts in appendix A, while here we bring out the most salient issues.

Modules and Linking In order to reflect the open nature of capability policies, we need a handle on programs, and on the extension of programs (through subclasses, mashups, imports etc). For this we use *modules*, M, to describe parts of programs, and $*$ to describe the combination of two programs into one larger program.

A *module* is essentially a collection of definitions. In ML, modules could be structures and functors, in Java they could be class definitions and packages. In $\mathcal{L}ng_{\mathtt{cj}}$, modules are class declarations (*c.f.* App. A.1). Thus, the class definitions $M_{\mathtt{Mint}}$ and $M_{\mathtt{Purse}}$ from Fig. 1 are modules.

Remember that adherence to policies often relies on the correct use of restrictive features. Therefore, in $\mathcal{L}ng_{\mathtt{cj}}$we support annotations like private and final, and the type rules forbid access to private fields or methods outside their classes, forbid extensions of final classes and redefinition of method declared as final in the superclass, and forbid assignment to final fields outside their contractor (*c.f.* App. A.2).

The operator $*$ is a linking operation. In general, linking performs some compatibility checks, and therefore is only partially defined. For example, because the field `balance` is private, $M_{\mathtt{Purse}} * M'$ would be undefined, if M' contained somewhere the expression $\mathtt{newPurse}(\_,\_).\mathtt{balance}$. In App. A.3 the operation $*$ is only defined if it gives rise to a well-formed module.

Modules are not directly executable, but are necessary for the execution of *code snippets*. In $\mathcal{L}ng_{\mathtt{cj}}$code snippets are expressions, *c.f.* App. A.1. We use variables $code$, $code'$ to range over code snippets.

Runtime Execution Context and Expression Evaluation Execution contexts, $ctxt$, stand for runtime contexts in which code snippets are executed. In the case of Java, a context is a stack frame (i.e. a mapping from variable names to object addresses or scalar values), and a heap (i.e. a mapping from object addresses to objects), *c.f.* App. A.4.

Execution of a code snippet $code$ for a module M takes a context $ctxt$ and returns a value $\mathtt{v}$ and a new context $ctxt'$. We describe this through a large step semantics, of the shape $M,\ ctxt,\ code \rightsquigarrow ctxt',\ \mathtt{v}'$. We define this relation for $\mathcal{L}ng_{\mathtt{cj}}$ in App. A.4.

Arising Runtime Configurations When considering adherence to policies, we it is essential to examine only those runtime configurations (*i.e.,* context and code pairs) which may arise through the execution of the given modules. For example, if we allowed *any* well-formed configuration (well-formed in the sense of the type system) to be examined, then we would be unable to ascertain that **Pol_5**, which guarantees that balances are always positive, is adhered to.

Thus, $\mathcal{A}rising(\mathtt{M})$ is the set of runtime configurations which may occur during execution of some initial configuration ($ctxt_0$,$expr_0$). These concepts are defined in App. A.6; the sets are always defined, even if $M,\ ctxt_0,\ expr_0$ does not terminate.

Objects Accessible from a Context; Objects used during execution Object capabilities draw on the requirement that some heap entities might not be accessible from all the stack frames. For example, a context containing and having access to a purse object `prs`, will also contain the mint object `mnt` which created that purse, but it need not have access to that `mnt` object, since the field `mint` is private in `Purse`.

We therefore distinguish between the sets $\mathcal{A}ccAll(\texttt{M}, ctxt)$ resp. $\mathcal{A}ccPub(\texttt{M}, ctxt)$ which return the set of objects which are accessible from the frame in $ctxt$ through *any* path resp. through paths which do not visit private fields, unless they belong to objects of the same class as `this`. The definitions appear in App. A.8 and A.8.

We use the notation $\texttt{z} :_{ctxt} \texttt{c}$ to indicate that `z` is the name of an object which exists in the heap of $ctxt$ and belongs to class `c` - no requirement that there should be a path from the frame to this object. The notation $\texttt{z}, \texttt{z}' :_{ctxt} \texttt{c}$ stands for $\texttt{z} :_{ctxt} \texttt{c} \wedge \texttt{z}' :_{ctxt} \texttt{c}$. Finally, we define $\mathcal{U}sed(\texttt{M}, ctxt, code)$, as the set of all addresses used during execution of $code$ in the context $ctxt$ - for full definitions *c.f.* App. A.7.

4.2 Giving precise meaning to the policies

Armed with the concepts from section 4.1, we turn our attention to the precise meaning of the six policies. An important aspect of our approach is that we quantify over modules, extensions to modules, over the code being executed, and over execution contexts.

We discuss the policies in order of increasing complexity of their specification. We were surprised how many different interpretations we uncovered while developing this part of the work.

The fifth policy **Pol_5**, "Balances are always non-negative integers", is akin to object invariants [23, 31, 42]. We can express the policy directly, by requiring that a module M satisfies **Pol_5**, if for all M' legal extensions of M, and runtime configurations $(ctxt, expr)$ arising through execution of the augmented program $M * M'$, the configuration $(ctxt, expr)$ will produce a context $ctxt'$, in which the balance is positive.

Module M satisfies policy **Pol_5**
iff
$\forall\, M'.\, \forall (code, ctxt) \in \mathcal{A}rising(M * M'),$ and $\texttt{prs} :_{ctxt} \texttt{Purse}$:
$M * M', ctxt, code \rightsquigarrow ctxt', \texttt{v} \Rightarrow \texttt{prs.balance}_{ctxt'} \geq 0$

The subscripts in the path expression `prs.balance` indicate whether the path is looked up in the old, or the new context ($\texttt{prs.balance}_{ctxt}$ vs $\texttt{prs.balance}_{ctxt'}$).

Execution uses the extended module $M * M'$, where M' is universally quantified. This reflects the open nature of capability policies. It is essential to allow $M * M'$ in the execution, because this supports calling methods and accessing fields defined in M but also in M'.

We implicitly assume that quantification over linked modules only quantifies over those where the $*$ operator is defined. Thus,

$\forall M'.\forall(ctxt, code) \in \mathcal{A}rising(M * M')$
implicitly stands for
$\forall M' s.t. M * M'$is defined, $\forall code, ctxt, s.t.(ctxt, code) \in \mathcal{A}rising(M * M')$.

The third policy **Pol_3**, stating "The mint can only inflate its own currency", could mean that the currency of a mint never decreases, or that the mint cannot affect the currency of a different mint. As the second meaning is a corollary of **Pol_2**, we consider the first meaning only.

We first define the *currency* of a mint, which is an implicit assertion in the sense from section 3, as it quantifies over all objects of the heap:

$$Currency_{ctxt}(\texttt{mnt}) = \sum_{\texttt{p} \in Ps(\texttt{mnt})_{ctx}} \texttt{p.balance}_{ctxt}$$
$$\text{where } Ps(\texttt{mnt})_{ctxt} = \{\texttt{p} \mid \texttt{p} :_{ctxt} \texttt{Purse} \wedge \texttt{p.mint}_{ctxt} = \texttt{mnt}_{ctxt}\}$$

We can now express the policy.

Module M satisfies policy **Pol_3**
iff
$\forall\, M'.\, \forall(ctxt, code) \in \mathcal{A}rising(M * M')$, and $\texttt{mnt}:_{ctxt}\texttt{Mint}$
$M * M', ctxt, code \rightsquigarrow ctxt', \texttt{v}$
$\Rightarrow$
$Currency(\texttt{mnt})_{ctxt} \leq Currency(\texttt{mnt})_{ctxt'}$

Pol_3 describes a monotonic property, and is therefore related to history invariants [17]. However, it differs from history invariants through its open nature, and hence the quantification over M'. Note that in the conclusion we talk about the values of functions in the old context (*i.e.* $Currency(\texttt{mnt})_{ctxt}$) as well as those in the new context (*i.e.* $Currency(\texttt{mnt})_{ctxt'}$).

The first policy. **Pol_1** states"With two purses of the same mint, one can transfer money between them". It can be understood to mean that if `p1` and `p2` are purses of the same mint, then the method call $\texttt{p1.deposit}(\texttt{p2}, \texttt{m})$ will transfer the money. Therefore, it can specified through a Hoare Logic triple as follows:

$$\{\ \texttt{p1.mint} = \texttt{p2.mint} \wedge \texttt{p1.amount} = \texttt{k1} \wedge \texttt{p2.amount} = \texttt{k2} + \texttt{m}\ \}$$
$$\texttt{p1.deposit}(\texttt{p2}, \texttt{m})$$
$$\{\ \texttt{p1.amount} = \texttt{k1} + \texttt{m} \wedge \texttt{p2.amount} = \texttt{k2}\ \}$$

Using a similar notation to that we used so far, we can write **Pol_1** as:

Module M satisfies policy **Pol_1**
iff

$$\forall\, M'.\, \forall\, (ctxt, \texttt{p1.deposit(p2,m)}) \in \mathcal{A}rising(M * M'),\ \text{with } \texttt{p1,p2}:_{ctxt} \texttt{Purse}:$$
$$\texttt{p1.mint}_{\,ctxt} = \texttt{p2.mint}_{ctxt} \ \wedge\ \texttt{p2.balance}_{ctxt} \geq \texttt{m}$$
$$\wedge\, M * M', ctxt, \texttt{p1.deposit(p2)} \rightsquigarrow ctxt', \texttt{v}$$
$$\Rightarrow$$
$$\texttt{p1.balance}_{ctxt'} = \texttt{p1.balance}_{ctxt} + \texttt{m} \wedge\ \texttt{p2.balance}_{ctxt'} = \texttt{p2.balance}_{ctxt} - \texttt{m}.$$

The above specification ranges over all module extensions, M', and thus covers a larger set of runtime configurations than if it was expressed without the quantification over M'. Therefore it guarantees that the code M' can do nothing to break the behaviour of the `deposit` method from M, thus forcing the method `deposit` to be final, or a system with contracts which ensures any subclasses will satisfy the same contract.

The above is perhaps an over-specification, as it prescribes *how* the transfer is to take place by explicitly calling the $\texttt{p1.deposit(p2)}$ method. Instead, we may want to only requiring that it should be *possible* for the transfer to take place, without discussing features of the program design. Therefore, we define a second, more general version of the policy, which only requires the existence of some code snippet that performs the transaction, and which applies existential qualification over the code:

Module M satisfies policy **Pol_1, vrs2**
iff
$$\forall (ctxt, \_) \in \mathcal{A}rising(M).\ \forall\, \texttt{p1}, \texttt{p2}:_{ctxt} \texttt{Purse}.$$
$$\texttt{p1.mint}_{\,ctxt} = \texttt{p2.mint}_{ctxt} \ \wedge\ \texttt{p2.balance} \geq \texttt{m} \ \wedge\ \texttt{p1}, \texttt{p2} \in \mathcal{A}ccPub(M, ctxt)$$
$$\Rightarrow$$
$$\exists\, code, \text{ such that } M, ctxt, code \rightsquigarrow ctxt', \texttt{v} \ \wedge$$
$$\texttt{p1.balance}_{ctxt'} = \texttt{p1.balance}_{ctxt} + \texttt{m} \wedge\ \texttt{p2.balance}_{ctxt'} = \texttt{p2.balance}_{ctxt} - \texttt{m}.$$

In the above specification, the existentially quantified code only appears in the conclusion of the specification. This gives the correct, existentially quantified meaning to the selection of $code$. We require that the two purses are accessible in $ctxt$ without reading private fields ($\mathcal{A}ccPub(M, ctxt)$). The specification does not range over extending modules, because the specification as given here implies the one quantifying over module extensions M' – proof is further work.

Another possible meaning of **Pol_1**, however, is that the function `deposit` may only be called if the two objects had the same mint:

Module M satisfies policy **Pol_1, vrs3**
iff
$$\forall M', ctxt, (ctxt, \texttt{p1.deposit(p2)}) \in \mathcal{A}rising(M * M').\ \ \forall\, \texttt{p1},\ \texttt{p2}:_{ctxt} \texttt{Purse}.$$
$$M * M', ctxt, \texttt{p1.deposit(p2,m)} \rightsquigarrow ctxt', \texttt{v}$$

$$\Rightarrow$$
$$\texttt{p1.mint}_{ctxt} = \texttt{p2.mint}_{ctxt}$$

Note that in the above specification the conclusion is only concerned with properties observable in the original context, $ctxt$, while he premise is concerned with properties observable in $ctxt$ as well as $ctxt'$. This reflects the deny nature of the policy.

A fourth possible meaning of **Pol_1** would be that a money transfer from `p1` to `p2` may take place only if `p1` and `p2` share the mint. This poses the challenge of identifying *the cause* of any difference in the balance; we therefore leave it for further work.

Finally, a fifth, and more straightforward meaning would mandate that the balance of a purse `p1` may change, only if `deposit` was executed on `p1` or with `p1` as an argument. This can be expressed as follows:

Module M satisfies policy **Pol_1, vrs5**
iff
$$\forall M'.\, \forall (ctxt, code) \in \mathcal{A}rising(M * M').\, \forall \texttt{p1} :_{ctxt} \texttt{Purse}.$$
$$M * M', ctxt, code \rightsquigarrow ctxt', \texttt{v} \;\wedge\; \texttt{p1.balance}_{ctxt} \neq \texttt{p1.balance}_{ctxt'}$$
$$\Rightarrow$$
$$\exists ctxt', \text{ s.t. } \quad (ctxt', \texttt{p1.deposit}(\_, \_)) \in \mathcal{R}each(M * M', ctxt, code) \text{ or}$$
$$(ctxt', \_.\texttt{deposit}(\texttt{p1}, \_)) \in \mathcal{R}each(M * M', ctxt, code)$$

The assertion $(ctxt', \texttt{p1.deposit}(\_, \_)) \in \mathcal{R}each(M * M', ctxt, code)$ expresses that execution of the configuration $(ctxt, code)$ will reach a point where it calls the method `deposit` on the receiver `p1` – *c.f.* App. A.5.

The fourth policy **Pol_4**, "No one can affect the balance of a purse they don't have", says that if some runtime configuration affects the balance of some purse `prs`, then the original runtime configuration must have had access to the `prs` itself.

Module M satisfies policy **Pol_4**
iff
$$\forall\, M', (ctxt, code) \in \mathcal{A}rising(M * M').\;\; \forall \texttt{prs} :_{ctxt} \texttt{Purse}:$$
$$M * M', ctxt \rightsquigarrow ctxt', \texttt{v} \;\wedge\; \texttt{prs.balance}_{ctxt} \neq \texttt{prs.balance}_{ctxt'}$$
$$\Rightarrow$$
$$\mathit{HasAccess}(M * M', ctxt, code, \texttt{prs})$$

Note that we have not yet specified the meaning of $\mathit{HasAccess}(M * M', ctxt, code, \texttt{prs})$. We will discuss the possible meanings for this together with the second policy:

The second policy. **Pol_2**, stating "Only someone with the mint of a given currency can violate conservation of that currency.", is along the pattern of **Pol_4**, in that it mandates that certain changes of state (here change in currency) may only happen if the originating context had some property (here access to the mint).

Module M satisfies policy **Pol_2**
iff
$$\forall\, M',\, (ctxt, code) \in \mathcal{A}rising(M * M').\ \forall \texttt{mnt} :_{ctxt} \texttt{Mint}:$$
$$M * M', ctxt, code \rightsquigarrow ctxt', \texttt{v} \ \wedge\ Currency_{ctxt}(\texttt{mnt}) \neq Currency_{ctxt'}(\texttt{mnt})$$
$$\Rightarrow$$
$$HasAccess(M * M', ctxt, code, \texttt{mnt})$$

We now need to fix the meaning of $HasAccess(M * M', ctxt, code, \texttt{mnt})$. We have the following three candidates:

1. $\texttt{mnt} \in \mathcal{A}ccAll(M, cxtx)$, i.e. that $ctxt$ has a path from the stack frame to `mnt` which involves any fields.
2. $\texttt{mnt} \in \mathcal{U}ses(M, cxtx, code)$, i.e. that execution of $code$ in the context of $ctxt$ will at some point use the object `mnt`.
3. $\texttt{mnt} \in \mathcal{A}ccPub(cxtx, code)$, i.e. that $ctxt$ has a path from the stack frame to `mnt` which involves only public fields, or private fields from the same class as the current recover.

Given lemma 1 from App. A.8, the choices 2) and 3) give stronger guarantees, and are therefore to be preferred.

The module $M_{\texttt{Purse}} * M_{\texttt{Mint}}$ satisfies **Pol_2** with meaning 2) – and therefore also with meaning 1), by application of lemma 1. A proof sketch for why $M_{\texttt{Purse}} * M_{\texttt{Mint}}$ satisfies **Pol_2** with meaning 2) appears in section 5.

Interestingly, $M_{\texttt{Purse}} * M_{\texttt{Mint}}$ does *not* satisfy **Pol_2** with meaning 3). More importantly, without the concept of package and package-local classes, or some concept of ownership, it is impossible to write an implementation for `Purse` so that it satisfies **Pol_2** with meaning 3). Namely, we can always write another class `Cheat` with a private field `myMint` of class `Mint`, and which leaks this field through a public method `leak`. Then, in a context where `x` points to a `Cheat` object, the code snippet `new Purse(x.leak(), 300)`, affects the currency of `x.myMint`, even though the initial context did not have public access to `x.myMint`. More discussion in section 6.3.

The sixth policy – "A reported successful deposit can be trusted as much as one trusts the purse one is depositing into" – relates to trust, a question left to further work.

4.3 Capability Policy Specification Triples

The meanings of policies given in the previous section vary, but they share common characteristics:

- They have the form that execution of some code under some conditions, guarantees some conclusion (except for **Pol_2,vrs2**).
- Conditions may refer to properties of the state before as well as after execution.
- The code may be universally or existentially quantified, or explicitly given.
- Conclusions may refer to properties of the state before as well as after execution.

We propose *policy triples*, consisting of three components. The first and third component stand for the conditions and conclusions mentioned above, and they describe properties of the context before as well as after the execution. In order to distinguish between the latter two, we use the subscripts `OLD` and `NEW`. The second component may fully specify some code, or quantify over all possible codes `ANY`, or just require that some code exists `SOME`.

In the syntax from below, PL_Code stands for programming panguage code, $Func$ stands for user-defined functions (such as $Currency$) as well as system defined functions (such as $\mathcal{A}ccAll$ or $\mathcal{U}sed$), and $Pred$ stands for user or system-defined predicates (such as $\in$ or $\geq$).

$$
\begin{array}{lll}
Policy & ::= & PolicyTriple* \\
PolicyTriple & ::= & < \ \ Cond, Code, Cond \ \ > \\
Cond & ::= & Pred(Entity^*) \mid Cond \wedge Cond \mid Cond \vee Cond \\
Entity & ::= & Arg_{Annot} \mid Func(\ Arg*)_{Annot} \\
Annot & ::= & \texttt{OLD} \mid \texttt{NEW} \\
Arg & ::= & Path \mid Value \\
Code & ::= & \texttt{ANY} \mid \texttt{SOME} \mid PL\_Code
\end{array}
$$

In figure 2 we express some of the policies through policy triples. Requirements such as $\texttt{mnt} : {}_{\texttt{OLD}}\texttt{Mint}$ are left implicit, as they follow from the use of the identifiers, e.g. from $Currency(\texttt{mnt})_{\texttt{OLD}}$.

Our triples are a generalisation of Hoare logics. Namely,

- The assumptions and the conclusions may refer to properties of the runtime context before as well as after execution, while in Hoare triples the assumptions may only refer to properties of the context before execution of the code.
- The second component may specify the code exactly (as is Hoare triples), or through existential or universal quantification.
- The properties under consideration may contain accessibility or tracing properties.
- The interpretation of the policy triples is always open, i.e. there is an implicit quantification over modules M' extending the module M expected to satisfy the policy.

A module M satisfies a triple $< Cond_1, Cd_1, Cond_2 >$ iff for all M', execution of Cd_1 with code $M * M'$ in a context $ctxt$ leading to $ctxt'$, then if $Cond_1$ holds on $ctxt$ and $ctxt'$, then $Cond_2$ holds on $ctxt$ and $ctxt'$. In further work we will define a mapping from such triples to their interpretation.

5 Towards Reasoning about Capability Policies

The main challenges in reasoning about programs' adherence to capability policies are the deny elements of policies, and the combination of rely and deny steps. We have no

$$\begin{array}{lll}
\textbf{Pol_1, vrs2} & \equiv (& \texttt{p1.mint}_{\texttt{OLD}} = \texttt{p2.mint}_{\texttt{OLD}} \ \wedge \ \texttt{p2.balance}_{\texttt{OLD}} \geq \texttt{m}, \\
& & \texttt{SOME}, \\
& & \texttt{p1.balance}_{\texttt{NEW}} = \texttt{p1.balance}_{\texttt{OLD}} + \texttt{m} \ \ \wedge \\
& & \texttt{p2.balance}_{\texttt{NEW}} = \texttt{p2.balance}_{\texttt{OLD}} + \texttt{m} \qquad) \\
\textbf{Pol_2} & \equiv (& Currency(\texttt{mnt})_{\texttt{OLD}} \neq Currency(\texttt{mnt})_{\texttt{NEW}}, \\
& & \texttt{ANY}, \\
& & \texttt{mnt}_{\texttt{OLD}} \in \mathcal{U}sed \qquad) \\
\textbf{Pol_3} & \equiv (& \texttt{true}, \\
& & \texttt{ANY}, \\
& & Currency(\texttt{mnt})_{\texttt{OLD}} \leq Currency(\texttt{mnt})_{\texttt{NEW}} \qquad) \\
\textbf{Pol_4} & \equiv (& \texttt{prs.balance}_{\texttt{OLD}} \neq \texttt{prs.balance}_{\texttt{NEW}}, \\
& & \texttt{ANY}, \\
& & \texttt{prs}_{\texttt{OLD}} \in \mathcal{U}sed \qquad) \\
\textbf{Pol_5} & \equiv (& \texttt{true}, \\
& & \texttt{ANY}, \\
& & \texttt{prs.balance}_{\texttt{NEW}} \geq 0 \qquad)
\end{array}$$

Fig. 1. Some of the five policies expressed through policy triples

full logics yet, but have initial ideas, which we discuss in terms of our example. We consider briefly the code from Fig.1, and sketch the proof for **Pol_1**, and **Pol_2**. The use of restrictive elements is crucial in reasoning about the Joe-E code. The arguments have an flavour of *abduction*, in that we argue that if something were to happen, then the code or the context in which it happened needs to have a certain property.

Verification sketch of **Pol_1** *in Joe-E/Java* The treatment of **Pol_1** requires nothing more than standard Hoare Logic: if `prs1` and `prs2` share `mints`, then a call of

$$\texttt{prs1.deposit}(\texttt{prs2}, \texttt{amt})$$

transfers `amt` from `prs2` to `prs1` (for appropriate amounts and balances). This can be established by using the usual extension of Hoare Logics to reason about object oriented programs [30, 38]. The treatment of **Pol_1, vrs2** then follows directly, by replacing the existentially quantified *code* by $\texttt{prs1.deposit}(\texttt{prs2}, \texttt{amt})$.

Verification sketch of **Pol_2** *in Joe-E/Java* In contrast, **Pol_2** requires a novel kind of reasoning, which involves the calculation of necessary, rather then sufficient conditions for certain effects to happen. Verification of **Pol_2** can be done through the following steps, annotated by (S1), ... (S11) below:

Assume that $Currency(\texttt{mnt})_{ctx} \neq Currency(\texttt{mnt})_{ctxt'}$. (S1): Then, by the definition of $Currency$, either $Ps(\texttt{mnt})_{ctxt} \neq Ps(\texttt{mnt})_{ctxt'}$, or there exists some $\texttt{prs} \in Ps(\texttt{mnt})_{ctxt}$, such that $\texttt{prs.balance}_{ctxt} \neq \texttt{prs.balance}_{ctxt'}$.

1st Case: $Ps(\texttt{mnt})_{ctxt} \neq Ps(\texttt{mnt})_{ctxt'}$. (S2): Then, either $\exists\ \texttt{prs} \in Ps(\texttt{mnt})_{ctxt} \setminus Ps(\texttt{mnt})_{ctxt'}$, or $\exists\ \texttt{prs} \in Ps(\texttt{mnt})_{ctxt'} \setminus Ps(\texttt{mnt})_{ctxt}$

1.1st Case: $\exists\, \texttt{prs} \in Ps(\texttt{mnt})_{ctxt} \setminus Ps(\texttt{mnt})_{ctxt'}$. (S3): Then, because there is no garbage collection, $\texttt{prs.mint}_{ctxt} \neq \texttt{prs.mint}_{ctxt'}$. (S4): However, this is impossible because the field `mint` is `final`.

1.2nd Case: $\exists\, \texttt{prs} \in Ps(\texttt{mnt})_{ctxt'} \setminus Ps(\texttt{mnt})_{ctxt}$. (S5): If `prs` was defined in $ctxt$ then $\texttt{prs.mint}_{ctxt} \neq \texttt{prs.mint}_{ctxt'}$, which is impossible as in the first case. (S6): If `prs` was undefined in $ctxt'$, then it must have been created through calling one of the two `Purse` constructors. (S7) The constructor $\texttt{Purse}(\texttt{Purse})$ does not affect the currency of the mint. (S8) The constructor $\texttt{Purse}(\texttt{Mint}, \texttt{balance})$ does effect the currency of the mint, but does require access to the mint object, i.e. "uses" it.

2nd Case: $Ps(\texttt{mnt})_{ctxt} = Ps(\texttt{mnt})_{ctxt'}$, and $\exists\, \texttt{prs} \in Ps(\texttt{mnt})_{ctxt}$, such that $\texttt{prs.balance}_{ctxt} \neq \texttt{prs.balance}_{ctxt'}$. (S9): Since the class `Purse` is final, and the fuel `balance` is `private`, the only way to modify this fields is through method in `Purse`. (S10): The only method that changes a `Purse`'s balance is `deposit`. (S11) However, the method `deposit` leaves the currency of a `Mint` constant.

In summary, the argument requires eleven steps (S1)-(S11). Of these steps, three are about normal bookkeeping and set theoretic manipulations (S1,S2,S5), four are using restrictive features (S2, S3, S4,S9), one analyses the structure if the code (S6), and three analyse the semantics of the method bodies (S7,S8,S11).

6 Discussion

In formalising the Mint capability policies, we came across a number of other potential policies that were not captured explicitly in the original six policies.

6.1 Fresh Mints & Purses

Pol_7A Only fresh mint objects are returned from mints and purses.
Pol_7B Only fresh purse objects are returned from mints and purses.

By a "fresh" object, we mean a newly allocated object that has not previous been returned out of the mint-and-purse system. These policies are trivially satisfied by the code in Fig. 2: the construct that returns mints is the default constructor of the `Mint` class, and by the semantics of Java, this will always return a new object. Similarly, the only two methods that return a purse object are the two constructors of the `Purse` class.

Although not explicit, these polices seem to be required implicitly for the other policies to be useful to clients of the Mint system. Consider **Pol_2**: *"Only someone with the mint of a given currency can violate conservation of that currency"* and **Pol_4**: *"No one can affect the balance of a purse they don't have."* These policies talk about having a mint or a purse, but don't say anything about how that mint or purse was obtained. These policies only make sense if the mint and purse capabilities are strictly controlled. The underlying assumption is that a capability effectively belongs to whichever client first requested it, and that only this client can distribute the capability further. **Pol_7A** and **Pol_7B** make this ownership assumption explicit.

These "fresh obejct" policies may appear trivial because of the simplicity and relatively small size of the mint and purse system. But even a small change to the implementation of the system can break these policies. For example, adding a public `getMint` accessor that returns a purses's mint (useful, perhaps, for clients with purses from multiple mints) would break policy **Pol_7B** and destroy the integrity of the whole system.

6.2 Subsidiary Capabilities

The simplicity of the Mint example code obscures the need for an additional policy:

Pol_8 Objects implementing Purses and Mints must never be exposed to clients.

Consider an alternative implementation of Purses, where the Mint contained a map from every Purse to its balance. This design is in some sense the complement of Fig. 2: where that design has a `Mint` class that is a pure capability ("`class Mint(){ }`", with no state or behaviour), here the Purse objects would be pure capabilites and the state (the map from purses to their balances) and behaviour (the `deposit` method) would be within the Mint class. The risk this offers is that the original six policies do not mention the internal map object (neither do 7A & B) so an accessor that returned that map would not breach any policy, even though a client with the map capability could do essentially anything to a mint and all its purses. The issue here is another assumption: that there no subsidiary objects in the implementation. Policies such as **Pol_8** ensure that the system will remain secure even if the implementation is changed to require subsidiary objects.

6.3 Supersidiary Capabilities

The third issue we consider relates to supersidiary capabilties, that is, when a Mint or a Purse is combined into a larger structre. In a crucial sense, the Fig. 2 design already involves supersidiary capabilities: each Purse holds the capability to its mint.

The problem this rasies can be see with respect to **Pol_2**: *"Only someone with the mint of a given currency can violate conservation of that currency"*. Purses have mints. So it follows that purses may violate currency conservation. We can guard against this specific aspect with a final policy:

Pol_9 Purses objects must not manipulate their mint capabilities to inflate the currency.

but we can only know we have to write this policy because the purse objecst are effectively subsidiary to the module containing all the declarations in the system. The difficulty is much harder to address when a mint, say is part of some supersidiary system — a `Country` object perhaps:

```
1 class Country {
2   private final Mint myMint = new Mint();
3   private final Purse myTreasury = new Purse();
4   public void inflate() {
5     Purse tmpPurse = new Purse(myMint,1000000000)
6     myTreasury.deposit(tmpPurse,1000000000)
7   }
```

A `Country` has a mint, and its treasury (a purse belonging to that mint). The `inflate` method creates a new temporary purse containing a billion dollars from thin air, and then deposits that into the treasury — Perhpas this method should have been called `quantitativeEasing`. Now consider a supersidiary client of a `Country` object — the finance minister say. The finance minister does *not* have a reference to the mint (presumably there is an independent central bank) so by **Pol_2** she should not be able to inflate the currency. If, however, the finance minister calls `myCountry.inflate` then the currency will be inflated all the same. (This is the complementary situation to the Purse holding the mint capability and not inflating; here the finance minister doesn't hold the mint capability but does inflate). Formally, these distinctions seem to depend upon the precise semantics of $HasAccess$ (see section 4.2). Elucidating these dependencies in detail we leave to future work.

7 Related Work

Object-capabilities were first introduced [24] seven years ago, and many recent studies manage or verify safety or correctness of object-capability programs.

Google's Caja [27] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* capabilities. Sandboxing has been validated formally: Maffeis et al. [19] develop a model of Javascript, demonstrate that it obeys two principles of object-capability systems and show how untrusted applications can be prevented from interfering with the rest of the system. Alternatively, Taly et al. [44] model JavaScript APIs in Datalog, and then carry out a Datlog search for an "attacker" from the set of all valid API calls. This search is similar to the quantification over potential code snippets in our model. Murray and Lowe [28] model object-capability programs in CSP, and use a model checker to ensure program executions do not leak information.

Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [15], including pointer analyses. Bhargavan et al. x[4] extend language-based sandboxing techniques to support "defensive" components that can execute successfully in otherwise untrusted environments. Meredith et al. [21] encode policies as types in higher order reflective π-calculus.. Politz et al. [33] use a JavaScript typechecker to check properties such as *"multiple widgets on the same page cannot communicate."* — somewhat similar in spirit to our **Pol_4**. Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [18]. Dimoulas et al. [8] generalise the language and typeckecher based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities.

The WebSand [5, 20] and Jeeves [51] projects use dynamic techniques to monitor safe execution of policies. Richards et al. [36] extended this approach by incorporating explicit dynamic ownership of objects (and thus of capabilities), and policies that may examine the history of objects' computations. While these dynamic techniques can restrict or terminate the execution of a component that breaches its security policies, they cannot guarantee in advance that such violations can never happen.

Compared with all these approaches, our work focuses on *general* tecnhiques for specifying (and ultimately verifying) capability policies, whereas these systems are

generally much more *specific*: focusing on one (or a small number) of actual policies. A few formal verification frameworks address JavaScript's highly dynamic, prototype-based semantics. Gardner et al. [11] developed a formalisation of JavaScript based on separation logic and verified examples. Xiong and Qin et al. [50, 34] worked on similar lines. Swamy et al. [43] recently developed a mechanised verification technique for JavaScript based on the Dijkstra Monad in the F* programming language. Finally, Jang et al. [14] developed a machine-checked proof of five important properties of a web browser — again similar to our simple deny policies — such as *"cookies may not be shared across domains"* by writing the minimal kernel of the browser in Coq.

8 Conclusions and Future Work

In paper, we have advocated that capability policies are necessary for reasoning about programs using object-capability security. We have argued that capability policies must be program centred, fine grained, open, and support both rely and deny elements.

These novel features of the policies require novel features in specifications. We have proposed capability triples to specify policies, which incorporate quantification over program code (to model open policies) predicates over pre- and post- conditions (to handle deny conditions), and by describing paths through programs, are centre upon those programs' designs. Finally, we have sketched out informal arguments of how adherence to policies may be argued, and we have shown how efforts at specifying policies precisely can uncover additional implicit policies that can be made explicit.

The arguments we have used do not fit the Hoare Logic nor the type-inference format. Nevertheless, they reflect the way one informally reasons about code. They argue in terms of the footprint of a property, and of the set of method calls which might affect that footprint. They consider the uses of restrictive language features (*e.g.* `final`) in the program to reduce that set. They also use rely reasoning (*e.g.* calls to `deposit` or `Purse`(`Purse`) preserve the currency in the mint).

We want to develop a formal logic to support reasoning about capability policies. Such a logic will need to combine both rely and deny steps. It will have the usual Hoare Logic rules, as well as inference rules for the calculation of footprints of properties, the effect of restrictive features, for the passing of object capabilities, for lexically scoped languages. To prove soundness of our logic [35] we will need to expand the approach to deal with the deny arguments, perhaps applying ideas from provenance [32], and considerate reasoning [41].

References

1. Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.
2. Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *PLDI*, 2005.
3. Jesper Bengtson, Kathiekeyan Bhargavan, Cedric Fournet, Andrew Gordon, and S.Maffeis. Refinement Types for Secure Implementations. *ACM ToPLAS*, 2011.

4. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.
5. Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Programming Languages and Analysis for Security (PLAS)*, 2011.
6. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*. ACM, 1998.
7. Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.
8. Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. 2013.
9. Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*. Springer, 2009.
10. Sophia Drossopoulou and James Noble. The need for capability policies. In *(FTfJP)*, 2013.
11. Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, 2012.
12. Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *MFPS*. Elsevier, ENTCS, 2001.
13. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
14. Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.
15. Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-Chieh Shan. An Analysis of the Mozilla Jetpack Extension FrameworK. In *ECOOP*, Springer, 2012.
16. Butler W. Lampson and Howard E. Sturgis. Reflection on an Operating System Design. *Communications of the ACM*, 19(5), 1976.
17. K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, 2007.
18. Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, September 2013.
19. S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy*, 2010.
20. Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
21. L.G. Meredith, Mike Stay, and Sophia Drossopoulou. Policy as types. arXiv:1307.7766 [cs.CR], July 2013.
22. Adrian Mettler, David Wagner, and Tyler Close. Joe-E a Security-Oriented Subset of Java. In *NDSS*, 2010.
23. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
24. Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.
25. Mark Samuel Miller. Secure Distributed Programming with Object-capabilities in JavaScript. Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com, October 2011.
26. Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.
27. Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.
28. Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.
29. Roger Needham. Protection systems and protection implementations. In *Joint Computer Conference*, pages 571–578, 1972.

30. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.
31. Matthew Parkinson. Class invariants: the end of the road? In *IWACO*, 2007.
32. Roly Perera, Umut Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *ICFP*. ACM, 2012.
33. Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of javascript sandboxing. In *USENIX Security*, 2011.
34. Shengchao Qin, Aziem Chawdhary, Wei Xiong, Malcolm Munro, Zongyan Qiu, and Huibiao Zhu. Towards an axiomatic verification system for javascript. In *TASE*, pages 133–141, 2011.
35. Azalea Raad and Sophia Drossopoulou. A Sip of the Chalice. In *FTFJP*, July 2011.
36. Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible access control for JavaScript. In *OOPSLA*, pages 305–322, 2013.
37. Jerome H. Saltzer. Protection and the control of information sharing in Multics. *CACM*, 17(7):p.389ff, 1974.
38. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames. *ToPLAS*, 2012.
39. Marc Stiegler. The lazy programmer's guide to security. HP Labs, www.object-oriented-security.org.
40. Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.
41. Alexander J. Summers and Sophia Drossopoulou. Considerate Reasoning and the Composite Pattern. In *VMCAI*, 2010.
42. Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for Flexible Object Invariants. In *IWACO*, ACM DL, July 2009.
43. Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *PLDI*, pages 387–398, 2013.
44. Ankur Taly, Ulfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy (SP)*, 2011.
45. The NewpeakTeam. Several Newspeak Documents. `newspeaklanguage.org/`, September 2012.
46. Tom van Cutsem. Membranes in Javascript. prog.vub.ac.be/ tvcutsem/invokedynamic/js-membranes.
47. M. V. Wilkes and R. M. Needham. The Cambridge CAP computer and its operating system, 1979.
48. Niklaus Wirth. *Programming in Modula-2*. Springer, 1982.
49. T. Wood and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.
50. Wei Xiong. *Verification and Validation of JavaScript*. PhD thesis, 2013, Durham University.
51. Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.

A Formal Definition of $\mathcal{L}ng_{\mathtt{cj}}$

A.1 Modules and Syntax

Modules map class identifiers to their superclass, field definitions and method declarations. Extensibility annotations, *ea*, are attached to classes and methods, and they forbid (noext) or allow (exts) subclasses, resp. redefinition of the method in a subclass. Privacy annotations (*pa*) are attached to fields and methods, and restrict (priv) or allow

(pub) access outside the defining class. Mutability annotations, *ma*, make fields immutable (fin) or mutable (vol). We expect that constructors will be defined as methods with the identifier `constr`.

$$
\begin{array}{lll}
\mathit{Module} & = & \mathit{ClassId} \longrightarrow (\ \mathit{ea} \quad \times \\
 & & \qquad \mathit{ClassId} \quad \times \\
 & & \qquad (\mathit{FieldId} \longrightarrow \mathit{ma}\ \mathit{pa}\ \mathit{ClassId}\ ) \quad \times \\
 & & \qquad (\mathit{MethId} \longrightarrow \mathit{ea}\ \mathit{pa}\ \mathit{methBody}\ ) \quad)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{ea} & ::= & \mathsf{noext} \mid \mathsf{exts} \\
\mathit{pa} & ::= & \mathsf{priv} \mid \mathsf{pub} \\
\mathit{ma} & ::= & \mathsf{fin} \mid \mathsf{mut} \\
\mathit{meth} & ::= & \mathit{ClassId}\ m\ \ (\ \mathit{ClassId}\ \mathtt{x}\ )\ \ \{\ e\ \} \\
\text{Used } e & ::= & e.f \mid e.f := e \mid e.m\,(\,e\,) \mid \mathsf{new}\ c \mid \mathtt{x} \mid \mathtt{this} \mid \mathtt{null}
\end{array}
$$

A.2 The Type System of $\mathcal{L}ng_{\mathtt{cj}}$

The lookup functions $\mathcal{F}$ and $\mathcal{M}$ return corresponding field and method definitions.

$$
\mathcal{F}(M,\mathtt{c},\mathtt{f}) \quad = \quad \begin{cases} \text{undefined} & \text{if } \mathtt{c} = \mathtt{Object}, \\ M(\mathtt{c})\downarrow_3(\mathtt{f}) & \text{if} \quad M(\mathtt{c})\downarrow_3(\mathtt{f}) \text{ is defined}, \\ \mathcal{F}(M, M(\mathtt{c})\downarrow_1, \mathtt{f}) & \text{otherwise.} \end{cases}
$$

$$
\mathcal{M}(M,\mathtt{c},\mathtt{m}) \ = \quad \begin{cases} \text{undefined} & \text{if } \mathtt{c} = \mathtt{Object}, \\ M(\mathtt{c})\downarrow_3(\mathtt{m}) & \text{if} \quad M(\mathtt{c})\downarrow_3(\mathtt{m}) \text{ is defined}, \\ \mathcal{M}(M, M(\mathtt{c})\downarrow_1, \mathtt{m}) & \text{otherwise.} \end{cases}
$$

Typing is expressed through the judgment $\Gamma \vdash \mathtt{e} : \mathtt{c}$, where Γ is a mapping from $\{\mathtt{this}, \mathtt{x}\}$ to the set of class identifiers, and from `constr` to a boolean (to indicate whether Γ belongs to a constructor's method body).

VarThisNull

$$
\frac{M(\mathtt{c}) \text{ is defined}}{M, \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})} \qquad M, \Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this}) M, \Gamma \vdash \mathtt{null} : \mathtt{c}
$$

fldRd

$$
\frac{\begin{array}{l} M, \Gamma \vdash \mathtt{e} : \mathtt{c} \\ \mathcal{F}(M, \mathtt{c}, \mathtt{f}) \ = \ ma\ pa\ \mathtt{c}' \\ pa = \mathsf{pub} \vee \Gamma(\mathtt{this}) = \mathtt{c} \end{array}}{M, \Gamma \vdash \mathtt{e.f} : \mathtt{c}'}
$$

fldAss

$$
\frac{\begin{array}{l} M, \Gamma \vdash \mathtt{e} : \mathtt{c} \\ \mathcal{F}(M, \mathtt{c}, \mathtt{f}) \ = \ ma\ pa\ \mathtt{c}'' \\ M, \Gamma \vdash \mathtt{e}' : \mathtt{c}' \\ ma = \mathsf{mut} \wedge (pa = \mathsf{pub} \vee \Gamma(\mathtt{this}) = \mathtt{c}) \\ M \vdash \mathtt{c}' \leq \mathtt{c}'' \end{array}}{M, \Gamma \vdash \mathtt{e.f} := \mathtt{e}' : \mathtt{c}'}
$$

fldInitConstr

$$
\frac{\begin{array}{l} \Gamma(\mathtt{this}) = \mathtt{c} \\ \mathcal{F}(M, \mathtt{c}, \mathtt{f}) \ = \ \mathsf{immut}\ pa\ \mathtt{c}'' \\ \Gamma(\mathsf{constr}) = true \\ M, \Gamma \vdash \mathtt{e}' : \mathtt{c}' \\ M \vdash \mathtt{c}' \leq \mathtt{c}'' \end{array}}{M, \Gamma \vdash \mathtt{this.f} := \mathtt{e}' : \mathtt{c}'}
$$

constrCall

$$
\frac{\begin{array}{l} \mathcal{M}(M, \mathtt{c}, \mathsf{constr}) \ = \ \_\ pa\ \mathtt{c}\ \mathsf{constr}(\ \_\ \_)\ \{\ \mathtt{e}\ \} \\ M, \Gamma \vdash \mathtt{e} : \mathtt{c}'' \\ pa = \mathsf{pub} \vee \Gamma(\mathtt{this}) = \mathtt{c} \\ M \vdash \mathtt{c}'' \leq \mathtt{c}' \end{array}}{M, \Gamma \vdash \mathsf{new}\,\mathtt{c}(\,\mathtt{e}\,) \ : \ \mathtt{c}}
$$

methCall

$$
\frac{\begin{array}{l} M, \Gamma \vdash \mathtt{e}_0 : \mathtt{c} \qquad M, \Gamma \vdash \mathtt{e}_1 : \mathtt{c}_1 \\ pa = \mathsf{pub} \vee \Gamma(\mathtt{this}) = \mathtt{c} \\ \mathcal{M}(M, \mathtt{c}_0, \mathtt{m}) \ = \ ea\ pa\ \mathtt{c}'\ \mathtt{m}(\ \mathtt{c}_3\ \mathtt{x})\ \{\ \_\ \} \\ M \vdash \mathtt{c}_1 \leq \mathtt{c}_3 \end{array}}{M, \Gamma \vdash \mathtt{e}_0.\mathtt{m}(\,\mathtt{e}_1) \ : \ \mathtt{c}'}
$$

A.3 Well-formed class and module, linking

well_formed_class

$$\frac{\begin{array}{l} M(\mathtt{c}) = \ ea\ \mathtt{c}'... \\ \mathtt{c}' = \mathtt{Object} \vee (ea = \mathsf{exts} \wedge M(\mathtt{c}') \text{ is defined}) \\ \forall \mathtt{f}: \ M(\mathtt{c}) \downarrow_3 (\mathtt{f}) = \mathtt{c}' \implies M(\mathtt{c}') \text{ is defined} \ \wedge \ \mathcal{F}(M, \mathtt{c}', \mathtt{f}) \text{is not defined} \\ \forall \mathtt{m}: \ M(\mathtt{c}) \downarrow_4 (\mathtt{m}) = \ ea'\ pa\ \mathtt{c}'\ \mathtt{m(\ c''\ x)\ \{\ e\ \}} \implies \\ \qquad M(\mathtt{c}'), M(\mathtt{c}'') \text{ are defined} \\ \qquad \Gamma(\mathtt{this}) = \mathtt{c},\ \Gamma(\mathtt{x}) = \mathtt{c}'',\ \Gamma(\mathsf{constr}) = (\mathtt{m} = \mathsf{constr}) \\ \qquad M, \Gamma \vdash \mathtt{e} : \mathtt{c}''' \qquad M \vdash \mathtt{c}''' \leq \mathtt{c}' \\ \qquad \mathcal{M}(M, \mathtt{c}', \mathtt{m}) \text{ is undefined} \ \vee \ (ea' = \mathsf{exts} \wedge \mathcal{M}(M, \mathtt{c}', \mathtt{m}) = \ ea'\ pa\ \mathtt{c}'\ \mathtt{m(\ c''\ x)\ \{\ \_\ \}}) \end{array}}{M \vdash \mathtt{c} \diamond}$$

$$\vdash M \diamond \qquad \text{iff} \qquad \forall \mathtt{c} \in dom(M): \ M \vdash \mathtt{c} \diamond$$

Linking of two modules M and M' is the union of their respective mappings, provided that the domains of the two modules are disjoint, and that the union of the mappings creates a well-formed module: $* : Module \times Module \longrightarrow Module$

$$M{*}M' = \begin{cases} M *_{aux} M', \text{ if} \vdash M *_{aux} M' \diamond, \text{and } dom(M) \cap dom(M') = \emptyset \\ \text{undefined}, \quad \text{otherwise.} \end{cases} \text{ where}$$

$$(M *_{aux} M')(\mathtt{c}) \ = \ M(\mathtt{c}), \text{if } M(\mathtt{c}) \text{ is defined}, \quad M'(\mathtt{c}), \text{otherwise.}$$

A.4 Runtime context

The runtime context consists of a stack frame, ϕ, and a heap χ. The frame is a tuple consisting of the address belonging to the receiver (`this`), and the value for the argument (`x`). Values are addresses or `null`). Addresses are ranged over by ι, and they are natural numbers. The heap maps addresses to objects. Objects are tuples consisting of the class of the object, and a mapping field identifiers onto values.

ctxt	=	*frame* × *heap*	$\phi \in$ *frame*	=	*addr* × *val*
$\chi \in$ *heap*	=	*addr* ⟶ *object*	$\mathtt{v} \in$ *val*	=	{ `null` } ∪ *addr*
object	=	*ClassId* × (*FieldId* ⟶ *val*)	$\iota \in$ *addr*	=	$\mathbb{N}$

The Operational Semantics of $\mathcal{L}ng_{\mathtt{cj}}$ Execution uses module M, and maps a runtime context $ctxt$ and expression `e` (code in the general case) onto a new context $ctxt'$ and a result.

$\rightsquigarrow$: *Module* × *ctxt* × *expr* ⟶ *ctxt* × *res*

res = { `nullPntrExc`, `stuckErr` } ∪ *val*

Here the salient rules for execution; we omit those which raise & propagate exceptions.

val

$$\frac{}{M,\ ctxt,\ \mathtt{v} \rightsquigarrow\ ctxt,\ \mathtt{v}}$$

thisPar

$$\frac{ctxt = (\iota, \mathtt{v}), \chi}{\begin{array}{l} M,\ ctxt,\ \mathtt{this} \rightsquigarrow\ ctxt,\ \iota \\ M,\ ctxt,\ \mathtt{x} \rightsquigarrow\ ctxt,\ \mathtt{v} \end{array}}$$

fldRd

$$\frac{\begin{array}{l} M,\ ctxt,\ \mathtt{e} \rightsquigarrow\ ctxt',\ \iota \\ ctxt = \_, \chi' \quad \chi'(\iota) \downarrow_2 (\mathtt{f}) = \mathtt{v} \end{array}}{M,\ ctxt,\ \mathtt{e.f} \rightsquigarrow\ ctxt',\ \mathtt{v}}$$

fldAss

$$\frac{\begin{array}{l} M,\ ctxt,\ \mathtt{e} \rightsquigarrow\ ctxt'',\ \iota \\ M,\ ctxt'',\ \mathtt{e}' \rightsquigarrow\ ctxt''',\ \mathtt{v} \\ ctxt''' = \phi''', \chi''', \ \chi' = \chi'''[\iota \mapsto \chi'''(\iota)[\mathtt{f} \mapsto \mathtt{v}]] \end{array}}{M,\ ctxt,\ \mathtt{e.f} := \mathtt{e}' \rightsquigarrow\ (\phi''', \chi'),\ \mathtt{v}}$$

new

$$\frac{\begin{array}{l} M,\, ctxt,\, \mathtt{e} \rightsquigarrow ctxt'',\, \mathtt{v} \\ \{\mathtt{f} \mid \mathcal{F}(M,\mathtt{c},\mathtt{f}) \text{ is defined } \} = \{\mathtt{f}_1,\ldots,\mathtt{f}_r\} \\ \iota \text{ is new in } \chi'' \\ ctxt'' = \phi,\chi'' \\ \chi''' = \chi''[\iota\mapsto(\,\mathtt{c},\ (\mathtt{f}_1 : \mathtt{null}\,,\ldots,\mathtt{f}_r : \mathtt{null}\,)\,)\,] \\ \mathcal{M}(M,\mathtt{c},\mathtt{constr}) = \_\,\_(\,\_\_)\,\{\,\mathtt{e}'\} \\ ctxt''' = (\iota,\mathtt{v}),\chi''' \\ M,\, ctxt''',\, \mathtt{e}' \rightsquigarrow ctxt',\, \mathtt{v}' \end{array}}{M,\, ctxt,\, \mathsf{new}\,\mathtt{c}(\,\mathtt{e})\ \rightsquigarrow ctxt',\, \iota}$$

methCall

$$\frac{\begin{array}{l} M,\, ctxt,\, \mathtt{e} \rightsquigarrow ctxt'',\, \iota \\ M,\, ctxt'',\, \mathtt{e}' \rightsquigarrow ctxt''',\, \mathtt{v} \\ ctxt''' = (\iota,\mathtt{v}),\chi''' \\ \chi'''(\iota)\downarrow_1 = \mathtt{c} \\ \mathcal{M}(M,\mathtt{c},\mathtt{m}) = \_\_\_\,m(\,\_\_rp\ \{\ \mathtt{e}''\,\} \\ ctxt'''' = (\iota,\mathtt{v}),\,\chi''' \\ M,\, ctxt'''',\, \mathtt{e}'' \rightsquigarrow ctxt',\, \mathtt{v} \end{array}}{M,\, ctxt,\, \mathtt{e}.\mathtt{m}(\,\mathtt{e}')\ \rightsquigarrow ctxt',\, \mathtt{v}}$$

A.5 $\mathcal{R}each$: Reachable Configurations

Configurations consist of runtime contexts and expressions. A configuration is reachable from another configuration, if the former may be required for the evaluation of the former after ny number of steps. We are only interested in the configurations which correspond to method calls, or constructor invocations. $\mathcal{R}each(M, ctxt, e)$ returns the sets of all configurations reachable from $ctxt, e$:

$$\mathcal{R}each\ :\ Program \times Context \times Expr \mapsto \mathcal{P}(Context \times Expr)$$

We define the function $\mathcal{R}each$ by cases on the structure of the expression, and depending on the execution of the expression. Note that the function $\mathcal{R}each(M, ctxt, expr)$, even when the execution should diverge. This is important, because it allows us to give meaning to capability policies without requiring termination. In case of divergence, $\mathcal{R}each(M, ctxt, \mathsf{null})$ will be an infinite set.

$$\begin{array}{lll}
\mathcal{R}each(M,ctxt,\mathsf{null}) & = \emptyset & \\
\mathcal{R}each(M,ctxt,\mathtt{this}) & = \emptyset & \\
\mathcal{R}each(M,ctxt,\mathtt{x}) & = \emptyset & \\
\mathcal{R}each(M,ctxt,\mathtt{e.f}) & = \mathcal{R}each(ctxt,\mathtt{e}) & \\
\mathcal{R}each(M,ctxt,\mathtt{e.f} := \mathtt{e}') & = \mathcal{R}each(M,ctxt,\mathtt{e}) & \\
 & \quad \cup \begin{cases} \mathcal{R}each(M,ctxt',\mathtt{e}'), & \\ \quad \text{if } \exists ctxt',\mathtt{v},s.t:\ \ M,ctxt,\mathtt{e} \rightsquigarrow ctxt',\mathtt{v} & \\ \emptyset, \quad \text{otherwise.} & \end{cases} & \\
\mathcal{R}each(M,ctxt,\mathsf{new}\,\mathtt{c}(\,\mathtt{e})\,) & = \mathcal{R}each(M,ctxt,\mathtt{e}) & \\
 & \quad \cup \begin{cases} \{ctxt',\mathtt{e}'\} & \\ \quad \text{if } (A): \exists ctxt'',\mathtt{v},s.t: & \\ \quad M,ctxt,\mathtt{e} \rightsquigarrow ctxt'',\mathtt{v}, & \\ \quad \mathcal{M}(M,\mathtt{c},\mathtt{constr}) = ____(\,__)\,\{\,\mathtt{e}'\} & \\ \quad \text{and where } ctxt'' = _,\chi'',\ \text{ and } ctxt' = (\iota,\mathtt{v}),\chi''' & \\ \quad \iota \text{ new in } \chi'',\ \chi''' = \chi''[\iota\mapsto(\,\mathtt{c},\ (\mathtt{f}_1 : \mathtt{null}\,,\ldots,\mathtt{f}_r : \mathtt{null}\,)\,)\,] & \\ \quad \{\mathtt{f} \mid \mathcal{F}(M,\mathtt{c},\mathtt{f}) \text{ is defined } \} = \{\mathtt{f}_1,\ldots,\mathtt{f}_r\} & \\ \emptyset, \quad \text{otherwise.} & \end{cases} & \\
 & \quad \cup \begin{cases} \mathcal{R}each(M,ctxt',\mathtt{e}'), \quad \text{if } (A) \text{ as above.} & \\ \emptyset, \quad \text{otherwise.} & \end{cases} &
\end{array}$$

$$\mathcal{R}each(M, ctxt, \texttt{e.m(e')}) = \mathcal{R}each(M, ctxt, \texttt{e})$$
$$\cup \begin{cases} \mathcal{R}each(M, ctxt', \texttt{e}') \\ \quad \text{if } (B) : \exists ctxt', \iota, s.t : \\ \quad M, ctxt, \texttt{e} \rightsquigarrow ctxt', \iota, \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\cup \begin{cases} \{ctxt''', \texttt{e}''\} \\ \quad \text{if } (B) \text{ as above, and } (C), \exists ctxt'', \texttt{v}, s.t : \\ \quad M, ctxt', \texttt{e}' \rightsquigarrow ctxt'', \texttt{v}, \\ \quad \text{and where } ctxt'' = \_, \chi'', \ \chi''(iota) \downarrow_1 = \texttt{c} \\ \quad \mathcal{M}(M, \texttt{c}, \texttt{m}) = \_\_\_\ \texttt{m(}\ \_\ \_\texttt{)}\ \{\ \texttt{e}''\} \\ \quad \text{and where } ctxt''' = (\iota, \texttt{v}), \chi''. \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\cup \begin{cases} \mathcal{R}each(M, ctxt''', \texttt{e}'') \quad \text{if } (B) \text{ and } (C), \text{ as above.} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$

A.6 $\mathcal{A}rising$: configurations reachable from initial configurations

We define as initial configurations all configurations that may be encountered at the start of program execution.

$$\mathcal{I}nit \ : \ Program \mapsto \mathcal{P}(Context \times Expr)$$

Initial configuration should be as "minimal" as possible, We therefore construct a heap which has only one object. And we require that the expression be well typed under the assumption that `this`and `x` are denoting objects of class `Object`.

$$\begin{aligned} \mathcal{I}nit(M) \ = \ & \{\ (\ ctxt_0, \texttt{e}) \mid \exists \texttt{c}.\ M, \Gamma_0 \vdash \texttt{e} : \texttt{c}\ \} \\ & \text{where } ctxt_0 \ = \ ((\iota_0, \mathsf{null}), \chi_0), \\ & \text{and } dom(\chi_0) = \{\ \iota\ \}, \text{and } \chi_0(\iota_0) = (\texttt{Object}, \emptyset), \\ & \text{and } \Gamma = \texttt{this} \mapsto \texttt{Object}, \texttt{x} \mapsto \texttt{Object}, constr \mapsto \texttt{false}. \end{aligned}$$

The *arising* configurations are those which may be reached by executing an initial configuration:

$$\mathcal{A}rising \ : \ Program \mapsto \mathcal{P}(Context \times Expr)$$

Arising configurations they are defined as follows:

$$\mathcal{A}rising(M) \ = \ \bigcup_{(ctxt, \texttt{e}) \in \mathcal{I}nit(M)} \mathcal{R}each(M, ctxt, \texttt{e})$$

A.7 $\mathcal{U}sed$: the objects reads by execution of a configuration

We are interested in collecting all addresses read during execution of a configuration:

$$\mathcal{U}sed \ : \ Program \times Context \times Expr \mapsto \mathcal{P}(Addr)$$

$\mathcal{U}sed$ is defined by cases on the structure of the expression, and depending on the execution of this expression. We use similar cases as those in the definition in A.5.

$$\mathcal{U}sed(M, ctxt, \mathsf{null}) \ = \emptyset$$
$$\mathcal{U}sed(M, ctxt, \texttt{this}) = \{\phi(\texttt{this})\} \ \text{ where } ctxt = \phi, \_$$
$$\mathcal{U}sed(M, ctxt, \texttt{x}) \ = \{\phi(\texttt{x})\} \ \text{ where } ctxt = \phi, \_$$
$$\mathcal{U}sed(M, ctxt, \texttt{e.f}) \ = \mathcal{U}sed(M, ctxt, \texttt{e}) \cup \begin{cases} \{\chi'(\iota) \downarrow (\texttt{f})\} \\ \quad \text{if } \exists ctxt', \iota, s.t : \\ \quad M, ctxt, \texttt{e} \rightsquigarrow (\phi', \chi'), \iota \\ \emptyset, \quad \text{otherwise.} \end{cases}$$

$$\mathcal{U}sed(M, ctxt, \texttt{e.f} := \texttt{e}') = \mathcal{U}sed(M, ctxt, \texttt{e.f})$$
$$\cup \begin{cases} \mathcal{U}sed(M, ctxt', \texttt{e}'), \\ \quad \text{if } \exists ctxt', \texttt{v}, s.t: \\ \quad M, ctxt, \texttt{e} \rightsquigarrow ctxt', \texttt{v} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\mathcal{U}sed(M, ctxt, \texttt{new c(e)}) = \mathcal{U}sed(M, ctxt, \texttt{e})$$
$$\cup \begin{cases} \{\ \iota\ \} \ \text{ if } (A), \ \text{ where } (A) \text{ as defined in A.5} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\cup \begin{cases} \mathcal{U}sed(M, ctxt', \texttt{e}'), \quad \text{if } (A) \text{ as above.} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\mathcal{U}sed(M, ctxt, \texttt{e.m(e')}) = \mathcal{U}sed(M, ctxt, \texttt{e})$$
$$\cup \begin{cases} \mathcal{U}sed(M, ctxt', \texttt{e}') \ \text{ if } (B), \ \text{ where } (B) \text{ as defined in A.5} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\cup \begin{cases} \{ctxt''', \texttt{e}''\} \\ \quad \text{if } (B) \text{ and } (C), \text{ where } (B), (C) \text{ as defined in A.5} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$
$$\cup \begin{cases} \mathcal{R}each(M, ctxt''', \texttt{e}'') \quad \text{if } (B) \text{ and } (C), \text{ as above.} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$

A.8 $\mathcal{A}ccAll$: objects accessible through any paths, $\mathcal{A}ccPub$: objects accessible through public paths

We collect the addresses of all objects which are accessible from a stack frame through any paths which go through public or through private fields, through the function $\mathcal{A}ccAll$: We also collect the addresses of all objects which are accessible from a stack frame through paths which go through public fields, or through private fields which, however, belong to objects of the same class as the current receiver:

$$\mathcal{A}ccAll \ : \ Program \times Context \mapsto \mathcal{P}(Addr)$$
$$\mathcal{A}ccPub \ : \ Program \times Context \mapsto \mathcal{P}(Addr)$$

These sets are defined as follows:

$$\mathcal{A}ccAll(M, ctxt) = \begin{cases} \{\ \iota,\ \texttt{v}\ \} \ \cup \\ \{\chi(\iota')\downarrow_2(\texttt{f}) \ \mid \ \iota' \in \mathcal{A}ccAll(M, ctxt) \\ \quad \wedge \ \chi(\iota')\downarrow_1 = \texttt{c}' \ \wedge \ \mathcal{F}(M, \texttt{c}', \texttt{f}) = \_\_ \ \} \end{cases}$$
where $ctxt \ = \ ((\iota, \texttt{v}), \chi)$.

$$\mathcal{A}ccPub(M, ctxt) = \begin{cases} \{\ \iota,\ \texttt{v}\ \} \ \cup \\ \{\chi(\iota')\downarrow_2(\texttt{f}) \ \mid \ \iota' \in \mathcal{A}ccPub(M, ctxt) \\ \quad \wedge \ \chi(\iota')\downarrow_1 = \texttt{c}' \ \wedge \ \mathcal{F}(M, \texttt{c}', \texttt{f}) = \_ pa \_ \\ \quad \wedge \ (pa = \texttt{pub} \vee \texttt{c} = \texttt{c}') \ \} \end{cases}$$
where $ctxt \ = \ ((\iota, \texttt{v}), \chi)$ and $\chi(\iota)\downarrow_1 = \texttt{c}$.

Lemma 1. *For all M, $ctxt$, $ctxt'$, $\texttt{e}$, $\texttt{v}$ s.t. $M, ctxt, \texttt{e} \rightsquigarrow ctxt', \texttt{v}$:*

$$\mathcal{A}ccPub(M, ctxt) \subseteq \mathcal{A}ccAll(M, ctxt) \quad \wedge \quad \mathcal{U}sed(M, ctxt, \texttt{e}) \subseteq \mathcal{A}ccAll(M, ctxt)$$

The lemma is not surprising, and the full proof is further work. Note that in general, there is no subset relation between $\mathcal{A}ccPub(M, ctxt)$ and $\mathcal{U}sed(M, ctxt, \texttt{e})$.