Towards Capability Policy Specification and Verification

Sophia Drossopoulou
Imperial College, London
scd@doc.ic.ac.uk

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

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 tend to be tangled with the parts implementing the functionality.

In this paper we argue that the policies followed 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.

To write policy specifications, we propose execution abstractions, which talk about various properties of a program's execution. We use execution abstractions to write the formal specification of five out of the six informal policies in the mint example, famous in the object capability literature. In these specifications, the conclusions but also the premises may relate to the state before as well as after execution, the code may be existentially or universally quantified, and interpretation quantifies over all modules extending the current module.

In the process of writing these specifications, we uncovered several different and plausible alternative meanings for the policies of the mint example, and also discovered some new policies not mentioned in the original papers.

Finally, we demonstrate how we can prove that the example implemented in Java satisfies the capability policies. These proofs make extensive use of the guarantees provided by type system features 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 [8], *e.g.* PDP-1, operating systems *e.g.* CAL-TSS [18], and the CAP computer and operating system [48]. 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 [27] apply capabilities to object-oriented programming. 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, programmers must create an intermediate proxy object which offers only restricted services, delegating them back to 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 [24, 29, 46] and are increasingly used for the provision of security in web programming in industry [30, 41, 47].

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 final fields, private and protected members, or final classes [42, 49]. 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 *restrictive* features.

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 code program implementing them. We also argue 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 an earlier position paper, we anticipated expressing such policies through extensions of temporal logics [11].

In this paper we propose that capability policies can be specified through *execution abstractions*, which are, essentially observations relating to program execution, accessibility, reachability and tracing. For example, execution abstractions can say things like "execution of a given code snippet in a given runtime context will access a certain field", or "it is possible to reach certain code through execution of some initial code". We define a toy Java-like language with some restrictive features, and use it to give precise meaning to execution abstractions.

We follow the Mint example [29] to illustrate our ideas; using these abstractions we give precise specifications to five out of the six policies proposed informally in that paper. In these specifications, the conclusions but also the premises may relate to the state before as well as after execution, the code may be existentially or universally quantified, and interpretation quantifies over all modules extending the current module. In the process of developing the mint specifications, we were surprised by the many different, and plausible interpretations we found for the policies.

We then sketch proofs showing that the Mint code written in JoE/Java adheres to the capability policies. In doing so, we make heavy use of restrictive language features; this was surprising for us, since in traditional program verification, but also in the verification of refinement properties, restrictive features have played only a small role.

Finally, we discuss five further policies, which we discovered in the course of this work, and which we think should have been proposed along with the original six [29].

The contributions of our work are as follows:

- We argue that the specification of capability policies requires concepts that go beyond the features of current specification languages.
- We propose *execution abstractions*, and use them to specify five of the six policies given in the Mint example. We formalise a number of alternative specifications in our language of execution abstractions.
- We give a set of lemmas which support reasoning about adherence to such policies, apply these lemmas informally, and prove adherence to some of the Mint policies.

The rest of the paper is organised as follows: Section 2 presents the Mint [24] as an example of object-capability programming, implemented in Joe-E/Java. Based on that example, Section 3 distills the characteristics of capability policies. Section 4 then outlines executions abstractions, uses them to express those policies, and discusses alternative interpretations. Section 5 explores reasoning about capability policies expressed in our specification language. Section 6 discusses further useful policies not listed in [24], Section 7 surveys related work, and Section 8 concludes.

2. Object-Capability Example

We use as running example a system for electronic money proposed in [29]. 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 [29]:

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, and also no garbage collection of purses — or at least, not of purses with a nonzero balance.

Several different implementations have been proposed for the mint. Fig.1 contains an implementation in Joe-E [24], 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 — 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.

3. Capability Policies

We use the term *capability policy* to describe 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 [32, 39, 48]. A range of object capability policies are discernible from the literature [27–29].

Capability policies generally have the following characteristics:

- They are *program centred*: they talk about properties of programs rather than properties of specifications or protocols.
- They are *fine-grained*: they can talk about *individual objects*, while *coarse-grained* policies only talk about large components such as file servers or the DOM.
- They are *open*. *Open* requirements must be satisfied for any use of the code *extended in any possible manner* — e.g. through dynamic loading, inheritance, subclassing, mashups, mixins, reflection, intercession, or any other extension mechanism supported by the programming language. This is in contrast to *closed* specifications that need only be satisfied for the actual code snippet itself.
- They have *rely* as well as *deny* elements. Rely elements essentially promise that execution of a code snippet in a state satisfying a given pre-condition will reach another state which satisfies some post-condition [14]. Deny elements promise that if execution of a code snippet reaches a certain state, or changes state in a certain way, or accesses some program entity, then the code snippet 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, because:

- They are program centred, since they refer to the actual mint program.
- They are fine grained, as they refer to individual purse and mint objects;
- Even though not explicitly stated there, the policy in [29] is expected to be open; any extension of the code, e.g. by inheritance, 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 the two purses belong to the same mint leads to a state satisfying the post-condition where the money has been transferred.
  - **Pol_2** is a deny requirement; it says that a mint's currency may be changed by some code snippet only if that code snippet makes a function call to the mint object owning the currency. **Pol_3** is another deny requirement; it says that if a mint's currency should change, it increases. **Pol_4** is also a deny requirement, preventing objects that cannot access a purse from modifying the purse's balance.
  - **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 all the code in the system preserves this property.

Deny policies are related to *deny-guarantee* specifications [10] which can forbid particular memory locations from being modified either by the current thread, or by any 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 of a mint), rather than just specific locations.

Deny policies are also related to *correspondence assertions* [13, 50], 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.

```
public final class Mint {       }   //  the Mint capability

public final class Purse {
    private final Mint mint;
    private long balance;

    public Purse(Mint mint, long balance) {
       if (balance<0)
             { throw new IllegalArgtException();  };
       this.mint = mint;
       this.balance = balance;
    }

    public Purse(Purse prs) {
       this.mint = prs.mint;
       this.balance = 0;
    }

    public void deposit(Purse prs, long amnt) {
       if ( mint!=prs.mint || amnt>prs.balance || amnt+balance<0 )
             { throw new IllegalArgtException(); };
       prs.balance -= amnt;
       balance += amnt;   }
    }
}
```

Figure 1. The Mint example, taken from Miller et al., [26].

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 the whole 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 by 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. We start by introducing *execution abstractions*, the concepts necessary to give precise meaning to policies (Sect 4.1), and use these to describe the meaning of the Mint policies (Sect 4.2). Finally we demonstrate how encoding informal policies in our model can explicate a range of different interpretations of those policies (Sect 4.3).

4.1 Execution Abstractions

In this section we introduce *execution abstractions*. We describe their manifestation in a "capability-safe" Java-subset, which we call $\mathcal{C}_\mathtt{j}$. We give a precise definition of $\mathcal{C}_\mathtt{j}$ and all concepts in appendix A, while here we bring out the most salient issues. $\mathcal{C}_\mathtt{j}$ is imperative, supports classes, methods and field, and allows for a simple form of overloading based on the number of the parameters. Thus, except for numbers (which can be encoded), conditionals (which can also be encoded), and exceptions (which can be encoded through stuck executions involving null-dereferencing), $\mathcal{C}_\mathtt{j}$can express all of the Mint example from Fig. 1.

Modules and Linking To model the open nature of capability policies, we need to describe both the program we are checking, and potential extensions of that program (through subclasses, mashups, imports etc). For this we use *modules*, M, to denote programs, and $*$ to describe the combination of two programs into one larger program. Java modules are class definitions and packages: in $\mathcal{C}_\mathtt{j}$, we only model classes. More in App. A.1, where we also model the class definitions from Fig. 1 as modules $\mathsf{M}_{\mathsf{Mint}}$ and $\mathsf{M}_{\mathsf{Purse}}$.

Adherence to policies often relies on the correct use of restrictive features. To model these features, $\mathcal{C}_\mathtt{j}$ supports the method and class annotations private and final. The type rules forbid access to private fields or methods outside their classes, forbid extensions of final classes, forbid redefinitions of final methods in subclasses, and forbid assignment to final fields outside their constructor; see App. A.2.

The $*$ operator links modules together into new modules. Thus, $\mathsf{M}_{\mathsf{Purse}} * \mathsf{M}_{\mathsf{Mint}}$ is a module. Linking performs some compatibility checks, and therefore $*$ is only partially defined. For example, because the field balance is private, $\mathsf{M}_{\mathsf{Purse}} * \mathsf{M}'$ would be undefined, if M' contained the expression x.balance, in a configuration where x was a Purse. In App. A.3 the operation $*$ is only defined if it gives rise to a well-formed module.

Code Modules are not directly executable, but are necessary for the execution of *code snippets*. In $\mathcal{C}_{\mathrm{j}}$ code snippets are expressions, see App. A.1. We use the variables code, code$'$ to range over code snippets.

Runtime Configurations and Expression Evaluation Execution takes place in the context of runtime configurations $\kappa \in RTConf$. A configuration is a stack frame and heap. A stack frame is a tuple consisting of the following four components: the address of the receiver, a mapping giving values to the formal parameters, the class identifier, and the method identifier of the method being executed. A heap is a mapping from object addresses to objects. For example, the configuration $\kappa_1 = ((\iota_3, (\mathsf{prs} \mapsto \iota_2, \mathsf{amount} \mapsto 500\_000\_000), \mathsf{Purse}, \mathsf{deposit}), \chi)$ could be encountered during execution of the method deposit from Purse, while $\kappa_2 = ((\iota_4, (\mathsf{prs} \mapsto \iota_2), \mathsf{Purse}, \mathsf{constr}), \chi)$ could be encountered during execution of the constructor Purse(Purse). More in App. A.4.

Execution of a code snippet code for a module M takes a configuration κ and returns a value v and a new configuration κ'. We describe this through a large step semantics, of the shape M, κ, code $\rightsquigarrow$ κ', v$'$. We define this in App. A.6.

Reached and Arising Snapshots When verifying adherence to policies, it is essential to consider only those snapshots (*i.e.,* configuration and code pairs) which may arise through the execution of the given modules. For example, if we considered *any* well-formed snapshots (well-formed in the sense of the type system), then we would be unable to show that **Pol_5** is obeyed by the mint example. Namely, **Pol_5** guarantees that balances are always positive: configurations where the balance is negative are well-formed, but will never actually arise in the execution of the program.

$\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$, defined in App. A.7, is the set of snapshots corresponding to the start of the execution of the body of any constructor or method called in the process of executing code in the context of M and κ. For example, $(\kappa_2, \mathsf{this.mint := prs.mint; this.balance := 0}) \in \mathcal{R}each(\mathsf{M}, \kappa_1, \mathsf{p1.deposit(p2); p3 = newPurse(p2)})$, where $\kappa_2 = ((\iota_3, (\mathsf{prs} \mapsto \iota_2), \mathsf{Purse}, \mathsf{constr}), \chi)$, and $\iota_2 = \kappa_1 \downarrow_2$ (p2). Note that $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$, corresponds to the *complete* body of a method; *e.g.* $(\kappa_3, \mathsf{this.balance := 0}) \notin \mathcal{R}each(\mathsf{M}, \kappa_4, \mathsf{p3 = newPurse(p2)})$ for any κ_3 and κ_4. Note also, that $\mathcal{A}rising(\mathsf{M})$ is always defined, even though it may be infinite if execution of M, κ_0, code$_0$ does not terminate.

$\mathcal{A}rising(\mathsf{M})$ is the set of snapshots which may be reached during execution of some initial snapshot, κ_0, code$_0$. Similarly to $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$, the function $\mathcal{A}rising(\mathsf{M})$ is always defined. More in App. A.8.

Accessible and Used Objects The principle of least authority requires that a reference to an object does **not** grant permission to access all the capabilities that object holds (i.e. to all the other objects to which that object refers). For example, a configuration having access to a purse object prs cannot necessarily access that purse's mint object, since the field mint is private in Purse. To model this, distinguish between $\mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$ — the set of all objects which are accessible from the frame in κ through *any* path — and $\mathcal{A}cc\mathcal{P}ub(\mathsf{M}, \kappa)$ — the set of all objects accessible through paths which include only public fields, and private fields of objects of the same class as this; see App. A.10.

We use the notation z $:_\kappa$ c to indicate that z is the name of an object which exists in the heap of κ and belongs to class c — with no requirement that there should be a path from the frame to this object; see App. A.4.

We also use the notation $\kappa \in$ c to express that the currently executing method in κ comes from c, and $\kappa \in$ M that the class of the currently executing method is defined in M; see App. A.4.

We define $\mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{code})$ as the set of all addresses used during execution of code in the configuration κ; see App. A.9.

Pure Expressions, Predicates and their Interpretation $\mathcal{C}_{\mathrm{j}}$ provides path expressions p (*i.e.,* i.e. expressions which only involve field reads). We assume, without definition, function identifiers f, predicate identifiers P, with the usual semantics.

For example, mint, prs.mint are paths, $+$ is a function symbol, and $\geq$ is a predicate symbol. Moreover, the function Currency is defined as

$$\mathsf{Currency(mnt)} = \sum_{\mathsf{p} \in Ps(\mathsf{mnt})} \mathsf{p.balance}$$
$$\text{where } Ps(\mathsf{mnt}) = \{\mathsf{p} \mid \mathsf{p} : \mathsf{Purse} \ \land \mathsf{p.mint} = \mathsf{mnt}\}$$

Such terms are interpreted in the context of runtime configurations,

$$\lceil \cdot \rfloor \ : \ Path \longrightarrow RTConf \longrightarrow Value$$
$$\lceil \cdot \rfloor \ : \ Func\_Id \times Var\_Id^* \longrightarrow RTConf \longrightarrow Value$$
$$\lceil \cdot \rfloor \ : \ Pred\_Id \longrightarrow \mathcal{P}\,(Value^*)$$

so that $\lceil \mathsf{p} \rfloor_\kappa = v$ if p is a path and $\emptyset, \kappa, \mathsf{p} \rightsquigarrow \kappa, v$. Function and predicate application have the expected meaning, involving any necessary unfoldings of the definitions, so that $\lceil \mathsf{f(p1, ...pn)} \rfloor_\kappa$=$\lceil fFbody[\mathsf{p1}/x1, ...\mathsf{pn}/xn] \rfloor_\kappa$, where $fBody$ is the function definition of f, with free variables $x1$,... xn. Finally, $\lceil P(p1, ...pn) \rfloor_\kappa = \lceil P \rfloor (\lceil p1 \rfloor_\kappa, ...\lceil pn \rfloor_\kappa)$.

4.2 Interpretations of the Mint policies

Armed with the concepts defined in Sect. 4.1, we turn our attention to the precise meaning of the first five policies from the Mint example. (We do not address the sixth policy as our formalisation does not yet incorporate trust). An important aspect of our approach is that we quantify over modules, extensions to modules, over the code being executed, and over configurations.

We discuss the policies in order of increasing complexity of their specification, rather than in numerical order. In Sect. 4.3 we shall discuss some possible alternative interpretations of these policies, and in Sect. 6 we consider some additional policies.

The fifth policy **Pol_5**, "Balances are always non-negative integers", is akin to a class invariant [25, 34, 43]. We can express the policy directly by requiring that a module M satisfies **Pol_5**, if for all M' legal extensions of M, and snapshots $(\kappa, \_)$ arising through execution of the augmented program $\mathsf{M} * \mathsf{M}'$, the balance is positive in κ.

Module M satisfies policy **Pol_5**
iff
$$\forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{prs} :_\kappa \mathsf{Purse}.$$
$$\lceil \mathsf{prs.balance} \rfloor_\kappa \geq 0$$

Note that the arising snapshots are considered in the context of the *extended* module $\mathsf{M} * \mathsf{M}'$, where M' is universally quantified. This reflects the open nature of capability policies, and allows calling methods and accessing fields defined in M but also in M' before reaching the snapshot (κ, code).

Note also, that $\mathcal{A}rising(\mathsf{M} * \mathsf{M}')$ catches snapshots at the *beginning* of a method execution. Therefore, if a method were to temporarily set balance to a negative value, but restored it to a positive value before returning, would *not* violate **Pol_5**.

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 we shall see later on, the second interpretation is a corollary of **Pol_2**; here we analyse the first interpretation:

Module M satisfies policy **Pol_3**
iff
$$\forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_\kappa \mathsf{Mint}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\Longrightarrow$$
$$\lceil \mathsf{Currency(mnt)} \rfloor_\kappa \leq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$$

Namely, we require that for any arising snapshot (κ, code), and any execution originating from (κ, code) and leading to a new configuration κ', the Currency at the old configuration is less than or equal to the currency at the new configuration. Therefore, in the conclusion we talk about the values of functions in the old configuration (*i.e.* $\lceil \mathsf{Currency(mnt)} \rfloor_\kappa$) as well as those in the new configuration (*i.e.* $\lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$). Conclusions which are in terms of the old as well as the new state are common in standard approaches to program specification.

Pol_3 describes a monotonic property, and is therefore related to history invariants [19]. **Pol_3** differs from history invariants through its open nature, given by the quantification over M'.

The first policy. **Pol_1** states "With two purses of the same mint, one can transfer money between them". We can understand **Pol_1** to mean that if p1 and p2 are purses of the same mint, then the method call $\mathsf{p1.deposit(p2, m)}$ will transfer the money. In section 4.3, we shall present two other possible meanings for this policy. We write the first interpretation of **Pol_1** as:

Module M satisfies policy **Pol_1A**
iff
$$\forall\, \mathsf{M}'.\forall\ (\kappa, \mathsf{p1.deposit(p2, m)}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').$$
$$\forall \mathsf{p1}, \mathsf{p2} :_\kappa\ \mathsf{Purse}.$$
$$\lceil \mathsf{p1.mint} \rfloor_\kappa = \lceil \mathsf{p2.mint} \rfloor_\kappa \quad \wedge \quad \lceil \mathsf{p2.balance} \rfloor_\kappa \geq \mathsf{m}$$
$$\wedge \quad \mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2, m)} \rightsquigarrow \kappa', \mathsf{v}$$
$$\Longrightarrow$$
$$\lceil \mathsf{p1.balance} \rfloor_{\kappa'} = \lceil \mathsf{p1.balance} \rfloor_\kappa + \mathsf{m}$$
$$\wedge \quad \lceil \mathsf{p2.balance} \rfloor_{\kappa'} = \lceil \mathsf{p2.balance} \rfloor_\kappa - \mathsf{m}.$$

The specification **Pol_1A** again ranges over all module extensions, M', and thus covers a larger set of runtime configurations than a Hoare triple,[1] or if we elided the quantification over M'. This policy requires that the code M' can do nothing to break the behaviour of the deposit method from M, thus either requiring the use of restrictive features (*e.g.* forcing the method deposit to be final, or the class Purse to be final, or package confined), or the use of contracts, where subclasses are implicitly expected to satisfy the superclass's contract,

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.

[1] We could specify this in a Hoare triple as follows:

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

Module M satisfies policy **Pol_4**
iff
$$\forall\, \mathsf{M}',\, (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{prs} :_\kappa \mathsf{Purse}.$$
$$\mathsf{M} * \mathsf{M}', \kappa \leadsto \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{prs.balance} \rfloor_\kappa \neq \lceil \mathsf{prs.balance} \rfloor_{\kappa'}$$
$$\Longrightarrow$$
$$\lceil \mathsf{prs} \rfloor_\kappa \in \mathcal{U}sed(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code})$$

Note that in contrast to the previous policies, and in contrast to the standard approach to program specification, the premise of the policy is in terms of both the old configuration (here $\lceil \mathsf{prs.balance} \rfloor_\kappa$ and the new configuration (here $\lceil \mathsf{prs.balance} \rfloor_{\kappa'}$).

The second policy. **Pol_2**, stating "Only someone with the mint of a given currency can violate conservation of that currency.", is similar to **Pol_4**, in that it mandates that a change (here a change in the currency) may only happen if the originating configuration had access to an entity (here access to the mint).

Module M satisfies policy **Pol_2A**
iff
$$\forall\, \mathsf{M}',\, (\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_\kappa \mathsf{Mint}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \leadsto \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{Currency(mnt)} \rfloor_\kappa \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$$
$$\Longrightarrow$$
$$\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{U}sed(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code})$$

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

- They refer to a fixed module M, and all its legal extensions M'.
- They specify that execution of some code, under some conditions, guarantees some conclusions.
- Both conditions *and* conclusions may refer to properties of the state *before* as well as *after* execution.
- The code may be universally or existentially quantified, or explicitly given.

4.3 Alternative interpretations of the Mint Policies

Because our policy descriptions have precise semantics — unlike the informal English policies from the original Mint example — a single English policy can have a number of plausible interpretations in our notation. We explore some of these alternatives here; we were surprised how many different interpretations we uncovered while analysing this example.

The first policy revisited. **Pol_1** states "With two purses of the same mint, one can transfer money between them". In section 4.2 we proposed as possible meaning that the call $\mathsf{p1.deposit(p2)}$ will transfer the money.

This is perhaps an over-specification, as it prescribes *how* the transfer is to take place — by calling the $\mathsf{p1.deposit(p2)}$ method. Alternatively, we may want to only require that it is *possible* for the transfer to take place, without constraining the program design. We can define a second, more general version of the policy, which only requires the existence of a code snippet that performs the transaction, provided that purses p1 and p2 share the same mint, that p2 has sufficient funds, and that they are both accessible in κ without reading private fields ($\mathcal{A}ccPub(\mathsf{M}, \kappa)$).

Module M satisfies policy **Pol_1B**
iff
$$\forall (\kappa, \_) \in \mathcal{A}rising(\mathsf{M}).\ \forall \mathsf{p1}, \mathsf{p2} :_\kappa \mathsf{Purse}.$$
$$\lceil \mathsf{p1.mint} \rfloor_\kappa = \lceil \mathsf{p2.mint} \rfloor_\kappa\ \wedge\ \lceil \mathsf{p2.balance} \rfloor_\kappa \geq \mathsf{m}$$
$$\wedge\ \lceil \mathsf{p1} \rfloor_\kappa, \lceil \mathsf{p2} \rfloor_\kappa \in \mathcal{A}ccPub(\mathsf{M}, \kappa)$$
$$\Longrightarrow$$
$$\exists\, \mathsf{code}.\ \forall \mathsf{M}'.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \leadsto \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{p1.balance} \rfloor_{\kappa'} = \lceil \mathsf{p1.balance} \rfloor_\kappa + \mathsf{m}$$
$$\wedge\ \lceil \mathsf{p2.balance} \rfloor_{\kappa'} = \lceil \mathsf{p2.balance} \rfloor_\kappa - \mathsf{m}.$$

Note that this policy requires that execution of the code has the required properties for *all* extending modules M'.

Another possible meaning of **Pol_1** is that deposit can be called successfully only if the two purses belonged to the same mint:

Module M satisfies policy **Pol_1C**
iff
$$\forall \mathsf{M}'.\ \forall (\kappa, \mathsf{p1.deposit(p2)}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2, m)} \leadsto \kappa', \mathsf{v}$$
$$\Longrightarrow$$
$$\lceil \mathsf{p1.mint} \rfloor_\kappa = \lceil \mathsf{p2.mint} \rfloor_\kappa$$

The requirement $\mathsf{M} * \mathsf{M}', \kappa, \mathsf{p1.deposit(p2, m)} \leadsto \kappa', \mathsf{v}$ is crucial in the premise, in that it ensures that execution does not lead to an error (our current definition of the language $\mathcal{C}_\mathsf{j}$ does not support exceptions). Note, also, that in this specification the conclusion is only concerned with properties observable in the original configuration, κ, while the premise is concerned with properties observable in κ as well as κ'. This reflects the deny nature of the policy.

Finally, a fourth, 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_1D**
iff
$$\forall \mathsf{M}'.\,\forall(\kappa,\mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{p1} :_\kappa \mathsf{Purse}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{p1.balance} \rfloor_\kappa \neq \lceil \mathsf{p1.balance} \rfloor_{\kappa'}$$
$$\Longrightarrow$$
$$\exists \kappa',\ \text{s.t.}$$
$$(\kappa', \_) \in \mathcal{R}each(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code})$$
$$\wedge\quad \kappa' = (\_, \_, \mathsf{Purse}, \mathsf{deposit})$$
$$\wedge\quad (\lceil \mathsf{this} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_\kappa \vee \lceil \mathsf{prs} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_\kappa)$$

The assertion $(\kappa', \_) \in \mathcal{R}each(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \wedge \kappa' = (\_, \_, \mathsf{Purse}, \mathsf{deposit})$ guarantees that execution of the snapshot (κ, code) will reach a point where it calls the method deposit from Purse. – see App. A.7. The assertion $(\lceil \mathsf{this} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_\kappa \vee \lceil \mathsf{prs} \rfloor_{\kappa'} = \lceil \mathsf{p1} \rfloor_\kappa)$ guarantees that the receiver or the first argument of that method call will be $\lceil \mathsf{p1} \rfloor_{\kappa}$.

The second policy revisited. **Pol_2**, "Only someone with the mint of a given currency can violate conservation of that currency." mandates that a change in the currency may only happen if the originating configuration had access to the mint. In section 4.2 we took "access to" to mean that the code executed eventually would read the mint object (i.e. that the mint was in the set $\mathcal{U}sed$). We see three alternative interpretations for the meaning of *having access to*:

1. $\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{code})$, i.e. that execution of code in the context of κ will at some point use the object mnt.
2. $\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$, i.e. that κ has a path from the stack frame to mnt which involves any fields.
3. $\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{A}cc\mathcal{P}ub(\kappa, \mathsf{code})$, i.e. that κ has a path from the stack frame to mnt which involves only public fields, or private fields from the same class as the current receiver.

This means there are two further ways in which **Pol_2** may be understood:

Module M satisfies policy **Pol_2B**
iff
$$\forall\ \mathsf{M}'.\,\forall(\kappa,\mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_\kappa \mathsf{Mint}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{Currency(mnt)} \rfloor_\kappa \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$$
$$\Longrightarrow$$
$$\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M} * \mathsf{M}', \kappa)$$

Module M satisfies policy **Pol_2C**
iff
$$\forall\ \mathsf{M}'.\,\forall(\kappa,\mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_\kappa \mathsf{Mint}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\wedge\ \lceil \mathsf{Currency(mnt)} \rfloor_\kappa \neq \lceil \mathsf{Currency}_{\kappa'}\mathsf{(mnt)} \rfloor_{\kappa}$$
$$\Longrightarrow$$
$$\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{A}cc\mathcal{P}ub(\mathsf{M} * \mathsf{M}', \kappa)$$

Our interpretation of **Pol_2** in section 4.2 uses the first choice. In section 5 we prove adherence to **Pol_2A**. Moreover, lemma 1 from App. A.10 guarantees that $\mathsf{mnt} :_\kappa \mathsf{Mint}$ and $\lceil mnt \rfloor_\kappa \in \mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{code})$ imply that $\lceil mnt \rfloor_\kappa \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$, therefore this also means that the code also adheres to **Pol_2B**.

What about **Pol_2C**? It gives a stronger guarantee than **Pol_2B** (this also follows from lemma 1 from App. A.10), and therefore is to be preferred over **Pol_2B**. However, $M_{\mathsf{Purse}} * M_{\mathsf{Mint}}$ does *not* satisfy **Pol_2C**. 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_2C**. 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 configuration 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 configuration did not have public access to x.myMint. This is discussed further in section 6.3.

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. In this section we show how to reason that a program conforms to the policies. The arguments have the 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 certain properties — such as the possession of particular capabilities. The use of restrictive language features is crucial in reasoning about the Mint program, because objects need restrictive features to protect the capabilities they hold against external agents seeking to undermine the system.

We first outline the guarantees provided by restrictive language features (section 5.1). Armed with these guarantees, we then sketch the verification of **Pol_1A** and **Pol_1B** (section 5.2), **Pol_4** (section 5.3), and **Pol_5** (section 5.4). We then and revisit the concept of *framing* (section 5.5) and use it to verify **Pol_2A** (section 5.6). We do not show the proof of **Pol_3**, as it uses a combination of the arguments for the proof of **Pol_2A** and **Pol_1A**.

5.1 The guarantees of restrictive language features

The restrictive features in programming languages were proposed in order to restrict possible program behaviours. In particular, for well-formed programs, the private and final annotations make the following guarantees:

G1 If an object, which exists in snapshot (κ, code), has a final field whose value changes during execution of (κ, code), then the method called in κ was a constructor for that object.

G2 If class c is final and belongs to module M, and c contains method m whose body does not contain further method calls, and if $\mathsf{M}, \kappa, \mathsf{x.m(y)} \rightsquigarrow \kappa', v$, and the call in κ was the method m from c, then for all further modules M′, execution behaves the same, *i.e.* : $\mathsf{M} * \mathsf{M}', \kappa, \mathsf{x.m(y)} \rightsquigarrow \kappa', v$.

G3 If class c has a private field, and an object of that class exists in an arising snapshot (κ, code), then there exists a sequence of method invocations reached from the execution of (κ, code), which uniquely determine the value of the field in κ. Moreover all these methods were defined in class c.

Since the linking operator $*$ is undefined if it does not lead to a well-formed program, and since the interpretation of all policies involve liking an external module (M′) to the current one, we implicitly expect programs to be well-formed, and thus guarantee **G1-G3**.

Moreover, our programming language $\mathcal{C}_{\mathsf{j}}$ makes further guarantees over program executions of well-formed

G4 If an object exists in κ' but not in κ, and if snapshot (κ, code) results in κ', then execution of (κ, code) reaches the call of a constructor of the class of the object.

G5 If an object exists in κ, and if (κ, code) results in κ', and a field value changes between κ and κ' then execution of (κ, code) uses that object.

5.2 Verification sketch for Pol_1

We first prove that if p1 and p2 share mints, then, calling
p1.deposit(p2, m)
transfers m from p2 to p1 (for appropriate m and balances). The latter can be established using a Hoare Logic extension which can reason about object oriented programs [33, 40].

We can establish that the property holds for all extensions M′, and thus adherence to **Pol_1A**, by application of the result above, by inspection of the method body for deposit, and by **G2**. We can then establish adherence to **Pol_1B**, by application of the result from above, and by replacing the existentially quantified code by p1.deposit(p2, m).

5.3 Verification sketch for Pol_4

Proof of **Pol_4** follows straightforwardly from **G_5**. The proof is so easy because balance is represented through a field; it would have been more complex to establish if looking up the balance involved a function call, and if the balance was not directly stored in the purse, as we do for instance in the alternative implementation of the Mint, given in Figure 11 in the appendix and discussed in section 6.2 below.

5.4 Verification sketch for Pol_5

With an object-oriented Hoare logic we can establish that the two constructors of Purse create objects with positive balance, and that the deposit method preserves balance as positive for both objects. Moreover, balance is private, and the class Purse is final. With the above result, and applying **G2** and **G3**, we obtain adherence to **Pol_5**.

5.5 Framing

Policies **Pol_3** and **Pol_2** talk about the the value of Currency, which, in the terminology of section 3, is a pervasive entity. To deal with that, we will employ the concept of framing known, *e.g.* , from separation logic [15, 37].

For a term t, a configuration κ, and S a set of pairs of address and field identifiers, we say that $\kappa \models t\, \mathbf{frm}\, S$, if the value of the fields in S determines the value of the term t. In other words, for two configurations κ and κ' and set S such that $\kappa \models t\, \mathbf{frm}\, S$, and $\kappa' \models t\, \mathbf{frm}\, S$, if $\lceil \iota.\mathsf{f} \rfloor_\kappa = \lceil \iota.\mathsf{f} \rfloor_{\kappa'}$ for all $(\iota, \mathsf{f}) \in S$, we have that $\lceil t \rfloor_\kappa = \lceil t \rfloor_{\kappa'}$:
This gives rise to the following framing property:

F1 For term t and configurations κ and κ', sets S and S', and $\kappa \models t\, \mathbf{frm}\, S$, and $\kappa' \models t\, \mathbf{frm}\, S'$ if $\lceil t \rfloor_\kappa \neq \lceil t \rfloor_{\kappa'}$, then $S \neq S'$, or $\exists (\iota, \mathsf{f}) \in S$, $s.t.$ $\lceil \iota.f \rfloor_\kappa \neq \lceil \iota.f \rfloor_{\kappa'}$.

Moreover, with $\mathsf{M} \models t\, \mathbf{frm}\, \mathsf{M}'$ we express that the value of t depends only on objects of classes defined in module M. In more detail, $\mathsf{M} \models t\, \mathbf{frm}\, \mathsf{M}'$, if for any two configurations κ, κ', if the frames coincide (*i.e.* $\kappa \downarrow_1 = \kappa' \downarrow_1$) and the set of objects of classes from M′ coincide (*i.e.* for all $\mathsf{c} \in dom(\mathsf{M}')$ we have that $\{\iota \,|\, \kappa \downarrow_2 (\iota) \downarrow_1 = \mathsf{c}\}$ = $\{\iota \,|\, \kappa' \downarrow_2 (\iota) \downarrow_1 = \mathsf{c}\}$), and have identical contents, (*i.e.* for all ι, if $\kappa \downarrow_2 (\iota) \downarrow_1 = \mathsf{c}$ then $\kappa \downarrow_2 (\iota) = \kappa' \downarrow_2 (\iota)$), then $\lceil t \rfloor_\kappa = \lceil t \rfloor_{\kappa'}$.

We obtain that if $\mathsf{M} \models t\, \mathbf{frm}\, \mathsf{M}'$, and M′ contains private fields only, then the value of t can be modified only through execution of methods from M′:

F2 For a term t and module M such that $\mathsf{M} \models t\, \mathbf{frm}\, \mathsf{M}'$, and any snapshot $(\kappa_1, \mathsf{code}_1)$ arising from an initial configuration whose execution leads to (κ_2, v), if $\lceil t \rfloor_{\kappa_1} \neq \lceil t \rfloor_{\kappa_2}$, then execution of $(\kappa_1, \mathsf{code}_1)$ reaches a snapshot (κ, code) such that code is the body of some method from M′, and execution of (κ, code) evaluates to (κ', v') and $\lceil t \rfloor_\kappa \neq \lceil t \rfloor_{\kappa'}$.

5.6 Verification sketch for Pol_2

The verification of adherence to **Pol_2** employs the concept of framing, as well as the guarantees of restrictive language features from section 5.1.
Remember the function Currency, given in section 4.1:

$$\mathsf{Currency(mnt)} = \sum_{\mathsf{p} \in Ps(\mathsf{mnt})} \mathsf{p.balance}$$

where $Ps(\mathsf{mnt}) = \{\mathsf{p} \mid \mathsf{p} : \mathsf{Purse}\ \wedge \mathsf{p.mint} = \mathsf{mnt}\}$

We can establish that:
(∗) $\kappa \models \mathsf{Currency(x)}\ \mathbf{frm}\ \{\ (\iota, \mathsf{balance}) \mid \iota \in \lceil Ps(\mathsf{x}) \rfloor_\kappa\ \}$.
(∗∗) $\forall \mathsf{M}'.\ \mathsf{M}_{\mathsf{Purse}} * \mathsf{M}' \models \mathsf{Currency(x)}\ \mathbf{frm}\ \mathsf{M}_{\mathsf{Purse}}$

In order to prove **Pol_2A** we shall prove the following auxiliary lemma, which states that if the currency of a mint was modified, then this must have involved calling the constructor Purse(Mint, long).

Aux Lemma For any M'
(1) $\mathsf{M}_{Purse} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', v$,
(2) $\lceil \mathsf{Currency(mnt)} \rfloor_\kappa \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$,
(3) κ is a call of a method or constructor from Purse,
then:
(4) κ is a call of Purse(Mint, long).

Proof of Aux Lemma From (3), and by inspection $\mathsf{M}_{\mathsf{Purse}}$, we obtain that there exist three cases:

1st Case: κ is a call of Purse(Purse):
By Hoare logic on the body of Purse(Purse), we obtain, $\forall \mathsf{mt}.\ \lceil \mathsf{Currency(mt)} \rfloor_\kappa = \lceil \mathsf{Currency(mt)} \rfloor_{\kappa'}$. This is a contradiction with (2).

2nd Case: κ is a call of Purse(Mint, long):
done!

3rd Case: κ is a call of deposit(Purse, long):
By Hoare logic on the body of deposit(Purse, long) we obtain, $\forall \mathsf{mt}.\ \lceil \mathsf{Currency(mt)} \rfloor_\kappa = \lceil \mathsf{Currency(mt)} \rfloor_{\kappa'}$. This is a contradiction with (2).

We can now prove adherence to **Pol_2A**.
Lemma: $\mathsf{M}_{\mathsf{Purse}}$ *satisfies* **Pol_2A**
For any M', if
(1) $\mathsf{M}_{Purse} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', v$,
(2) $\lceil \mathsf{Currency(mnt)} \rfloor_\kappa \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'}$,
then:
(3) $\lceil \mathsf{mnt} \rfloor_\kappa \in \mathcal{U}sed(\mathsf{M}_{Purse} * \mathsf{M}', \kappa, \mathsf{code})$

Proof From (∗∗), and by **F2**, we obtain that $\exists \mathsf{code}'.\exists \kappa'', \kappa'''$.
(1) $(\kappa'', \mathsf{code}') \in \mathcal{R}each(\mathsf{M}_{Purse} * \mathsf{M}', \kappa, \mathsf{code})$
(2) $\mathsf{M}_{Purse} * \mathsf{M}', \kappa'', \mathsf{code}' \rightsquigarrow \kappa''', v$,
(3) $\lceil \mathsf{Currency(mnt)} \rfloor_{\kappa''} \neq \lceil \mathsf{Currency(mnt)} \rfloor_{\kappa'''}$,
(4) κ'' is a call of a method or constructor from Purse.
From (2)-(4) and **Aux Lemma** we obtain that κ'' was the call to constructor Purse(Mint, long). Therefore, the argument to that call is mnt, which gives (3). Done.

Characteristics of the proofs Adherence to the policies is demonstrated through a combination of proof steps similar to Hoare-logic-style reasoning, with steps based on framing, and steps leveraging the guarantees given by private and final annotations. We claim that the ingredients we used in the proof steps shown above correspond to the concepts programmers use when thinking about their code.

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. We discuss the need for these policies, and formalize them in our language.

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 previously been returned out of the mint-and-purse system. These policies are trivially satisfied by the code in Fig. 1: 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 object” policies may appear trivial because of the simplicity and relatively small size of the mint and purse system, but even an apparently trivial change to the implementation of the system can break these policies. For example, adding a public `getMint` accessor that returns a purses’ mint (useful, perhaps, for clients with purses from multiple mints) would break **Pol_7B** and destroy the integrity of the whole system. Given a purse, an attacker could call `getMint` to get a mint, and then call `Purse(mint, amount)` to coin any amount of money.

We now write out **Pol_7a** in our policy language:

Module M satisfies policy **Pol_7A**
iff

$$\forall\ \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{mnt} :_{\kappa'} \mathsf{Mint}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\wedge\ \kappa \in \mathsf{M}'\ \wedge\ \lceil \mathsf{mnt} \rfloor_{\kappa'} \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M} * \mathsf{M}', \kappa')$$
$$\Longrightarrow$$
$$\lceil \mathsf{mnt} \rfloor_{\kappa'} \notin dom(\kappa \downarrow_2)\ \vee\ \lceil \mathsf{mnt} \rfloor_{\kappa'} \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M} * \mathsf{M}', \kappa)$$

In other words, if execution of code external to M (expressed through $\kappa\in\mathsf{M}'$) obtains access to a Mint object (expressed through $\lceil\mathsf{mnt}\rfloor_\kappa \in \mathcal{AccAll}(\mathsf{M} * \mathsf{M}', \kappa')$and $\mathsf{mnt} :_{\kappa'} \mathsf{Mint}$), then that object is fresh (expressed through $\lceil\mathsf{mnt}\rfloor_{\kappa'} \notin dom(\kappa \downarrow_2)$), or κ already had access to that object (expressed through $\lceil\mathsf{mnt}\rfloor_{\kappa'} \in \mathcal{AccAll}(\mathsf{M} * \mathsf{M}', \kappa)$).

The specification of **Pol_7B** is similar:

Module M satisfies policy **Pol_7B**
iff

$$\forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{Arising}(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{prs} :_{\kappa'} \mathsf{Purse}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v}$$
$$\wedge\ \kappa \in \mathsf{M}' \ \wedge\ \lceil\mathsf{prs}\rfloor_{\kappa'} \in \mathcal{AccAll}(\mathsf{M} * \mathsf{M}', \kappa')$$
$$\Longrightarrow$$
$$\lceil\mathsf{prs}\rfloor_{\kappa'} \notin dom(\kappa \downarrow_2) \ \vee\ \lceil\mathsf{prs}\rfloor_{\kappa'} \in \mathcal{AccAll}(\mathsf{M} * \mathsf{M}', \kappa)$$

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 — perhaps modelling an implementation of the Mint system on top of a relational database (see Fig. 11 in App. A.11). This design is in some sense the complement of Fig. 1: where that design has a `Mint` class that is a pure capability ("`class Mint(){ }`", with no state or behaviour), here the Purse objects are pure capabilites and all the application's state (the database, that is, the map from purses to their balances) and behaviour (the `deposit` method) are within the Mint class. The risk this offers is that the original six policies do not mention any internal database object (neither do 7A & B) so an accessor that returned the map capability would not breach any policy, even though a client with the map 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, so there can be no representation exposure. Policies such as **Pol_8** ensure that the system will remain secure even if the implementation is changed to require subsidiary objects.

To express the intention of **Pol_8** we could require that any changes to the balance of a Purse were due to some method being executed by the class Purse itself.

Module M satisfies policy **Pol_8**
iff

$$\forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{Arising}(\mathsf{M} * \mathsf{M}').\ \forall \mathsf{prs} :_{\kappa} \mathsf{Purse}.$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v} \ \wedge\ \kappa \notin \mathsf{M}$$
$$\wedge\ \lceil\mathsf{prs.balance}\rfloor_\kappa \neq \lceil\mathsf{prs.balance}\rfloor_{\kappa'}$$
$$\Longrightarrow$$
$$\exists \kappa'', \kappa'''.\exists \mathsf{v}'.\exists \mathsf{code}'.$$
$$(\kappa'', \mathsf{code}') \in \mathcal{Reach}(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \ \wedge\ \kappa'' \in \mathsf{Purse}$$
$$\wedge\ \mathsf{M} * \mathsf{M}', \kappa'', \mathsf{code}' \rightsquigarrow \kappa''', \mathsf{v}'$$
$$\wedge\ \lceil\mathsf{prs.balance}\rfloor_\kappa = \lceil\mathsf{prs.balance}\rfloor_{\kappa''}$$
$$\wedge\ \lceil\mathsf{prs.balance}\rfloor_{\kappa''} \neq \lceil\mathsf{prs.balance}\rfloor_{\kappa'''}$$

The module consisting of class HashMap and of class Purse as given in figure 11 satisfies **Pol_8**. This formulation of the policy has the disadvantage that it forces class HashMap into the same module as Mint, and thus precludes any further use of that class in the remaining program M'. Note that if the field database was declared public, then the two classes would not satisfy **Pol_8**, even though they would satisfy **Pol_2A**.

6.3 Supersidiary Capabilities

The third issue we consider relates to supersidiary capabilities, that is, when a Mint or a Purse is combined into a larger structure. In a crucial sense, the Fig. 1 design already involves supersidiary capabilities: each Purse holds the capability to its mint.

The problem this raises can be seen 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 Purse 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 objects are effectively subsidiary to the module containing all the declarations in the system.

We can specify **Pol_9** as follows:

Module M satisfies policy **Pol_9**
iff

$$\forall\, \mathsf{M}'.\ \forall \mathsf{prs} :_\kappa \mathsf{Purse}.$$
$$\forall(\kappa, \mathsf{code}) \in \mathcal{Arising}(\mathsf{M} * \mathsf{M}').$$
$$\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \rightsquigarrow \kappa', \mathsf{v} \ \wedge\ \kappa \in \mathsf{Purse}$$
$$\Longrightarrow$$
$$\lceil\mathsf{Currency(prs.mint)}\rfloor_\kappa = \lceil\mathsf{Currency(prs.mint)}\rfloor_{\kappa'}$$

In other words, execution of some code originating in class Purse (expressed through $\kappa \in \mathsf{Purse}$) does not affect the currency the purse's mint.

The problem of subsidiary capabilities is much harder to address when a mint, say, is part of some supersidiary system — a `CentralBank` object perhaps:

```
class CentralBank {
  private final Mint myMint = new Mint();
  private final Purse myTreasury = new Purse();
  public void inflate() {
    Purse tmpPurse = new Purse(myMint,1000000000)
    myTreasury.deposit(tmpPurse,1000000000)
  }
```

A `CentralBank` 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 — perhaps this method should have been called `quantitativeEasing`. Now consider a supersidiary client of a `CentralBank` object — the finance minister say. The finance minister does *not* have a reference to the mint (because the central bank is supposed to be independent!) so by **Pol_2** she should not be able to inflate the currency. If, however, the finance minister calls `myCentralBank.inflate` then the currency will be inflated all the same. (This is a 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). Arguably the solution here is simple: if the central bank is supposed to be independent, only the governor of the bank should have a capability for the `CentralBank` object, and particularly not the finance minister!

Pol_10 Only the govenor of the central bank can have a capability to the `CentralBank` object.

We can formalise that policy by requiring that any change to the currency of the central bank was due to a method executed by the governor of the central bank, *i.e.* where cb.governor is the receiver.

$$
\begin{array}{c}
\mathsf{Module\ M\ \ satisfies\ policy\ } \textbf{Pol\_10} \\
\text{iff} \\
\forall\, \mathsf{M}'.\ \forall(\kappa, \mathsf{code}) \in \mathcal{A}rising(\mathsf{M} * \mathsf{M}'). \\
\forall \mathsf{cb} :_{\kappa} \mathsf{CentralBank}. \\
\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code} \ \rightsquigarrow\ \kappa', \mathsf{v} \\
\wedge\ \lceil \mathsf{Currency(cb)} \rfloor_{\kappa} \neq \lceil \mathsf{Currency(cb)} \rfloor_{\kappa'} \\
\Longrightarrow \\
\exists \kappa'', \kappa'''.\exists \mathsf{v}'.\exists \mathsf{code}. \\
(\kappa'', \mathsf{code}') \in \mathcal{R}each(\mathsf{M} * \mathsf{M}', \kappa, \mathsf{code}) \\
\wedge\ \ (\kappa'' \downarrow_1) \downarrow_1 = \lceil \mathsf{cb.governor} \rfloor_{\kappa} \\
\mathsf{M} * \mathsf{M}', \kappa'', \mathsf{code} \ \rightsquigarrow\ \kappa''', \mathsf{v}' \\
\wedge\ \lceil \mathsf{Currency(cb)} \rfloor_{\kappa} = \lceil \mathsf{Currency(cb)} \rfloor_{\kappa''} \\
\wedge\ \lceil \mathsf{Currency(cb)} \rfloor_{\kappa''} \neq \lceil \mathsf{Currency(cb)} \rfloor_{\kappa'''}
\end{array}
$$

7. Related Work

Object-capabilities were first introduced [27] seven years ago, and many recent studies manage or verify safety or correctness of object-capability programs.

Google's Caja [30] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* capabilities. Sandboxing has been validated formally: Maffeis et al. [21] 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. [45] model JavaScript APIs in Datalog, and then carry out a Datalog 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 [31] 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 [17], including pointer analyses. Bhargavan et al. [4] extend language-based sandboxing techniques to support "defensive" components that can execute successfully in otherwise untrusted environments. Meredith et al. [23] encode policies as types in higher order reflective π-calculus.. Politz et al. [35] 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"* [20]. Dimoulas et al. [9] generalise the language and typechecker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities.

The WebSand [5, 22] and Jeeves [52] projects use dynamic techniques to monitor safe execution of policies. Richards et al. [38] 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* techniques 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. [12] developed a formalisation of JavaScript based on separation logic and verified examples. Xiong and Qin et al. [36, 51] worked on similar lines. Swamy et al. [44] recently developed a mechanised verification technique for JavaScript based on the Dijkstra Monad in the F* programming language. Finally, Jang et al. [16] 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 this paper, we have advocated that capability policies are necessary for reasoning about programs using object-capability security. We have argued that capability policies are 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 execution abstractions, and developed a capability specification style, which incorporates universal and existential quantification over program code, explicit naming of snapshots before, after and during execution, and their use in premises and in conclusions. We have used our approach to specify most of the Mint example.

We have shown how efforts at specifying policies precisely can uncover ambiguities in policies' interpretations, and can help find additional implicit policies that can be made explicit. We have proposed another five policies for the Mint, and formulated then in our language.

Finally, we have sketched how we can prove that programs adhere to capability policies. The arguments we have used include, but are not restricted to Hoare Logic and the type-inference steps. We have argued in terms of the objects accessible from a code snippet, the set of methods that a code snippet can call, and considered how restrictive language features restrict the possible execution scenaria.

We believe that our work so far, and in particular the fact that we were able to express so many policies, to characterize the difference between alternative interpretations, and uncover new policies, has demonstrated that the approach is a valid proof of concept. More generally, we believe that an approach based on the way we express policies and on the proof steps from section 5, can help reflect some of the thought process during program development and program understanding.

Of course, being at the proof-of-concept stage, several aspects need developed. In further work, we want to refine the execution abstractions, to develop a programmer-friendly notation for specifications, to consider the specification of the further policies we uncovered as well as other policies from the literature, and extend our toy language to encompass further salient programming language features.

We also want to develop a formal logic to support reasoning about code's adherence to capability policies.

Finally, inspired by the original Mint work, we want to consider the specification and verification of capabilities in other programming languages, in particular, languages without static types.

Acknowledgments

We are deeply grateful to Greg Meredith, Mark Miller, and Mike Stay for many fruitful debates on Google Groups, when we were trying to determine what a capability policy really is, who the the stakeholders are, and how they should be quantified over. The focus on reachable snapshots only was pointed out by Mark Miller at these meeting.

We also thank Gareth Smith, Sylvan Clebsch, Robert O'Callaghan, Neal Glew, Sergio Maffeis, Lawrence Tratt, Marco Servetto, Gavin Bierman, Chris Hawlblitzel, Manuel Fahndrich, Kim Bruce, and many anonymous reviewers for discussions on this material; their questions, answers, and challenges helped us clarify our ideas.

This work is partly supported by the Royal Society of New Zealand Marsden Fund.

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] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *OOPSLA*, 2002.

[8] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.

[9] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. 2013.

[10] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*. Springer, 2009.

[11] Sophia Drossopoulou and James Noble. The need for capability policies. In *(FTfJP)*, 2013.

[12] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, 2012.

[13] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *MFPS*. Elsevier, ENTCS, 2001.

[14] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[15] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM Press, 2001.

[16] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, 2012.

[17] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-Chieh Shan. An Analysis of the Mozilla Jetpack Extension FrameworK. In *ECOOP*, Springer, 2012.

[18] Butler W. Lampson and Howard E. Sturgis. Reflection on an Operating System Design. *Communications of the ACM*, 19(5), 1976.

[19] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, 2007.

[20] 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.

[21] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy*, 2010.

[22] Jonas Magazinius, Alejandro Russo, and Andrei Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.

[23] L.G. Meredith, Mike Stay, and Sophia Drossopoulou. Policy as types. arXiv:1307.7766 [cs.CR], July 2013.

[24] Adrian Mettler, David Wagner, and Tyler Close. Joe-E a Security-Oriented Subset of Java. In *NDSS*, 2010.

[25] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[26] Mark S. Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.

[27] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[28] Mark Samuel Miller. Secure Distributed Programming with Object-capabilities in JavaScript. Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com, October 2011.

[29] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.

[30] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.

[31] Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.

[32] Roger Needham. Protection systems and protection implementations. In *Joint Computer Conference*, pages 571–578, 1972.

[33] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.

[34] Matthew Parkinson. Class invariants: the end of the road? In *IWACO*, 2007.

[35] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of javascript sandboxing. In *USENIX Security*, 2011.

[36] 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.

[37] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[38] Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible access control for JavaScript. In *OOPSLA*, pages 305–322, 2013.

[39] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *CACM*, 17(7):p.389ff, 1974.

[40] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames. *ToPLAS*, 2012.

[41] Marc Stiegler. The lazy programmer's guide to security. HP Labs, www.object-oriented-security.org.

[42] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[43] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for Flexible Object Invariants. In *IWACO*, ACM DL, July 2009.

[44] 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.

[45] 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.

[46] The NewpeakTeam. Several Newspeak Documents. `newspeaklanguage.org/`, September 2012.

[47] Tom van Cutsem. Membranes in Javascript. prog.vub.ac.be/ tvcutsem/invokedynamic/js-membranes.

[48] M. V. Wilkes and R. M. Needham. The Cambridge CAP computer and its operating system, 1979.

[49] Niklaus Wirth. *Programming in Modula-2*. Springer, 1982.

[50] T. Wood and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.

[51] Wei Xiong. *Verification and Validation of JavaScript*. PhD thesis, 2013, Durham University.

[52] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.

A. Formal Definition of $\mathcal{C}_{\mathtt{j}}$

A.1 Modules and Syntax

In figure 2 we give the syntax of our toy language. 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

$$\begin{array}{lll}
\textit{Module} & = & \textit{ClassId} \longrightarrow (\ \textit{ea} \quad \times \\
 & & \qquad \textit{ClassId} \quad \times \\
 & & \qquad (\textit{FieldId} \longrightarrow \textit{ma pa ClassId}\ ) \quad \times \\
 & & \qquad (\textit{MethId} \times \mathbb{N} \ \longrightarrow \textit{ea pa meth}\ ) \quad) \\
\textit{ea} & ::= & \mathsf{noext} \ | \ \mathsf{exts} \\
\textit{pa} & ::= & \mathsf{priv} \ | \ \mathsf{pub} \\
\textit{ma} & ::= & \mathsf{fin} \ | \ \mathsf{mut} \\
\textit{meth} & ::= & \textit{ClassId}\ m \ \ (\ (\textit{ClassId ParId})^* \ ) \ \{ \ e \ \} \\
e & ::= & e\,.f \ | \ e\,.f{:=}e \ | \ e\,.m\,(\,e^*\,) \ | \ \mathsf{new}\,c\,(\,e^*\,) \ | \ \mathsf{x} \ | \ \mathsf{this} \ | \ \mathsf{null} \\
\mathsf{code} & ::= & \qquad\qquad\qquad\qquad e
\end{array}$$

Figure 2. Syntax

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. We do not support overloading.

Method definitions map mat hod identifiers and natural numbers to methods. The natural number reflects the number of formal parameters, and thus supports a simple kind of overloading. For example, the class definition for Mint corresponds to the module $\mathsf{M_{Mint}}$, such that $dom(\mathsf{M_{Mint}}) = \{(\mathsf{Mint})\}$, and where $\mathsf{M}(\mathsf{Mint}) = (\mathsf{noext}, \mathsf{Object}, \emptyset, Ms_0)$, where we use $\emptyset$ to indicate an empty mapping.

On the other hand, Purse corresponds to the module $\mathsf{M_{Purse}}$, such that $dom(\mathsf{M_{Purse}}) = \{\mathsf{Purse}\}$, and $\mathsf{M}(\mathsf{Purse}) = (\mathsf{noext}, \mathsf{Object}, Fs, Ms)$, where $dom(Fs) = \{\mathsf{mint}, \mathsf{balance}\}$, and $dom(Ms) = \{(\mathsf{constr}, 1), (\mathsf{constr}, 2), (\mathsf{deposit}, 2)\}$. We elide the full definition of Fs and Ms, but mention that $Fs(\mathsf{balance}) = (\mathsf{mut}, \mathsf{priv}, \mathsf{int})$.

A.2 The Type System of $\mathcal{C}_\mathsf{j}$

The lookup functions $\mathcal{F}$ and $\mathcal{M}$ return corresponding field and method definitions; they are defined in figure 3. Note that method lookup ($\mathcal{M}$) returns the method definition as well as the class where the method was found - the latter information is necessary to be stored in the stack frame.

Typing is expressed through the judgment $\Gamma \vdash \mathsf{e} : \mathsf{c}$. The typing environment, Γ, is a mapping from thisand any method parameter to a identifier, and from constr to a boolean (to indicate whether Γ belongs to a constructor's method body). We need the latter in order to allow assignment to final fields of the receiver within the constructor.

Subtyping, $\mathsf{M} \vdash c <: c'$, is defined as the reflexive transitive closure of the subclass relation, where $\mathsf{M}(\mathsf{c}) = (\_, \mathsf{c}', \_, \_)$ implies that $\mathsf{M} \vdash \mathsf{c} <: \mathsf{c}'$. The rules for typing appear in figure 4.

A.3 Well-formed class and module, linking

Well formed classes is defined in figure 5. 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:

$$* \ : \ Module \times Module \ \longrightarrow \ Module$$

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

where

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

A.4 Runtime configurations

Runtime configurations, defined in Fig. 6, consist of a stack frame, ϕ, and a heap χ. The frame is a tuple consisting of the address belonging to the receiver (this), a mapping from parameter identifiers to values, the class identifier (c) and the method identifier (m) from which the current method has been taken. 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 from field identifiers onto values.

We define what it means for an address ι to belong to certain class c in a given configuration as follows:

DEFINITION 1. $\iota :_{\kappa,\mathsf{M}} \mathsf{c}$ *iff* $\mathsf{M} \vdash \kappa \downarrow_2 (\iota) \downarrow_1 <: \mathsf{c}$.

In other words, $\iota :_{\kappa,\mathsf{M}} \mathsf{c}$ if the class of the object at ι is a subclass of c, judged in the module M. We also use the notation $\iota :_\kappa \mathsf{c}$ as a shorthand for $\iota :_{\kappa,\mathsf{M}} \mathsf{c}$ when the module can be taken from the context.

We also define what it means for a runtime configuration to belong to a class c or a module M:

DEFINITION 2. $\kappa \in \mathsf{c}$ *iff* $(\kappa \downarrow_1) \downarrow_3 = \mathsf{c}$.

DEFINITION 3. $\kappa \in \mathsf{M}$ *iff* $\kappa \in \mathsf{c}$ *and* $\mathsf{c} \in dom(\mathsf{M})$.

In other words, $\kappa \in \mathsf{c}$ if the currently executing method in κ comes from c, and $\kappa \in \mathsf{M}$ if the class of the currently executing method is defined in M.

A.5 Snapshots

An execution snapshot distills all the runtime information during one moment of execution. It consists of a runtime configuration and expression.

$$Snapshot \ = \ RTConf \times Expr$$

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

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

Figure 3. Lookup

VarThisNull

$$\frac{\mathsf{M}(\mathsf{c}) \text{ is defined}}{\begin{array}{l} \mathsf{M},\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x}) \\ \mathsf{M},\Gamma \vdash \mathsf{this} : \Gamma(\mathsf{this}) \\ \mathsf{M},\Gamma \vdash \mathsf{null} : \mathsf{c} \end{array}}$$

fldRd

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

fldAss

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

fldInitConstr

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

constrCall

$$\frac{\begin{array}{l} \mathcal{M}(\mathsf{M},\mathsf{c},\mathsf{constr},\mathsf{n}) = \_\; pa\; \mathsf{c}\; \mathsf{constr}(\; \mathsf{c}_1\, \mathsf{x}_1, \ldots .\mathsf{c}_\mathsf{n}\, \mathsf{x}_\mathsf{n}\;)\; \{\; \mathsf{e}\; \} \\ \mathsf{M},\Gamma \vdash \mathsf{e}_\mathsf{i} : \mathsf{c}'_\mathsf{i} \quad \text{for } i \in \{1..n\} \\ pa = \mathsf{pub} \vee \Gamma(\mathsf{this}) = \mathsf{c} \\ \mathsf{M} \vdash \mathsf{c}'_\mathsf{i} \leq \mathsf{c}_\mathsf{i} \quad \text{for } i \in \{1..n\} \end{array}}{\mathsf{M},\Gamma \vdash \mathsf{new}\, \mathsf{c}(\; \mathsf{e}_1, \ldots \mathsf{e}_\mathsf{n}) : \mathsf{c}}$$

methCall

$$\frac{\begin{array}{l} \mathsf{M},\Gamma \vdash \mathsf{e} : \mathsf{c} \qquad \mathsf{M},\Gamma \vdash \mathsf{e}_\mathsf{i} : \mathsf{c}'_\mathsf{i} \quad \text{for } i \in \{1..n\} \\ pa = \mathsf{pub} \vee \Gamma(\mathsf{this}) = \mathsf{c} \\ \mathcal{M}(\mathsf{M},\mathsf{c},\mathsf{m},\mathsf{n}) = (ea\; pa\; \mathsf{c}'\; \mathsf{m}(\; \mathsf{c}_1\, \mathsf{x}_1, \ldots .\mathsf{c}_\mathsf{n}\, \mathsf{x}_\mathsf{n})\; \{\; \_\; \}\;, \_) \\ \mathsf{M} \vdash \mathsf{c}'_\mathsf{i} \leq \mathsf{c}_\mathsf{i} \quad \text{for } i \in \{1..n\} \end{array}}{\mathsf{M},\Gamma \vdash \mathsf{e}_0.\mathsf{m}(\; \mathsf{e}_1, \ldots \mathsf{e}_\mathsf{n}) : \mathsf{c}'}$$

Figure 4. Types

well_formed_class

$$\frac{\begin{array}{l} \mathsf{M}(\mathsf{c}) = ea\; \mathsf{c}' \ldots \\ \mathsf{c}' = \mathsf{Object} \vee (ea = \mathsf{exts} \wedge \mathsf{M}(\mathsf{c}') \text{ is defined}) \\ \forall \mathsf{f} :\; \mathsf{M}(\mathsf{c}) \downarrow_3 (\mathsf{f}) = \mathsf{c}' \implies \mathsf{M}(\mathsf{c}') \text{ is defined} \wedge \mathcal{F}(\mathsf{M},\mathsf{c}',\mathsf{f}) \text{is not defined} \\ \forall \mathsf{m} : Id, n : \mathbb{N} :\; \mathsf{M}(\mathsf{c}) \downarrow_4 (\mathsf{m},n) = ea'\; pa\; \mathsf{c}'\; \mathsf{m}(\; \mathsf{c}_1\, \mathsf{x}_1, \ldots .\mathsf{c}_\mathsf{n}\, \mathsf{x}_\mathsf{n})\; \{\; \mathsf{e}\; \} \implies \\ \qquad \mathsf{M}(\mathsf{c}'), \mathsf{M}(\mathsf{c}'') \text{ are defined} \\ \qquad \Gamma(\mathsf{this}) = \mathsf{c},\; \Gamma(\mathsf{x}_\mathsf{i}) = \mathsf{c}_\mathsf{i},\; \text{for } i \in \{1..n\},\; \Gamma(\mathsf{constr}) = (\mathsf{m} = \mathsf{constr}) \\ \qquad \mathsf{M},\Gamma \vdash \mathsf{e} : \mathsf{c}''' \qquad \mathsf{M} \vdash \mathsf{c}''' \leq \mathsf{c}' \\ \qquad \mathcal{M}(\mathsf{M},\mathsf{c}',\mathsf{m},\mathsf{n}) \text{ is undefined} \;\vee\; (ea' = \mathsf{exts} \wedge \mathcal{M}(\mathsf{M},\mathsf{c}',\mathsf{m}) = (ea'\; pa\; \mathsf{c}'\; \mathsf{m}(\; \mathsf{c}_1\, \mathsf{x}_1, \ldots .\mathsf{c}_\mathsf{n}\, \mathsf{x}_\mathsf{n})\; \{\; \_\; \}\;, \_)) \end{array}}{\mathsf{M} \vdash \mathsf{c}\Diamond}$$

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

Figure 5. Well Formed classes and modules

$\kappa \in RTConf$	=	$frame \times heap$	$\phi \in frame$	=	$addr \times (ParId \to Value) \times ClassId \times MethId$
$\chi \in heap$	=	$addr \longrightarrow object$	$\mathsf{v} \in val$	=	$\{\ \mathsf{null}\ \} \cup addr$
$object$	=	$ClassId \times (\ FieldId \longrightarrow val\ )$	$\iota \in addr$	=	$\mathbb{N}$

Figure 6. Runtime Configurations

A.6 The Operational Semantics of $\mathcal{C}_{\mathsf{j}}$

Execution uses module M, and maps a snapshot onto a new configuration κ' and a result.

$$\begin{array}{lcl} \leadsto & : & \mathit{Module} \times \mathit{Snapshot} \longrightarrow \mathit{RTConf} \times \mathit{res} \\ \mathit{res} & = & \{ \,\mathsf{nullPntrExc},\ \mathsf{stuckErr}\, \} \cup \mathit{val} \end{array}$$

In Figure 7 we give the salient rules for execution; we omit those which raise and propagate exceptions.

A.7 $\mathcal{R}each$: Reachable Snapshots

We are interested in the snapshots required for the evaluation of some other snapshot after *any* number of steps. $\mathcal{R}each(\mathsf{M}, \kappa, e)$ returns the sets of all snapshots reachable from κ, e:

$$\mathcal{R}each \ : \ Program \times Snapshot \longrightarrow \mathcal{P}(Snapshot)$$

We define the function $\mathcal{R}each$ in terms of $\mathcal{R}each_{aux}$ as follows:

$$\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code}) = \{ \ (\kappa, \mathsf{code}) \ \} \ \cup \ \mathcal{R}each_{aux}(\mathsf{M}, \kappa, \mathsf{code})$$

We define the function $\mathcal{R}each_{aux}$ in Figure 8, by cases on the structure of the expression, and depending on the execution of the expression. Note that the function $\mathcal{R}each(\mathsf{M}, \kappa, e)$ is defined, 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, $\mathcal{R}each(\mathsf{M}, \kappa, \mathsf{code})$ may be an infinite set.

A.8 $\mathcal{A}rising$: snapshots reachable from initial

We define as initial snapshots those that may be encountered at the start of program execution.

$$\mathcal{I}nit \ : \ Program \longrightarrow \mathcal{P}(Snapshot)$$

Initial snapshots should be "minimal". 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.

$$\begin{array}{lcl} \mathcal{I}nit(\mathsf{M}) & = & \{ \ (\ \kappa_0, \mathsf{e}) \mid \exists \mathsf{c}.\ \ \mathsf{M}, \Gamma_0 \vdash \mathsf{e} : \mathsf{c} \ \} \end{array}$$

where $\kappa_0 \ = \ ((\iota_0, \emptyset, \_, \_), \chi_0)$,
and $dom(\chi_0) = \{ \ \iota_0 \ \}$, and $\chi_0(\iota_0) = (\mathsf{Object}, \emptyset)$,
and $\Gamma_0 = \mathsf{this} \mapsto \mathsf{Object}, \mathsf{c}onstr \mapsto \mathsf{false}$.

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

$$\mathcal{A}rising \ : \ Program \longrightarrow \mathcal{P}(Snapshot)$$

Arising configurations are defined as follows:

$$\mathcal{A}rising(\mathsf{M}) \quad = \quad \bigcup_{(\kappa, \mathsf{e}) \in \mathcal{I}nit(\mathsf{M})} \mathcal{R}each(\mathsf{M}, \kappa, \mathsf{e})$$

A.9 $\mathcal{U}sed$: objects read by execution of a snapshot

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

$$\mathcal{U}sed \ : \ Program \times Snapshot \longrightarrow \mathcal{P}(Addr)$$

$\mathcal{U}sed$ 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.7. The definitions appear in Figure 9.

A.10 $\mathcal{A}cc\mathcal{A}ll$: objects accessible through any paths, $\mathcal{A}cc\mathcal{P}ub$: 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 $\mathcal{A}cc\mathcal{A}ll$: 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. The definitions appear in Figure 10.

$$\begin{array}{l} \mathcal{A}cc\mathcal{A}ll \ : \ Program \times RTConfig \mapsto \mathcal{P}(Addr) \\ \mathcal{A}cc\mathcal{P}ub \ : \ Program \times RTConfig \mapsto \mathcal{P}(Addr) \end{array}$$

These sets are defined in Figure 10.

LEMMA 1. *For all M, κ, κ',* e, v

1. $\mathcal{A}cc\mathcal{P}ub(\mathsf{M}, \kappa) \subseteq \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$
2. $\mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{e}) \cap dom(\kappa \downarrow_1) \subseteq \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$
3. $\mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa') \cap dom(\kappa \downarrow_1) \subseteq \mathcal{A}cc\mathcal{A}ll(\mathsf{M}, \kappa)$ *for all* κ' *such that* $(\kappa', \_) \in \mathcal{R}eaches(\mathsf{M}, \kappa)$.

The proof of (1) proceeds by structural induction on the definition of $\mathcal{A}cc\mathcal{A}ll$. The proof of (2) follows from (3); the proof of (3) by structural induction over the definition of $\mathcal{R}eaches$ and $\mathcal{A}cc\mathcal{A}ll$. Note that in general, there is no subset relation between $\mathcal{A}cc\mathcal{P}ub(\mathsf{M}, \kappa)$ and $\mathcal{U}sed(\mathsf{M}, \kappa, \mathsf{e})$.

A.11 Alternative Implementation of the Mint Example

An alternative implementation of the mint example appears in figure 11 .

val

$$\frac{}{\mathsf{M},\ \kappa,\ \mathsf{v}\ \rightsquigarrow\ \kappa,\ \mathsf{v}}$$

thisPar

$$\frac{\kappa = (\iota, vals, \_, \_), \chi}{\begin{array}{l}\mathsf{M},\ \kappa,\ \mathsf{this}\ \rightsquigarrow\ \kappa,\ \iota\\ \mathsf{M},\ \kappa,\ \mathsf{x}\ \rightsquigarrow\ \kappa,\ vals(\mathsf{x})\end{array}}$$

fldRd

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

fldAss

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

new

$$\frac{\begin{array}{l}\kappa_1 = \kappa\\ \mathsf{M},\ \kappa_i,\ \mathsf{e_i}\ \rightsquigarrow\ \kappa_{i+1},\ \mathsf{v}_i\ \text{ for } i \in \{1..n\}\\ \{\mathsf{f}\ |\mathcal{F}(\mathsf{M},\mathsf{c},\mathsf{f}) \text{ is defined }\}\ =\ \{\mathsf{f}_1, \ldots, \mathsf{f}_r\}\\ \iota \text{ is new in } \chi''\\ \kappa_n = (\phi, \chi_n, \_, \_)\\ \chi''' = \chi_n[\iota \mapsto (\ \mathsf{c},\ (\mathsf{f}_1 : \mathsf{null}\ , \ldots, \mathsf{f}_r : \mathsf{null}\ )\ )\ ]\\ \mathcal{M}(\mathsf{M}, \mathsf{c}, \mathsf{constr}, \mathsf{n}) = (\_\ \_(\ c_1\ x_1, \ldots. c_n\ x_n)\ \{\ \mathsf{e}'\}\ , \mathsf{c}')\\ vals(x_i) = \mathsf{v}_i\ \text{ for } i \in \{1..n\}\\ \kappa''\ =\ ((\iota, vals, \mathsf{c}, \mathsf{constr}), \chi_n)\\ \mathsf{M},\ \kappa'',\ \mathsf{e}'\ \rightsquigarrow\ \kappa',\ \mathsf{v}'\end{array}}{\mathsf{M},\ \kappa,\ \mathsf{new\,c}(\ \mathsf{e}_1, \ldots \mathsf{e_n})\ \rightsquigarrow\ \kappa',\ \iota}$$

methCall

$$\frac{\begin{array}{l}\mathsf{M},\ \kappa,\ \mathsf{e}\ \rightsquigarrow\ \kappa_1,\ \iota\\ \mathsf{M},\ \kappa_i,\ \mathsf{e_i}\ \rightsquigarrow\ \kappa_{i+1},\ \mathsf{v}_i\ \text{ for } i \in \{1..n\}\\ \kappa_n = (\_, \chi_n)\\ \chi_n(\iota) \downarrow_1 = \mathsf{c}\\ \mathcal{M}(\mathsf{M}, \mathsf{c}, \mathsf{m}, \mathsf{n}) = (\_\ \_\ \_\ m(\ c_1\ x_1, \ldots c_n\ x_n)\ \ \{\ \mathsf{e}'\ \}\ , \mathsf{c}')\\ vals(x_i) = \mathsf{v}_i\ \text{ for } i \in \{1..n\}\\ \kappa''\ =\ ((\iota, vals, \mathsf{c}', \mathsf{m}),\ \chi_n)\\ \mathsf{M},\ \kappa'',\ \mathsf{e}'\ \rightsquigarrow\ \kappa',\ \mathsf{v}\end{array}}{\mathsf{M},\ \kappa,\ \mathsf{e.m}(\ \mathsf{e}_1, \ldots \mathsf{e_n})\ \rightsquigarrow\ \kappa',\ \mathsf{v}}$$

Figure 7. Execution

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{null}) \quad = \quad \emptyset$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{this}) \quad = \quad \emptyset$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{x}) \quad = \quad \emptyset$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e.f}) \quad = \quad \mathcal{R}each_{aux}(\kappa,\mathsf{e})$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e.f} := \mathsf{e}') \quad = \quad \mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e}) \cup \begin{cases} \mathcal{R}each_{aux}(\mathsf{M},\kappa',\mathsf{e}'), & \text{if } \exists\kappa',\mathsf{v}, s.t: \ \ \mathsf{M},\kappa,\mathsf{e} \leadsto \kappa',\mathsf{v} \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{new\ c(\ e)\ }) \quad = \quad \mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e}) \cup \begin{cases} \{\kappa',\mathsf{e}'\} \\ \quad \text{if } (A) : \exists\kappa'',\mathsf{v}, s.t: \ \ \mathsf{M},\kappa,\mathsf{e} \leadsto \kappa'',\mathsf{v}, \\ \quad \mathcal{M}(\mathsf{M},\mathsf{c},\mathsf{constr}) = \_\ \_\ \_\ \_(\ \_\_)\ \{\ \mathsf{e}'\} \\ \quad \text{and where } \kappa'' = \_,\chi'', \ \text{ and } \kappa' = (\iota,\mathsf{v}),\chi''' \\ \quad \iota \text{ new in } \chi'', \ \chi''' = \chi''[\iota\mapsto(\ \mathsf{c},\ (\mathsf{f}_1 : \mathsf{null}\ ,\ldots,\mathsf{f}_r : \mathsf{null}\ )\ )\ ] \\ \quad \{\mathsf{f} \mid \mathcal{F}(\mathsf{M},\mathsf{c},\mathsf{f}) \text{ is defined }\} = \{\mathsf{f}_1,\ldots,\mathsf{f}_r\} \\ \emptyset, \quad \text{otherwise.} \end{cases}$$

$$\cup \begin{cases} \mathcal{R}each_{aux}(\mathsf{M},\kappa',\mathsf{e}'), & \text{if } (A) \text{ as above.} \\ \emptyset, & \text{otherwise.} \end{cases}$$

$$\mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e.m(\ e')\ }) \quad = \quad \mathcal{R}each_{aux}(\mathsf{M},\kappa,\mathsf{e}) \cup \begin{cases} \mathcal{R}each_{aux}(\mathsf{M},\kappa',\mathsf{e}') & \text{if } (B) : \exists\kappa',\iota, s.t: \ \ \mathsf{M},\kappa,\mathsf{e} \leadsto \kappa',\iota, \\ \emptyset, & \text{otherwise.} \end{cases}$$

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

$$\cup \begin{cases} \mathcal{R}each_{aux}(\mathsf{M},\kappa''',\mathsf{e}'') & \text{if } (B) \text{ and } (C), \text{ as above.} \\ \emptyset, & \text{otherwise.} \end{cases}$$

Figure 8. Reachable configurations - for simplicity, we assume that methods and constructors have one parameter

$$
\begin{array}{lcl}
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{null}) & = & \emptyset \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{this}) & = & \{\phi(\mathsf{this})\}\ \text{ where } \kappa = \phi, \_ \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{x}) & = & \{\phi(\mathsf{x})\}\ \text{ where } \kappa = \phi, \_ \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e.f}) & = & \mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e}) \cup \begin{cases} \{\chi'(\iota)\downarrow(\mathsf{f})\} & \text{if } \exists \kappa', \iota, s.t:\ \mathsf{M},\kappa,\mathsf{e} \leadsto (\phi',\chi'),\iota \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e.f} := \mathsf{e}') & = & \mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e.f}) \cup \begin{cases} \mathcal{U}sed(\mathsf{M},\kappa',\mathsf{e}'), & \text{if } \exists \kappa', \mathsf{v}, s.t:\ \mathsf{M},\kappa,\mathsf{e} \leadsto \kappa',\mathsf{v} \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{new\ c(\ e)}\ ) & = & \mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e}) \\
 & & \cup \begin{cases} \{\ \iota\ \} & \text{if } (A), \text{ where } (A) \text{ as defined in A.7} \\ \emptyset, & \text{otherwise.} \end{cases} \\
 & & \cup \begin{cases} \mathcal{U}sed(\mathsf{M},\kappa',\mathsf{e}'), & \text{if } (A) \text{ as above.} \\ \emptyset, & \text{otherwise.} \end{cases} \\
\mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e.m(\ e')}\ ) & = & \mathcal{U}sed(\mathsf{M},\kappa,\mathsf{e}) \\
 & & \cup \begin{cases} \mathcal{U}sed(\mathsf{M},\kappa',\mathsf{e}') & \text{if } (B), \text{ where } (B) \text{ as defined in A.7} \\ \emptyset, & \text{otherwise.} \end{cases} \\
 & & \cup \begin{cases} \{\kappa''', \mathsf{e}''\} & \\ \quad \text{if } (B) \text{ and } (C), \text{ where } (B), (C) \text{ as defined in A.7} & \\ \emptyset, & \text{otherwise.} \end{cases} \\
 & & \cup \begin{cases} \mathcal{R}each(\mathsf{M},\kappa''',\mathsf{e}'') & \text{if } (B) \text{ and } (C), \text{ as above.} \\ \emptyset, & \text{otherwise.} \end{cases}
\end{array}
$$

Figure 9. Objects read - for simplicity, we assume that methods and constructors have one parameter

$$
\begin{array}{lcl}
\mathcal{A}cc\mathcal{A}ll(\mathsf{M},\kappa) & = & \{\ \iota,\ \mathsf{v}\ \}\ \cup\ \{\chi(\iota')\downarrow_2(\mathsf{f})\ \mid\ \iota' \in \mathcal{A}cc\mathcal{A}ll(\mathsf{M},\kappa)\ \wedge\ \chi(\iota')\downarrow_1 = \mathsf{c}'\ \wedge\ \mathcal{F}(\mathsf{M},\mathsf{c}',\mathsf{f}) = \_\ \_\ \} \\
 & & \text{where } \kappa\ =\ ((\iota,\mathsf{v},\_,\_),\chi). \\
\mathcal{A}cc\mathcal{P}ub(\mathsf{M},\kappa) & = & \{\ \iota,\ \mathsf{v}\ \}\ \cup\ \{\chi(\iota')\downarrow_2(\mathsf{f})\ \mid\ \iota' \in \mathcal{A}cc\mathcal{P}ub(\mathsf{M},\kappa)\ \wedge\ \chi(\iota')\downarrow_1 = \mathsf{c}' \\
 & & \qquad \wedge\ \mathcal{F}(\mathsf{M},\mathsf{c}',\mathsf{f}) = \_\ pa\ \_\ \wedge\ (pa = \mathsf{pub} \vee \mathsf{c} = \mathsf{c}')\ \} \\
 & & \text{where } \kappa\ =\ ((\iota,\mathsf{v},\_,\_),\chi).
\end{array}
$$

Figure 10. Accessible objects

```java
public final class Purse {       }   //  Purse pure capability

public final class Mint {
    private final HashMap<Purse,long> database = new HashMap<>();

    public Purse makePurse(long balance) {  // make a new purse
      Purse p = new Purse();
      database.put(p,balance);
      return p;
    }

    public void deposit(Purse from, Purse into, long amnt) {
       if ( (amount < 0)   ||
            (!database.contains(from)) ||
            (database.get(from) < amnt) ||
            (!database.contains(into)) )
                      { throw new IllegalArgtException(); };
       database.put(from, database.get(from) - amnt);
       database.put(into,   database.get(into) + amnt);
    }
}
```

Figure 11. An alternative Mint implementation using a map as a database.