# Secure the Clones - Static Enforcement of Policies for Secure Object Copying

Thomas Jensen, Florent Kirchner, David Pichardie

# Secure the Clones *

## Static Enforcement of Policies for Secure Object Copying

Thomas Jensen, Florent Kirchner, and David Pichardie

INRIA Rennes – Bretagne Atlantique, France
firstname.lastname@inria.fr

**Abstract.** Exchanging mutable data objects with untrusted code is a delicate matter because of the risk of creating a data space that is accessible by an attacker. Consequently, secure programming guidelines for Java stress the importance of using defensive copying before accepting or handing out references to an internal mutable object. However, implementation of a copy method (like clone()) is entirely left to the programmer. It may not provide a sufficiently deep copy of an object and is subject to overriding by a malicious sub-class. Currently no language-based mechanism supports secure object cloning. This paper proposes a type-based annotation system for defining modular copy policies for class-based object-oriented programs. A copy policy specifies the maximally allowed sharing between an object and its clone. We present a static enforcement mechanism that will guarantee that all classes fulfill their copy policy, even in the presence of overriding of copy methods, and establish the semantic correctness of the overall approach in Coq. The mechanism has been implemented and experimentally evaluated on clone methods from several Java libraries.

## 1 Introduction

Exchanging data objects with untrusted code is a delicate matter because of the risk of creating a data space that is accessible by an attacker. Consequently, secure programming guidelines for Java such as those proposed by Sun [13] and CERT [5] stress the importance of using defensive *copying* or *cloning* before accepting or handing out references to an internal mutable object. There are two aspects of the problem:

1. If the result of a method is a reference to an internal mutable object, then the receiving code may modify the internal state. Therefore, it is recommended to make copies of mutable objects that are returned as results, unless the intention is to share state.
2. If an argument to a method is a reference to an object coming from hostile code, a local copy of the object should be created. Otherwise, the hostile code may be able to modify the internal state of the object.

A common way for a class to provide facilities for copying objects is to implement a `clone()` method that overrides the cloning method provided by `java.lang.Object`. The following code snippet, taken from Sun's Secure Coding Guidelines for Java, demonstrates how a `date` object is cloned before being returned to a caller:

* This work was supported by the ANSSI, the ANR, and the *Région Bretagne*, respectively under the Javasec, Parsec, and Certlogs projects.

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone(); }
}
```

However, relying on calling a polymorphic `clone` method to ensure secure copying of objects may prove insufficient, for two reasons. First, the implementation of the `clone()` method is entirely left to the programmer and there is no way to enforce that an untrusted implementation provides a sufficiently *deep* copy of the object. It is free to leave references to parts of the original object being copied in the new object. Second, even if the current `clone()` method works properly, sub-classes may override the `clone()` method and replace it with a method that does not create a sufficiently deep clone. For the above example to behave correctly, an additional class invariant is required, ensuring that the `date` field always contains an object that is of class `Date` and not one of its sub-classes. To quote from the CERT guidelines for secure Java programming: *"Do not carry out defensive copying using the clone() method in constructors, when the (non-system) class can be subclassed by untrusted code. This will limit the malicious code from returning a crafted object when the object's clone() method is invoked."* Clearly, we are faced with a situation where basic object-oriented software engineering principles (sub-classing and overriding) are at odds with security concerns. To reconcile these two aspects in a manner that provides semantically well-founded guarantees of the resulting code, this paper proposes a formalism for defining *cloning policies* by annotating classes and specific copy methods, and a static enforcement mechanism that will guarantee that all classes of an application adhere to the copy policy. We do not enforce that a copy method will always return a target object that is functionally equivalent to its source. Rather, we ensure non-sharing constraints between source and targets, expressed through a copy policy, as this is the security-critical part of a copy method in a defensive copying scenario.

### 1.1 Cloning of Objects

For objects in Java to be cloneable, their class must implement the empty interface `Cloneable`. A default `clone` method is provided by the class `Object`: when invoked on an object of a class, `Object.clone` will create a new object of that class and copy the content of each field of the original object into the new object. The object and its clone share all sub-structures of the object; such a copy is called *shallow*.

It is common for cloneable classes to override the default clone method and provide their own implementation. For a generic `List` class, this could be done as follows:

```
public class List<V> implements Cloneable
{
    public V value;
    public List<V> next;

    public List(V val, List<V> next) {
        this.value = val;
```

```
        this.next = next; }

    public List<V> clone() {
        return new List(value,(next==null)?null:next.clone()); }
}
```

Notice that this cloning method performs a shallow copy of the list, duplicating the spine but sharing all the elements between the list and its clone. Because this amount of sharing may not be desirable (for the reasons mentioned above), the programmer is free to implement other versions of `clone()`. For example, another way of cloning a list is by copying both the list spine and its elements[1], creating what is known as a *deep* copy.

```
public List<V> deepClone() {
  return new List((V) value.clone(),
                  (next==null ? null : next.deepClone())); }
```

A general programming pattern for methods that clone objects works by first creating a shallow copy of the object by calling the `super.clone()` method, and then modifying certain fields to reference new copies of the original content. This is illustrated in the following snippet, taken from the class `LinkedList` in Fig. 8:

```
public Object clone() {  ...
  clone = super.clone();  ...
  clone.header = new Entry<E>(null, null, null); ...
  return clone;}
```

There are two observations to be made about the analysis of such methods. First, an analysis that tracks the depth of the clone being returned will have to be flow-sensitive, as the method starts out with a shallow copy that is gradually being made deeper. This makes the analysis more costly. Second, there is no need to track precisely modifications made to parts of the memory that are not local to the clone method, as clone methods are primarily concerned with manipulating memory that they allocate themselves. This will have a strong impact on the design choices of our analysis.

### 1.2 Copy Policies

The first contribution of the paper is a proposal for a set of semantically well-defined program annotations, whose purpose is to enable the expression of policies for secure copying of objects. Introducing a copy policy language enables class developers to state explicitly the intended behavior of copy methods. In the basic form of the copy policy formalism, fields of classes are annotated with `@Shallow` and `@Deep`. Intuitively, the annotation `@Shallow` indicates that the field is referencing an object, parts of which may be referenced from elsewhere. The annotation `@Deep(X)` on a field `f` means that *a*) the object referenced by this field `f` cannot itself be referenced from elsewhere, and *b*) the field `f` is copied according to the copy policy identified by `X`. Here, `X` is either the name of a specific policy or if omitted, it designates the default policy of the class of the field. For example, the following annotations:

[1] To be type-checked by the Java compiler it is necessary to add a cast before calling `clone()` on `value`. A cast to a sub interface of `Cloneable` that declares a `clone()` method is necessary.

```
class List  { @Shallow V value;  @Deep List next;  ...}
```

specifies a default policy for the class `List` where the `next` field points to a list object that also respects the default copy policy for lists. Any method in the `List` class, labelled with the `@Copy` annotation, is meant to respect this default policy.

In addition it is possible to define other copy policies and annotate specific *copy methods* (identified by the annotation `@Copy(...)`) with the name of these policies. For example, the annotation[2]

```
DL: { @Deep V value; @Deep(DL) List next;};
@Copy(DL) List<V> deepClone() {
  return new List((V) value.clone(),
                  (next==null ? null : next.deepClone())); }
```

can be used to specify a list-copying method that also ensures that the `value` fields of a list of objects are copied according to the copy policy of their class (which is a stronger policy than that imposed by the annotations of the class `List`). We give a formal definition of the policy annotation language in Section 2.

The annotations are meant to ensure a certain degree of non-sharing between the original object being copied and its clone. We want to state explicitly that the parts of the clone that can be accessed via fields marked `@Deep` are unaccessible from any part of the heap that was accessible before the call to `clone()`. To make this intention precise, we provide a formal semantics of a simple programming language extended with policy annotations and define what it means for a program to respect a policy (Section 2.2).

### 1.3 Enforcement

The second major contribution of this work is to make the developer's intent, expressed by copy policies, statically enforceable using a type system. We formalize this enforcement mechanism by giving an interpretation of the policy language in which annotations are translated into graph-shaped type structures. For example, the annotations of the `List` class defined above will be translated into the graph that is depicted to the right in Fig. 1 (`res` is the name given to the result of the copy method). The left part shows the concrete heap structure.

Unlike general purpose shape analysis, we take into account the programming methodologies and practice for copy methods, and design a type system specifically tailored to the enforcement of copy policies. This means that the underlying analysis must be able to track precisely all modifications to objects that the copy method allocates itself (directly or indirectly) in a flow-sensitive manner. Conversely, as copy methods should not modify non-local objects, the analysis will be designed to be more approximate when tracking objects external to the method under analysis, and the type system will accordingly refuse methods that attempt such non-local modifications. As a further design choice, the annotations are required to be verifiable modularly on a class-by-class basis without having to perform an analysis of the entire code base, and at a reasonable cost.

[2] Our implementation uses a sightly different policy declaration syntax because of the limitations imposed by the Java annotation language.

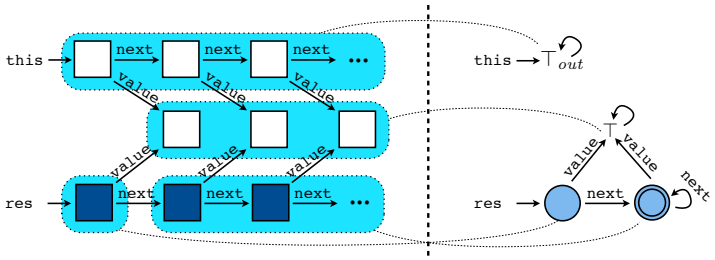



Fig. 1: A linked structure (left part) and its abstraction (right part).

As depicted in Fig. 1, concrete memory cells are either abstracted as *a*) $\top_{out}$ when they are not allocated in the copy method itself (or its callee); *b*) $\top$ when they are just marked as *maybe-shared*; and *c*) circle nodes of a deterministic graph when they are locally allocated. A single circle furthermore expresses a singleton concretization. In this example, the abstract heap representation matches the graph interpretation of annotations, which means that the instruction set that produced this heap state satisfies the specified copy policy.

Technically, the intra-procedural component of our analysis corresponds to heap shape analysis with the particular type of graphs that we have defined. Operations involving non-local parts of the heap are rapidly discarded. Inter-procedural analysis uses the signatures of copy methods provided by the programmer. Inheritance is dealt with by stipulating that inherited fields retain their "shallow/deep" annotations. Redefinition of a method must respect the same copy policy and other copy methods can be added to a sub-class. The detailed definition of the analysis, presented as a set of type inference rules, is given in Section 3.

## 2 Language and Copy Policies

$$x, y \in Var \qquad f \in Field \qquad m \in Meth \qquad cn \in Class_{\text{id}} \qquad X \in Policy_{\text{id}}$$

$$\begin{array}{rrl}
p \in & Prog & ::= \overline{cl} \\
cl \in & Class & ::= class\ cn\ [extends\ cn]\ \{\overline{pd}\ \overline{md}\} \\
pd \in & PolicyDecl & ::= X : \{\tau\} \\
\tau \in & Policy & ::= \overline{(X, f)} \\
md \in & MethDecl & ::= \mathsf{Copy}(X)\ m(x){:=}c \\
c \in & Comm & ::= x{:=}y \mid x{:=}y.f \mid x.f{:=}y \mid x{:=}null \\
 & & \quad \mid x := new\ cn \mid x{:=}m_{cn:X}(y) \mid x{:=}?(y) \mid return\ x \\
 & & \quad \mid c; c \mid if\ (*)\ then\ c\ else\ c\ fi \mid while\ (*)\ do\ c\ done
\end{array}$$

**Notations:** We write $\preceq$ for the reflexive transitive closure of the subclass relation induced by a (well-formed) program that is fixed in the rest of the paper. We write $\overline{x}$ a sequence of syntactic elements of form $x$.

Fig. 2: Language Syntax.

The formalism is developed for a small, imperative language extended with basic, class-based object-oriented features for object allocation, field access and assignment, and method invocation. A program is a collection of classes, organized into a tree-structured class hierarchy via the *extends* relation. A class consists of a series of copy

method declarations with each its own policy $X$, its name $m$, its formal parameter $x$ and commands $c$ to execute. A sub-class inherits the copy methods of its super-class and can re-define a copy method defined in one of its super-classes. We only consider copy methods. Private methods (or static methods of the current class) are inlined by the type checker. Other method calls (to virtual methods) are modeled by a special instruction $x{:=}?(y)$ that assigns an arbitrary value to $x$ and possibly modifies all heap cells reachable from $y$ (except itself). The other commands are standard. The copy method call $x{:=}m_{cn:X}(y)$ is a virtual call. The method to be called is the copy method of name $m$ defined or inherited by the (dynamic) class of the object stored in variable $y$. The subscript annotation $cn{:}X$ is used as a static constraint. It is supposed that the type of $y$ is guaranteed to be a sub-class of class $cn$ and that $cn$ defines a method $m$ with a copy policy $X$. This is ensured by standard bytecode verification and method resolution.

We suppose given a set of policy identifiers $Policy_{\text{id}}$, ranged over by $X$. A copy policy declaration has the form $X : \{\tau\}$ where $X$ is the identifier of the policy signature and $\tau$ is a policy. The policy $\tau$ consists of a set of field annotations $(X, f)\ ;\ \ldots$ where $f$ is a *deep* field that should reference an object which can only be accessed via the returned pointer of the copy method and which respects the copy policy identified by $X$. The use of policy identifiers makes it possible to write recursive definitions of copy policies, necessary for describing copy properties of recursive structures. Any other field is implicitly *shallow*, meaning that no copy properties are guaranteed for the object referenced by the field. No further copy properties are given for the sub-structure starting at *shallow* fields. For instance, the default copy policy of the class `List` presented in Sec. 1.2 writes: $\{(\texttt{List.default}, \texttt{next})\}$.

We assume that for a given program, all copy policies have been grouped together in a finite map $\Pi_p : Policy_{\text{id}} \to Policy$. In the rest of the paper, we assume this map is complete, *i.e.* each policy name $X$ that appears in an annotation is bound to a unique policy in the program $p$.

The semantic model of the language defined here is store-based:

$$\begin{array}{rl} l \in \mathit{Loc} & \\ v \in \mathit{Val} & = \mathit{Loc} \cup \{\diamond\} \\ \rho \in \mathit{Env} & = \mathit{Var} \to \mathit{Val} \\ o \in \mathit{Object} & = \mathit{Field} \to \mathit{Val} \\ h \in \mathit{Heap} & = \mathit{Loc} \rightharpoonup_{\text{fin}} (\mathit{Class}_{\text{id}} \times \mathit{Object}) \\ \langle \rho, h, A \rangle \in \mathit{State} & = \mathit{Env} \times \mathit{Heap} \times \mathcal{P}(\mathit{Loc}) \end{array}$$

A program state consists of an environment $\rho$ of local variables, a store $h$ of locations mapping[3] to objects in a heap and a set $A$ of locally allocated locations in the current method or one of its callees. This last component does not influence the semantic transitions: it is necessary to express the type system interpretation exposed in Sec. 3, but is not used in the final soundness theorem. Each object is modeled in turn as a finite function from field names to values (references or the specific $\diamond$ reference for null values). We do not deal with base values such as integers because their immutable values are irrelevant here.

The operational semantics of the language is defined (Fig. 3) by the evaluation relation $\rightsquigarrow$ between configurations $\mathit{Comm} \times \mathit{State}$ and resulting states $\mathit{State}$. The set of

[3] We note $\rightharpoonup_{\text{fin}}$ for partial functions on finite domains.

$$\frac{}{(x{:=}y, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto \rho(y)], h, A\rangle} \quad \frac{}{(x{:=}null, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto \diamond], h, A\rangle}$$

$$\frac{\rho(y) \in \mathrm{dom}(h)}{(x{:=}y.f, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto h(\rho(y), f)], h, A\rangle} \quad \frac{\rho(x) \in \mathrm{dom}(h)}{(x.f{:=}y, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho, h[(\rho(x), f) \mapsto \rho(y)], A\rangle}$$

$$\frac{l \notin \mathrm{dom}(h)}{(x := new\ cn, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto l], h[l \mapsto (cn, o_\diamond)], A \cup \{l\}\rangle}$$

$$\frac{}{(return\ x, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[ret \mapsto \rho(x)], h, A\rangle}$$

$$\frac{\begin{array}{c} h(\rho(y)) = (cn_y, \_) \quad lookup(cn_y, m) = \big(\mathsf{Copy}(X')\ m(a){:=}c\big) \quad cn_y \preceq cn \\ (c, \langle \rho_\diamond[a \mapsto \rho(y)], h, \emptyset\rangle) \rightsquigarrow \langle \rho', h', A'\rangle \end{array}}{(x{:=}m_{cn:X}(y), \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto \rho'(ret)], h', A \cup A'\rangle}$$

$$\frac{\begin{array}{c} \mathrm{dom}(h) \subseteq \mathrm{dom}(h') \quad \forall l \in \mathrm{dom}(h) \setminus \mathrm{Reach}_h(\rho(y)),\ h(l) = h'(l) \\ \forall l \in \mathrm{dom}(h) \setminus \mathrm{Reach}_h(\rho(y)),\ \forall l', l \in \mathrm{Reach}_{h'}(l') \Rightarrow l' \in \mathrm{dom}(h) \setminus \mathrm{Reach}_h(\rho(y)) \\ v \in \{\diamond\} + \mathrm{Reach}_h(\rho(y)) \cup (\mathrm{dom}(h') \setminus \mathrm{dom}(h)) \end{array}}{(x{:=}?(y), \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho[x \mapsto v], h', A \backslash \mathrm{Reach}_h^+(\rho(y))\rangle}$$

$$\frac{(c_1, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_1, h_1, A_1\rangle \quad (c_2, \langle \rho_1, h_1, A_1\rangle) \rightsquigarrow \langle \rho_2, h_2, A_2\rangle}{(c_1; c_2, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_2, h_2, A_2\rangle}$$

$$\frac{(c_1, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_1, h_1, A_1\rangle}{(if\ (*)\ then\ c_1\ else\ c_2\ fi, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_1, h_1, A_1\rangle} \quad \frac{(c_2, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_2, h_2, A_2\rangle}{(if\ (*)\ then\ c_1\ else\ c_2\ fi, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho_2, h_2, A_2\rangle}$$

$$\frac{}{(while\ (*)\ do\ c\ done, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho, h, A\rangle} \quad \frac{(c; while\ (*)\ do\ c\ done, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho', h', A'\rangle}{(while\ (*)\ do\ c\ done, \langle \rho, h, A\rangle) \rightsquigarrow \langle \rho', h', A'\rangle}$$

**Notations:** We write $h(l, f)$ for the value $o(f)$ such that $l \in \mathrm{dom}(h)$ and $h(l) = o$. We write $h[(l, f) \mapsto v]$ for the heap $h'$ that is equal to $h$ except that the $f$ field of the object at location $l$ now has value $v$. Similarly, $\rho[x \mapsto v]$ is the environment $\rho$ modified so that $x$ now maps to $v$. The object $o_\diamond$ is the object satisfying $o_\diamond(f) = \diamond$ for all field $f$, and $\rho_\diamond$ is the environment such that $\rho_\diamond(x) = \diamond$ for all variables $x$. We consider methods with only one parameter and name it $p$. *lookup* designates the dynamic lookup procedure that, given a class name $cn$ and a method name $m$, find the first implementation of $m$ in the class hierarchy starting from the class of name $cn$ and scanning the hierarchy bottom-up. It returns the corresponding method declaration. $ret$ is a specific local variable name that is used to store the result of each method. $\mathrm{Reach}_h(l)$ (resp. $\mathrm{Reach}_h^+(l)$) denotes the set of values that are reachable from any sequence (resp. any non-empty sequence) of fields in $h$.

Fig. 3: Semantic Rules.

locally allocated locations is updated by both the $x := new\ cn$ and the $x{:=}m_{cn:X}(y)$ statements. The execution of an unknown method call $x{:=}?(y)$ results in a new heap $h'$ that keeps all the previous objects that were not reachable from $\rho(l)$. It assigns the variable $x$ a reference that was either reachable from $\rho(l)$ in $h$ or that has been allocated during this call and hence not present in $h$.

## 2.1 Policies and Inheritance

We impose restrictions on the way that inheritance can interact with copy policies. A method being re-defined in a sub-class can impose further constraints on how fields of the objects returned as result should be copied. A field already annotated *deep* with policy $X$ must have the same annotation in the policy governing the re-defined method but a field annotated as *shallow* can be annotated *deep* for a re-defined method.

**Definition 1 (Overriding Copy Policies).** *A program $p$ is well-formed with respect to overriding copy policies if and only if for any method declaration* $\mathsf{Copy}(X')\ m(x){:=}\ldots$ *that overrides (*i.e. *is declared with this signature in a subclass of a class $cl$) another method declaration* $\mathsf{Copy}(X)\ m(x){:=}\ldots$ *declared in $cl$, we have*

$$\Pi_p(X) \subseteq \Pi_p(X').$$

*Example 1.* The `java.lang.Object` class provides a `clone()` method of policy $\{\}$ (because its native `clone()` method is *shallow* on all fields). A class `A` declaring two fields `f` and `g` can hence override the `clone()` method and give it a policy $\{(X, \mathsf{g})\}$. If a class `B` extends `A` and overrides `clone()`, it must assign it a policy of the form $\{(X, \mathsf{g});\ \ldots\ \}$ and could declare the field `f` as *deep*. In our implementation, we let the programmer leave the policy part that concerns fields declared in superclasses implicit, as it is systematically inherited.

### 2.2 Semantics of Copy Policies

The informal semantics of the copy policy annotation of a method is:

> A copy method satisfies a copy policy $X$ if and only if no memory cell that is reachable from the result of this method following only fields with *deep* annotations in $X$, is reachable from another local variable of the caller.

We formalize this by giving, in Fig. 4, a semantics to copy policies based on access paths. An access path consists of a variable $x$ followed by a sequence of field names $f_i$ separated by a dot. An access path $\pi$ can be evaluated to a value $v$ in a context $\langle \rho, h \rangle$ with a judgment $\langle \rho, h \rangle \vdash \pi \Downarrow v$. Each path $\pi$ has a root variable $\downarrow\pi \in \mathit{Var}$. A judgment $\vdash \pi : \tau$ holds when a path $\pi$ follows only deep fields in the policy $\tau$.

**Access path syntax**

$$\pi \in \mathbb{P} ::= x \mid \pi.f$$

**Access path evaluation**

$$\langle \rho, h \rangle \vdash x \Downarrow \rho(x) \qquad \frac{\langle \rho, h \rangle \vdash \pi \Downarrow l \quad h(l) = o}{\langle \rho, h \rangle \vdash \pi.f \Downarrow o(f)}$$

**Access path root**

$$\downarrow x = x \quad \downarrow\pi.f = \downarrow\pi$$

**Access path satisfying a policy**

We suppose given $\Pi_p : \mathit{Policy}_{\text{id}} \to \mathit{Policy}$ the set of copy policies of the considered program $p$.

$$\frac{}{\vdash x : \tau} \qquad \frac{(X_1\ f_1) \in \tau, (X_2\ f_2) \in \Pi_p(X_1), \cdots, (X_n\ f_n) \in \Pi_p(X_{n-1})}{\vdash x.f_1.\ldots.f_n : \tau}$$

**Policy semantics**

$$\frac{\left.\begin{array}{l} \forall \pi, \pi' \in \mathbb{P}, \forall l, l' \in \mathit{Loc},\ x = \downarrow\pi, \quad \downarrow\pi' \neq x, \\ \qquad\qquad \langle \rho, h \rangle \vdash \pi \Downarrow l\ , \quad \langle \rho, h \rangle \vdash \pi' \Downarrow l', \\ \qquad\qquad \vdash \pi : \tau \end{array}\right\} \text{implies } l \neq l'}{\rho, h, x \models \tau}$$

Fig. 4: Copy Policy Semantics

**Definition 2 (Secure Copy Method).** *A method $m$ is said* secure *wrt. a copy signature* $\mathsf{Copy}(X)\{\tau\}$ *if and only if for all heaps $h_1, h_2 \in \mathit{Heap}$, local environments $\rho_1, \rho_2 \in \mathit{Env}$, locally allocated locations $A_1, A_2 \in \mathcal{P}(\mathit{Loc})$, and variables $x, y \in \mathit{Var}$,*

$$(x{:=}m_{cn:X}(y), \langle \rho_1, h_1, A_1 \rangle) \rightsquigarrow \langle \rho_2, h_2, A_2 \rangle \ \textit{implies}\ \ \rho_2, h_2, x \models \tau$$

Note that because of virtual dispatch, the method executed by such a call may not be the method found in $cn$ but an overridden version of it. The security policy requires that all overriding implementations still satisfy the policy $\tau$.

**Lemma 1 (Monotonicity of Copy Policies wrt. Overriding).**

$$\tau_1 \subseteq \tau_2 \textit{ implies } \forall h, \rho, x,\ \ \rho, h, x \models \tau_2 \ \Rightarrow \rho, h, x \models \tau_1$$

*Proof.* Under these hypotheses, for all access paths $\pi$, $\vdash \pi : \tau_1$ implies $\vdash \pi : \tau_2$. Thus the result holds by definition of $\models$.

Thanks to this lemma, it is sufficient to prove that each method is secure wrt. its own copy signature to ensure that all potential overridings will be also secure.

## 3 Type and Effect System

The annotations defined in the previous section are convenient for expressing a copy policy but are not sufficiently expressive for reasoning about the data structures being copied. The static enforcement of a copy policy hence relies on a translation of policies into a graph-based structure (that we shall call types) describing parts of the environment of local variables and the heap manipulated by a program. In particular, the types can express useful alias information between variables and heap cells. In this section, we define the set of types, an approximation (sub-typing) relation $\sqsubseteq$ on types, and an inference system for assigning types to each statement and to the final result of a method.

The set of types is defined using the following symbols:

$$\begin{array}{ll} n \in \boldsymbol{N} & t \in \boldsymbol{t} = \boldsymbol{N} + \{\bot, \top_{out}, \top\} \\ \Gamma \in \mathit{Var} \to \boldsymbol{t} & \Delta \in \boldsymbol{\Delta} = \boldsymbol{N} \rightharpoonup_{\text{fin}} \mathit{Field} \to \boldsymbol{t} \\ \Theta \in \mathcal{P}(\boldsymbol{N}) & T \in \boldsymbol{T} = (\mathit{Var} \to \boldsymbol{t}) \times \boldsymbol{\Delta} \times \mathcal{P}(\boldsymbol{N}) \end{array}$$

We assume given a set $\boldsymbol{N}$ of nodes. A value can be given a *base type* $t$ in $\boldsymbol{N} + \{\bot, \top_{out}, \top\}$. A node $n$ means the value has been locally allocated. The symbol $\bot$ means that the value is equal to the null reference $\diamond$. The symbol $\top_{out}$ means that the value contains a location that cannot reach a locally allocated object. The symbol $\top$ is the specific "no-information" base type. A type is a triplet $T = (\Gamma, \Delta, \Theta) \in \boldsymbol{T}$ where

- $\Gamma$ is a typing environment that maps (local) variables to base types.
- $\Delta$ is a graph whose nodes are elements of $\boldsymbol{N}$. The edges of the graphs are labeled with field names. The successors of a node is a base type. Edges over-approximate the concrete points-to relation.
- $\Theta$ is a set of nodes that represents necessarily only one concrete cell each. Nodes in $\Theta$ are eligible to strong-update while others (weaks nodes) can only be weakly updated.

In order to link types to the heap structures they represent, we will need to state reachability predicates in the abstract domain. Therefore, the path evaluation relation is extended to types using the following inference rules:

$$\frac{}{[\Gamma, \Delta] \vdash x \Downarrow \Gamma(x)} \quad \frac{[\Gamma, \Delta] \vdash \pi \Downarrow n}{[\Gamma, \Delta] \vdash \pi.f \Downarrow \Delta[n, f]} \quad \frac{[\Gamma, \Delta] \vdash \pi \Downarrow \top}{[\Gamma, \Delta] \vdash \pi.f \Downarrow \top} \quad \frac{[\Gamma, \Delta] \vdash \pi \Downarrow \top_{out}}{[\Gamma, \Delta] \vdash \pi.f \Downarrow \top_{out}}$$

Notice both $\top_{out}$ and $\top$ are considered as sink nodes for path evaluation purposes [4].

### 3.1 From Annotation to Type

The set of all copy policies $\Pi_p \subseteq PolicyDecl$ can be translated into a graph $\Delta_p$ as described hereafter. We assume a naming process that associates to each policy name $X \in Policy_{\text{id}}$ of a program a unique node $n'_X \in \boldsymbol{N}$.

$$\Delta_p = \bigcup_{X:\{(X_1,f_1);\ldots;(X_k,f_k)\}\in\Pi_p} \left[(n'_X, f_1) \mapsto n'_{X_1}, \cdots, (n'_X, f_k) \mapsto n'_{X_k}\right]$$

Given this graph, a policy $\tau = \{(X_1, f_1); \ldots; (X_k, f_k)\}$ that is declared in a class $cl$ is translated into a triplet:

$$\Phi(\tau) = \left(n_\tau, \Delta_p \cup \left[(n_\tau, f_1) \mapsto n'_{X_1}, \cdots, (n_\tau, f_k) \mapsto n'_{X_k}\right], \{n_\tau\}\right)$$

Note that we *unfold* the possibly cyclic graph $\Delta_p$ with an extra node $n_\tau$ in order to be able to catch an alias information between this node and the result of a method, and hence declare $n_\tau$ as strong. Take for instance the type in Fig. 1: were it not for this unfolding step, the type would have consisted only in a weak node and a $\top$ node, with the variable `res` mapping directly to the former. Note also that it is not necessary to keep (and even to build) the full graph $\Delta_p$ in $\Phi(\tau)$ but only the part that is reachable from $n_\tau$.

### 3.2 Type Interpretation

The semantic interpretation of types is given in Fig. 5, in the form of a relation

$$\langle \rho, h, A\rangle \sim [\Gamma, \Delta, \Theta]$$

that states when a local allocation history $A$, a heap $h$ and an environment $\rho$ are coherent with a type $(\Gamma, \Delta, \Theta)$. The interpretation judgment amounts to checking that (i) for every path $\pi$ that leads to a value $l$ in the concrete memory and to a type $t$ in the graph, the auxiliary type interpretation $\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash v \sim t$ holds; (ii) every strong node in $\Theta$ represents a uniquely reachable value in the concrete memory. The auxiliary judgment $\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash v \sim t$ is defined by case on $t$. The null value is represented by any type. The symbol $\top$ represents any value and $\top_{out}$ those values that do not allow to reach a locally allocated location. A node $n$ represents a locally allocated memory location $l$ such that every concrete path $\pi$ that leads to $l$ in $\langle \rho, h\rangle$ leads to node $n$ in $\langle \Gamma, \Delta\rangle$.

We now establish a semantic link between policy semantics and type interpretation. We show that if the final state of a copy method can be given a type of the form $\Phi(\tau)$ then this is a secure method wrt. the policy $\tau$.

[4] The sink nodes status of $\top$ (resp. $\top_{out}$) can be understood as a way to state the following invariant enforced by our type system: when a cell points to an unspecified (resp. foreign) part of the heap, all successors of this cell are also unspecified (resp. foreign).

**Auxiliary type interpretation**

$$\frac{}{\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash \diamond \sim t} \quad \frac{}{\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash r \sim \top} \quad \frac{\text{Reach}_h(l) \cap A = \emptyset}{\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash l \sim \top_{out}}$$

$$\frac{l \in A \qquad n \in \text{dom}(\Sigma) \qquad \forall \pi,\ \langle \rho, h\rangle \vdash \pi \Downarrow l \ \Rightarrow \langle \Gamma, \Sigma\rangle \vdash \pi \Downarrow n}{\langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash l \sim n}$$

**Main type interpretation**

$$\frac{\left.\begin{array}{l}\forall \pi, \forall t, \forall l,\\ \quad [\Gamma, \Sigma] \vdash \pi \Downarrow t\\ \quad \langle \rho, h\rangle \vdash \pi \Downarrow r\end{array}\right\} \Rightarrow \langle \rho, h, A\rangle, [\Gamma, \Delta] \Vdash r \sim t \qquad \left.\begin{array}{l}\forall n \in \Theta,\ \forall \pi, \forall \pi', \forall l, \forall l',\\ \quad [\Gamma, \Sigma] \vdash \pi \Downarrow n \ \wedge\ [\Gamma, \Sigma] \vdash \pi' \Downarrow n\\ \quad \langle \rho, h\rangle \vdash \pi \Downarrow l \ \wedge\ \langle \rho, h\rangle \vdash \pi' \Downarrow l'\end{array}\right\} \Rightarrow l = l'}{\langle \rho, h, A\rangle \sim [\Gamma, \Delta, \Theta]}$$

Fig. 5: Type Interpretation

**Theorem 1.** *Let $\Phi(\tau) = (n_\tau, \Delta_\tau, \Theta_\tau)$, $\rho \in Env, A \in \mathcal{P}(Loc)$, and $x \in Var$. Assume that, for all $y \in Var$ such that $y$ is distinct from $x$, $A$ is not reachable from $\rho(y)$ in a given heap $h$,* i.e. *$Reach_h(\rho(y)) \cap A = \emptyset$. If there exists a state of the form $\langle \rho', h, A\rangle$, a return variable* res *and a local variable type $\Gamma'$ such that $\rho'(\mathsf{res}) = \rho(x)$, $\Gamma'(\mathsf{res}) = n_\tau$ and $\langle \rho', h, A\rangle \sim [\Gamma', \Delta_\tau, \Theta_\tau]$, then $\rho, h, x \models \tau$ holds.*

*Proof.* We consider two paths $\pi'$ and $\pi$ such that $x = \downarrow\pi$, $\downarrow\pi' \neq x$, $\langle \rho, h\rangle \vdash \pi' \Downarrow l$, $\vdash \pi : \tau$, $\langle \rho, h\rangle \vdash \pi \Downarrow l$ and look for a contradiction. Since $\vdash \pi : \tau$, there exists a node $n \in \Delta_\tau$ such that $[\Gamma', \Delta_\tau] \vdash \pi \Downarrow n$. Furthermore $\langle \rho', h\rangle \vdash \pi \Downarrow l$ so we can deduce that $l \in A$. Thus we obtain a contradiction with $\langle \rho, h\rangle \vdash \pi' \Downarrow l$ because any path that starts from a variable other than $x$ cannot reach the elements in $A$.

### 3.3 Sub-typing

**Value sub-typing judgment**

$$\frac{t \in \boldsymbol{t}}{\bot \leq_\sigma t} \quad \frac{t \in \boldsymbol{t} \backslash \boldsymbol{N}}{t \leq_\sigma \top} \quad \frac{}{\top_{out} \leq_\sigma \top_{out}} \quad \frac{n \in \boldsymbol{N}}{n \leq_\sigma \sigma(n)}$$

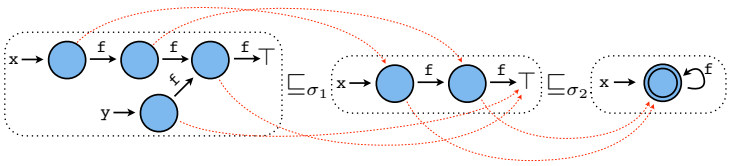
**Main sub-typing judgment**

$$\frac{\begin{array}{ll}(\text{ST}_1) & \sigma \in \boldsymbol{N}(\Delta_1) \to \boldsymbol{N}(\Delta_2) + \{\top\}\\ (\text{ST}_2) & \forall t_1 \in \boldsymbol{t}, \forall \pi \in \mathbb{P}, [\Gamma_1, \Delta_1] \vdash \pi \Downarrow t_1 \Rightarrow \exists t_2 \in \boldsymbol{t}, t_1 \leq_\sigma t_2 \wedge [\Gamma_2, \Delta_2] \vdash \pi \Downarrow t_2\\ (\text{ST}_3) & \forall n_2 \in \Theta_2,\ \exists n_1 \in \Theta_1,\ \sigma^{-1}(n_2) = \{n_1\}\end{array}}{(\Gamma_1, \Delta_1, \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)}$$

Fig. 6: Sub-typing

To manage control flow merge points we rely on a sub-typing relation $\sqsubseteq$ described in Fig. 6. A sub-type relation $(\Gamma_1, \Delta_1, \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)$ holds if and only if ($\text{ST}_1$) there exists a fusion function $\sigma$ from $\text{dom}(\Delta_1)$ to $\text{dom}(\Delta_2) + \{\top\}$. $\sigma$ is a mapping that merges nodes and edges in $\Delta_1$ such that ($\text{ST}_2$) every element $t_1$ of $\Delta_1$ accessible from a path $\pi$ is mapped to an element $t_2$ of $\Delta_2$ accessible from the same path, such that $t_1 \leq_\sigma t_2$. In particular, this means that all successors of $t_1$ are mapped to successors of $t_2$. Incidentally, because $\top$ acts as a sink on paths, if $t_1$ is mapped to $\top$, then all its successors are mapped to $\top$ too. Finally, when a strong node in $\Delta_1$ maps to a strong node in $\Delta_2$, this image node cannot be the image of any other node in $\Delta_1$—in other terms, $\sigma$ is injective on strong nodes ($\text{ST}_3$).

Intuitively, it is possible to go up in the type partial order either by merging, or by forgetting nodes in the initial graph. The following example shows three ordered types and their corresponding fusion functions. On the left, we forget the node pointed to by y and hence forget all of its successors (see ($ST_2$)). On the right we fusion two strong nodes to obtain a weak node.




The logical soundness of this sub-typing relation is formally proved with the following theorem.

**Theorem 2.** *For any type $T_1, T_2 \in \boldsymbol{T}$ and state $\langle \rho, h, A \rangle$, $T_1 \sqsubseteq T_2$ and $\langle \rho, h, A \rangle \sim [T_1]$ imply $\langle \rho, h, A \rangle \sim [T_2]$*

*Proof.* See [10] and the companion Coq development.

### 3.4 Type and Effect System

The type system verifies, statically and class by class, that a program respects the copy policy annotations relative to a declared copy policy. The core of the type system concerns the typability of commands, which is defined through the following judgment:

$$\Gamma, \Delta, \Theta \vdash c : \Gamma', \Delta', \Theta'.$$

The judgment is valid if the execution of command $c$ in a state satisfying type $(\Gamma, \Delta, \Theta)$ will result in a state satisfying $(\Gamma', \Delta', \Theta')$ or will diverge.

Typing rules are given in Fig. 7. We explain a selection of rules in the following. The rules for $\mathit{if}\ (*)\ \mathit{then}\ \ \mathit{else}\ \ \mathit{fi}$, $\mathit{while}\ (*)\ \mathit{do}\ \ \mathit{done}$, sequential composition and most of the assignment rules are standard for flow-sensitive type systems. The rule for $x := \mathit{new}$ "allocates" a fresh node $n$ with no edges in the graph $\Delta$ and let $\Gamma(x)$ reference this node.

There are two rules concerning the instruction $x.f{:=}y$ for assigning values to fields. If the variable $x$ is represented by node $n$, then either the node is strong and we update (or add) the edge in the graph $\Delta$ from node $n$ labeled $f$ to point to the value of $\Gamma(y)$, or it is only weak and we must merge the previous shape with its updated version.

As for method calls, two cases arise depending on whether the method is copy-annotated or not. In each case we must also discuss the type of the argument $y$. On the one hand, if a method is associated with a copy policy $\tau$, we compute the corresponding type $(n_\tau, \Delta_\tau)$ and type the result of $x{:=}m_{cn:X}(y)$ starting in $(\Gamma, \Delta, \Theta)$ with the result type consisting of the environment $\Gamma$ where $x$ now points to $n_\tau$, the heap described by the disjoint union of $\Delta$ and $\Delta_\tau$, and the set of strong nodes augmented with $n_\tau$. If $y$ is a locally allocated memory location of type $n$, we must remove all nodes reachable from $n$, and set all its successors to $\top$. On the other hand, the method is not associated with

**Command typing rules**

$$\frac{}{\Gamma,\Delta,\Theta \vdash x{:=}y : \Gamma[x \mapsto \Gamma(y)],\Delta,\Theta} \qquad \frac{n \text{ fresh in } \Delta}{\Gamma,\Delta,\Theta \vdash x := new\ cn : \Gamma[x \mapsto n],\Delta[(n,\_) \mapsto \bot],\Theta \cup \{n\}}$$

$$\frac{\Gamma(y) = t \qquad t \in \{\top_{out},\top\}}{[\Gamma,\Delta,\Theta] \vdash x{:=}y.f : \Gamma[x \mapsto t],\Delta,\Theta} \qquad \frac{\Gamma(y) = n}{\Gamma,\Delta,\Theta \vdash x{:=}y.f : \Gamma[x \mapsto \Delta[n,f]],\Delta,\Theta}$$

$$\frac{\Gamma(x) = n \quad n \in \Theta}{\Gamma,\Delta,\Theta \vdash x.f{:=}y : \Gamma,\Delta[n,f \mapsto \Gamma[y]],\Theta}$$

$$\frac{\Gamma(x) = n \quad n \notin \Theta \quad (\Gamma,\Delta[n,f \mapsto \Gamma[y]],\Theta) \sqsubseteq (\Gamma',\Delta',\Theta') \quad (\Gamma,\Delta,\Theta) \sqsubseteq (\Gamma',\Delta',\Theta')}{\Gamma,\Delta,\Theta \vdash x.f{:=}y : \Gamma',\Delta',\Theta'}$$

$$\frac{\begin{array}{c}\Gamma,\Delta,\Theta \vdash c_1 : \Gamma_1,\Delta_1,\Theta_1 \quad (\Gamma_1,\Delta_1,\Theta_1) \sqsubseteq (\Gamma',\Delta',\Theta') \\ \Gamma,\Delta,\Theta \vdash c_2 : \Gamma_2,\Delta_2,\Theta_2 \quad (\Gamma_2,\Delta_2,\Theta_2) \sqsubseteq (\Gamma',\Delta',\Theta')\end{array}}{\Gamma,\Delta,\Theta \vdash if\ (*)\ then\ c_1\ else\ c_2\ fi : \Gamma',\Delta',\Theta'}$$

$$\frac{\Gamma',\Delta',\Theta' \vdash c : \Gamma_0,\Delta_0,\Theta_0 \quad (\Gamma,\Delta,\Theta) \sqsubseteq (\Gamma',\Delta',\Theta') \quad (\Gamma_0,\Delta_0,\Theta_0) \sqsubseteq (\Gamma',\Delta',\Theta')}{\Gamma,\Delta,\Theta \vdash while\ (*)\ do\ c\ done : \Gamma',\Delta',\Theta'}$$

$$\frac{\Gamma,\Delta,\Theta \vdash c_1 : \Gamma_1,\Delta_1,\Theta_1 \quad \Gamma_1,\Delta_1,\Theta_1 \vdash c_2 : \Gamma_2,\Delta_2,\Theta_2}{\Gamma,\Delta,\Theta \vdash c_1;c_2 : \Gamma_2,\Delta_2,\Theta_2}$$

$$\frac{\Pi_p(X) = \tau \quad \Phi(\tau) = (n_\tau,\Delta_\tau) \quad nodes(\Delta) \cap nodes(\Delta_\tau) = \emptyset \quad (\Gamma[y] = \bot) \vee (\Gamma[y] = \top_{out})}{\Gamma,\Delta,\Theta \vdash x{:=}m_{cn:X}(y) : \Gamma[x \mapsto n_\tau],\Delta \cup \Delta_\tau,\Theta \cup \{n_\tau\}}$$

$$\frac{\begin{array}{c}\Pi_p(X) = \tau \quad \Phi(\tau) = (n_\tau,\Delta_\tau) \quad nodes(\Delta) \cap nodes(\Delta_\tau) = \emptyset \\ KillSucc_n(\Gamma,\Delta,\Theta) = (\Gamma',\Delta',\Theta') \quad \Gamma[y] = n\end{array}}{\Gamma,\Delta,\Theta \vdash x{:=}m_{cn:X}(y) : \Gamma'[x \mapsto n_\tau],\Delta' \cup \Delta_\tau,\Theta' \cup \{n_\tau\}}$$

$$\frac{(\Gamma[y] = \bot) \vee (\Gamma[y] = \top_{out})}{\Gamma,\Delta,\Theta \vdash x{:=}?(y) : \Gamma[x \mapsto \top_{out}],\Delta,\Theta} \qquad \frac{KillSucc_n(\Gamma,\Delta,\Theta) = (\Gamma',\Delta',\Theta') \quad \Gamma[y] = n}{\Gamma,\Delta,\Theta \vdash x{:=}?(y) : \Gamma'[x \mapsto \top_{out}],\Delta',\Theta'}$$

$$\Gamma,\Delta,\Theta \vdash return\ x : \Gamma[ret \mapsto \Gamma[x]],\Delta,\Theta$$

**Method typing rule**

$$\frac{\begin{array}{c}[\,\cdot \mapsto \bot][x \mapsto \top_{out}],\emptyset,\emptyset \vdash c : \Gamma,\Delta,\Theta \\ \Pi_p(X) = \tau \quad \Phi(\tau) = (n_\tau,\Delta_\tau) \quad (\Gamma,\Delta,\Theta) \sqsubseteq (\Gamma[ret \mapsto n_\tau],\Delta_\tau,\{n_\tau\})\end{array}}{\vdash \mathsf{Copy}(X)\ m(x){:=}c}$$

**Program typing rule**

$$\frac{\forall cl \in p,\ \forall md \in cl, \quad \vdash md}{\vdash p}$$

**Notations:** We write $\Delta[(n,\_) \mapsto \bot]$ for the update of $\Delta$ with a new node $n$ for which all successors are equal to $\bot$. We write $KillSucc_n$ for the function that removes all nodes reachable from $n$ (with at least one step) and sets all its successors equal to $\top$.

Fig. 7: Type System

a copy policy. If the parameter $y$ is null or not locally allocated we know that $x$ points to a non-locally allocated object. Else $y$ is a locally allocated memory location of type $n$, and we must kill all its successors in the abstract heap.

Finally, the rule for method definition verifies the coherence of the result of analysing the body of a method $m$ with its copy annotation $\Phi(\tau)$. Type checking extends trivially to all methods of the program.

Note the absence of a rule for typing an instruction $x.f{:=}y$ when $\Gamma(x) = \top$ or $\top_{out}$. In a first attempt, a sound rule would have been

$$\frac{\Gamma(x) = \top}{\Gamma,\Delta \vdash x.f{:=}y : \Gamma,\Delta[\,\cdot,f \mapsto \top]}$$

Because $x$ may point to any part of the local shape we must conservatively forget all knowledge about the field $f$. Moreover we should also warn the caller of the current

method that a field $f$ of his own local shape may have been updated. We choose to simply reject copy methods with such patterns. Such a strong policy has at least the merit to be easily understandable to the programmer: a copy method should only modify locally allocated objects to be typable in our type system. For similar reasons, we reject methods that attempt to make a method call on a reference of type $\top$ because we can not track side effect modifications of such methods without loosing the modularity of the verification mechanism.

We first establish a standard subject reduction theorem and then prove type soundness. We assume that all methods of the considered program are well-typed.

**Theorem 3 (Subject Reduction).** *Assume* $T_1 \vdash c : T_2$ *and* $\langle \rho_1, h_1, A_1 \rangle \sim [T_1]$. *If* $(c, \langle \rho_1, h_1, A_1 \rangle) \rightsquigarrow \langle \rho_2, h_2, A_2 \rangle$ *then* $\langle \rho_2, h_2, A_2 \rangle \sim [T_2]$.

**Theorem 4.** *If* $\vdash p$ *then all methods* $m$ *declared in the program* $p$ *are secure.*

For the proofs see [10] and the companion Coq development.

```
1 class LinkedList<E> implements Cloneable {
2   private @Deep Entry<E> header;
3
4   private static class Entry<E> {
5     @Shallow E element;
6     @Deep Entry<E> next;
7     @Deep Entry<E> previous;
8   }
9
10  @Copy public Object clone() {
11    LinkedList<E> clone = null;
12    clone = (LinkedList<E>) super.clone();
13    clone.header = new Entry<E>;
14    clone.header.next = clone.header;
15    clone.header.previous = clone.header;
16    Entry<E> e = this.header.next;
17    while (e != this.header) {
18      Entry<E> n = new Entry<E>;
19      n.element = e.element;
20      n.next = clone.header;
21      n.previous = clone.header.previous;
22      n.previous.next = n;
23      n.next.previous = n;
24      e = e.next;
25    }
26    return clone;
27  }
28 }
```

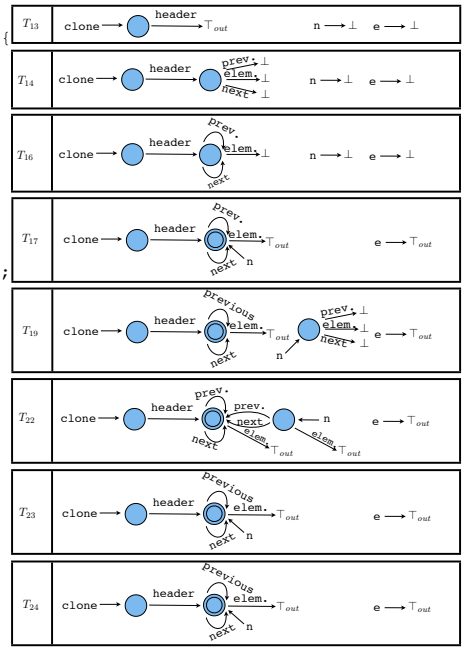



Fig. 8: Intermediate Types for `java.util.LinkedList.clone()`

*Example 2 (Case Study:* `java.util.LinkedList`*).* In this example, we demonstrate the use of the type system on a challenging example taken from the standard Java library. The class `java.util.LinkedList` provides an implementation of doubly-linked lists. A list is composed of a first cell that points through a field `header` to a collection of doubly-linked cells. Each cell has a link to the previous and the next cell and also to an element of (parameterized) type `E`. The clone method provided in `java.lang` library implements a shallow copy where only cells of type `E` may be shared between the source and the result of the copy. In Fig. 8 we present a modified version of the original source

code: we have inlined all method calls, except those to copy methods and removed exception handling that leads to an abnormal return from methods[5]. Note that one method call in the original code was virtual and hence prevented inlining. Is has been necessary to make a private version of this method. This makes sense because such a virtual call actually constitutes a potentially dangerous hook in a cloning method, as a re-defined implementation could be called when cloning a subclass of `Linkedlist`.

In Fig. 8 we provide several intermediate types that are necessary for typing this method ($T_i$ is the type before executing the instruction at line $i$). The call to `super.clone` at line 12 creates a shallow copy of the header cell of the list, which contains a reference to the original list. The original list is thus shared, a fact which is represented by an edge to $\top_{out}$ in type $T_{13}$.

The copy method then progressively constructs a deep copy of the list, by allocating a new node (see type $T_{14}$) and setting all paths `clone.header`, `clone.header.next` and `clone.header.previous` to point to this node. This is reflected in the analysis by a *strong update* to the node representing path `clone.header` to obtain the type $T_{16}$ that precisely models the alias between paths `clone.header`, `clone.header.next` and `clone.header.previous` (the Java syntax used here hides the temporary variable that is introduced to be assigned the value of `clone.header` and then be updated).

This type $T_{17}$ is the loop invariant necessary for type checking the whole loop. It is a super-type of $T_{16}$ (updated with $e \mapsto \top_{out}$) and of $T_{24}$ which represents the memory at the end of the loop body. The body of the loop allocates a new list cell (pointed to by variable `n`) (see type $T_{19}$) and inserts it into the doubly-linked list. The assignment in line 22 updates the weak node pointed to by path `n.previous` and hence merges the strong node pointed to by `n` with the weak node pointed to by `clone.header`, representing the spine of the list. The assignment at line 23 does not modify the type $T_{23}$.

Notice that the types used in this example show that a flow-insensitive version of the analysis could not have found this information. A flow-insensitive analysis would force the merge of the types at all program points, and the call to `super.clone` return a type that is less precise than the types needed for the analysis of the rest of the method.

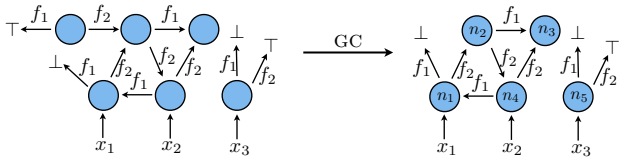
## 4 Inference

In order to type-check a method with the previous type system, it is necessary to infer intermediate types at each loop header and conditional junction points. A standard approach consists in turning the previous typing problem into a fixpoint problem in a suitable sup-semi-lattice structure. This section presents the lattice that we put on $(\boldsymbol{T}, \sqsubseteq)$. Proofs are generally omitted by lack of space but can be found in the companion report. Typability is then checked by computing a suitable least-fixpoint in this lattice. We end this section by proposing a widening operator that is necessary to prevent infinite iterations.

We write $\equiv$ for the equivalence relation defined by $T_1 \equiv T_2$ if and only if $T_1 \sqsubseteq T_2$ and $T_2 \sqsubseteq T_1$. Although this entails that $\sqsubseteq$ is a partial order structure on top of $(\boldsymbol{T}, \equiv)$,

[5] Inlining is automatically performed by our tool and exception control flow graph is managed as standard control flow but omitted here for simplicity.

equality and order testing remains difficult using only this definition. Instead of considering the quotient of $\boldsymbol{T}$ with $\equiv$, we define a notion of *well-formed* types on which $\sqsubseteq$ is antisymmetric. To do this, we assume that the set of nodes, variable names and field names are countable sets and we note $n_i$ (resp. $x_i$ and $f_i$) the $i$th node (resp. variable and field). A type $(\Gamma, \Delta, \Theta)$ is *well-formed* if every node in $\Delta$ is reachable from a node in $\Gamma$ and the nodes in $\Delta$ follow a canonical numbering based on a breadth-first traversal of the graph. Any type can be *garbage-collected* into a canonical well-formed type by removing all unreachable nodes from variables and renaming all remaining nodes using a fixed strategy based on a total ordering on variable names and field names and a breadth-first traversal. We note GC this transformation. The following example shows the effect of GC using a canonical numbering.




Since by definition, $\sqsubseteq$ only deals with reachable nodes, the GC function is a $\equiv$-morphism and respects type interpretation. This means than inference engine can at any time replace a type by a garbage-collected version. This is useful to perform an equivalence test in order to check fixpoint iteration ending.

**Lemma 2.** *For all well-formed types $T_1, T_2 \in \boldsymbol{T}$, $T_1 \equiv T_2$ iff $T_1 = T_2$.*

**Definition 3.** *Let $\sqcup$ be an operator that merges two types according to the algorithm in Fig. 9.*

```
// Initialization.
// α—nodes are sets in t.
// α—transitions can be
//    non—deterministic.
α = lift(Γ1, Γ2, Δ1 ∪ Δ2)

// Start with environments.
for {(x,t);(x,t')} ⊆ (Γ1 × Γ2) {
  fusion({t,t'})
}

// Propagate in α.
while ∃f ∈ Field, ∃u ∈ α, |succ(u,f)| > 1 {
  fusion(succ(u,f))
}

// Return to types.
(Γ,Δ,Θ) = ground(Γ1,Γ2,α)
```

```
// N is a set of t ∈ t.
// (|N|) denotes the node in α
//    labelled by the set N.
void fusion (N) {
  α ← α + (|N|)
  for t ∈ N {
    for f ∈ Field {
      if ∃u, α(t,f) = u {
        // Re—route outbound edges.
        α ← α[((|N|),f) ↦ u]
      }
      if ∃n', α(n',f) = t {
        // Re—route inbound edges.
        α ← α[(n',f) ↦ (|N|)]
    }}
  α ← α − u
}}
```

Fig. 9: Join Algorithm

The procedure has $T_1 = (\Gamma_1, \Delta_1, \Theta_1)$ and $T_2 = (\Gamma_2, \Delta_2, \Theta_2)$ as input, then takes the following steps.

1. It first makes the disjunct union of $\Delta_1$ and $\Delta_2$ into a non-deterministic graph (NDG) $\alpha$, where nodes are labelled by sets of elements in $\boldsymbol{t}$. This operation is performed by the `lift` function, that maps nodes to singleton nodes, and fields to transitions.

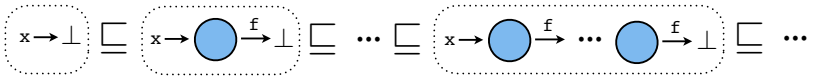
2. It joins together the nodes in $\alpha$ referenced by $\Gamma_i$ using the `fusion` algorithm[6].
3. Then it scans the NDG and merges all nondeterministic successors of nodes.
4. Finally it uses the `ground` function to recreate a graph $\Delta$ from the now-*deterministic* graph $\alpha$. This function operates by pushing a node set to a node labelled by the $\leq_\sigma$-sup of the set. The result environment $\Gamma$ is derived from $\Gamma_i$ and $\alpha$ before the $\Delta$-reconstruction.

All state fusions are recorded in a map $\sigma$ which binds nodes in $\Delta_1 \cup \Delta_2$ to nodes in $\Delta$.

**Theorem 5.** *The operator $\sqcup$ defines a sup-semi-lattice on types.*

*Proof.* See [10].

The poset structure does not enjoy the ascending chain condition. The following chain is an example infinite ascending chain.




We have then to rely on a widening [7] operator to enforce termination of fixpoint computation. Here we follow a very pragmatic approach and define a widening operator $\nabla \in \boldsymbol{T} \times \boldsymbol{T} \to \boldsymbol{T}$ that takes the result of $\sqcup$ and that collapses together (with the operator `fusion` defined above) any node $n$ and its predecessors such that the minimal path reaching $n$ and starting from a local variable is of length at least 2.

## 5 Experiments

The policy language and its enforcement mechanism has been implemented in the form of a security tool for Java byte code. Standard Java `@interface` declarations are used to specify native annotations, which enable development environments such as Eclipse or Netbeans to parse, identify and auto-complete `@Shallow`, `@Deep`, and `@Copy` tags. Source code annotations are being made accessible to bytecode analysis frameworks. Both the policy extraction and enforcement components are implemented using the Javalib/Sawja static analysis libraries[7] to derive annotations and intermediate code representations.

In its standard mode, the tool performs a modular verification of annotated classes. We have run experiments on several classes of the standard library (specially in the package `java.util`) and have successfully checked realistic copy signatures for them (see the companion web page for examples). These experiments have also confirmed that the policy enforcement mechanism facilitates re-engineering into more compact implementations of cloning methods in classes with complex dependencies, such as those forming the `gnu.xml.transform` package. For example, in the `Stylesheet` class

[6] Remark that $\Gamma_i$-bindings are not represented in $\alpha$, but that node set fusions are trivially traceable. This allows us to safely ignore $\Gamma_i$ during the following step and still perform a correct graph reconstruction.

[7] `http://sawja.inria.fr`

an inlined implementation of multiple deep copy methods for half a dozen fields can be rewritten to dispatch these functionalities to the relevant classes, while retaining the expected copy policy. This is made possible by the modularity of our enforcement mechanism, which validates calls to external cloning methods as long as their respective policies have been verified. However, some cloning methods will necessarily be beyond the reach of the analysis. We have identified one such method in GNU Classpath's `TreeMap` class, where the merging of information at control flow merge points destroys too much of the inferred type graph. A disjunctive form of abstraction seems necessary to verify a deep copy annotation on such programs and we leave this as a challenging extension.

The analysis is also capable of processing un-annotated methods, albeit with less precision than when copy policies are available—this is because it cannot rely on annotations to infer external copy method types. Nevertheless, this capability allows us to test our tool on two large code bases. The 17000 classes in Sun's `rt.jar` and the 7000 in the GNU Classpath have passed our scanner un-annotated. Among the 459 `clone`() methods we found in these classes, only 15 have been rejected because of an illegal assignment or method call and we were unable to infer the minimal signatures $\{\}$ (the same signature as `java.lang.Object.clone`()) in 78 methods. Our prototype confirms the efficiency of the enforcement technique because all these verifications took only 25s on a laptop computer.

Our prototype, the Coq formalization and proofs, as well as examples of annotated classes can be found at `http://www.irisa.fr/celtique/ext/clones`.

## 6 Related Work

Several proposals for programmer-oriented annotations of Java programs have been published following Bloch's initial proposal of an annotation framework for the Java language [4]. These proposals define the syntax of the annotations but often leave their exact semantics unspecified. A notable exception is the set of annotations concerning non-null annotations [8] for which a precise semantic characterization has emerged [9]. Concerning security, the GlassFish environment in Java offers program annotations of members of a class (such as `@DenyAll` or `@RolesAllowed`) for implementing role-based access control to methods.

To the best of our knowledge, the current paper is the first to propose a formal, semantically founded framework for secure cloning through program annotation and static enforcement. The closest work in this area is that of Anderson *et al.* [2] who have designed an annotation system for C data structures in order to control sharing between threads. Annotation policies are enforced by a mix of static and run-time verification. On the run-time verification side, their approach requires an operator that can dynamically "cast" a cell to an unshared structure. In contrast, our approach offers a completely static mechanism with statically guaranteed alias properties.

Aiken *et al.* proposes an analysis for checking and inferring local non-aliasing of data [1]. They propose to annotate C function parameters with the keyword `restrict` to ensure that no other aliases to the data referenced by the parameter are used during the execution of the method. A type and effect system is defined for enforcing this discipline statically. This analysis differs from ours in that it allows aliases to exist as long as

they are not used whereas we aim at providing guarantees that certain parts of memory are without aliases. The properties tracked by our type system are close to escape analysis [3, 6] but the analyses differ in their purpose. While escape analysis tracks locally allocated objects and tries to detect those that do not escape after the end of a method execution, we are specifically interested in tracking locally allocated objects that escape from the result of a method, as well as analyse their dependencies with respect to parameters.

Our static enforcement technique falls within the large area of static verification of heap properties. A substantial amount of research has been conducted here, the most prominent being region calculus [14], separation logic [11] and shape analysis [12]. Of these three approaches, shape analysis comes closest in its use of shape graphs. Shape analysis is a large framework that allows to infer complex properties on heap allocated data-structures like absence of dangling pointers in C or non-cyclicity invariants. In this approach, heap cells are abstracted by shape graphs with flexible object abstractions. Graph nodes can either represent a single cell, hence allowing strong updates, or several cells (summary nodes). *Materialization* allows to split a summary node during cell access in order to obtain a node pointing to a single cell. The shape graphs that we use are not intended to do full shape analysis but are rather specialized for tracking sharing in locally allocated objects. We use a different naming strategy for graph nodes and discard all information concerning non-locally allocated references. This leads to an analysis which is more scalable than full shape analysis, yet still powerful enough for verifying complex copy policies as demonstrated in the concrete case study `java.util.LinkedList`.

## 7 Conclusions and Perspectives

Cloning of objects is an important aspect of exchanging data with untrusted code. Current language technology for cloning does not provide adequate means for defining and enforcing a secure copy policy statically; a task which is made more difficult by important object-oriented features such as inheritance and re-definition of cloning methods. We have presented a flow-sensitive type system for statically enforcing copy policies defined by the software developer through simple program annotations. The annotation formalism is compatible with the inheritance-based object oriented programing language and deals with dynamic method dispatch. The verification technique is designed to enable modular verification of individual classes, in order to provide a framework that can form part of an extended, security-enhancing Java byte code verifier. By specifically targeting the verification of copy methods, we consider a problem for which it is possible to deploy a localized version of shape analysis that avoids the complexity of a full shape analysis framework.

The present paper constitutes the formal foundations for a secure cloning framework. All theorems except those of Section 4 have been mechanized in the Coq proof assistant. Mechanization has been of great help to get right the soundness arguments but has been made particularly challenging because of the storeless nature of our type interpretation.

Several issues merit further investigations in order to develop a full-fledged software security verification tool. In the current approach, virtual methods without copy policy annotations are considered as black boxes that may modify any object reachable from its arguments. An extension of our copy annotations to virtual calls should be worked out if we want to enhance our enforcement technique and accept more secure copying methods. More advanced verifications will be possible if we develop a richer form of type signatures for methods where the formal parameters may occur in copy policies, in order to express a relation between copy properties of returning objects and parameter fields. The challenge here is to provide sufficiently expressive signatures which at the same time remain humanly readable software contracts. The current formalisation has been developed for a sequential model of Java. We believe that the extension to interleaving multi-threading semantics would be feasible without major changes to the type system because we only manipulate thread-local pointers. Spelling out the formal details of this argument is left for further work.

## References


1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proc. of PLDI '03*, pages 129–140. ACM Press, 2003.
2. Z. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proc. of PLDI'09*, pages 98–109. ACM Press, 2009.
3. B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proc. of OOPSLA*, pages 20–34. ACM Press, 1999.
4. J. Bloch. *JSR 175: A metadata facility for the Java programming language*. `http://jcp.org/en/jsr/detail?id=175`, September 30, 2004.
5. CERT. *The CERT Sun Microsystems Secure Coding Standard for Java*, 2010. `https://www.securecoding.cert.org`.
6. J.D. Choi, M. G., M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Proc. of OOPSLA*, pages 1–19. ACM Press, 1999.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.
8. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. of OOPSLA'03*, pages 302–312. ACM Press, 2003.
9. L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 132–149. Springer Berlin, 2008.
10. T. Jensen, F. Kirchner, and D. Pichardie. Secure the clones, 2010. Extended version, available at `http://www.irisa.fr/celtique/ext/clones`.
11. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL'04*, pages 268–280. ACM Press, 2004.
12. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
13. Sun Develper Network. *Secure Coding Guidelines for the Java Programming Language, version 3.0*, 2010. `http://java.sun.com/security/seccodeguide.html`.
14. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.