SECURE THE CLONES

THOMAS JENSEN, FLORENT KIRCHNER, AND DAVID PICHARDIE

INRIA Rennes – Bretagne Atlantique, France
e-mail address: 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 fulfil 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 [17] and CERT [6] 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.

1998 ACM Subject Classification: I.1.2 Algorithms—Analysis of algorithms, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.3 Studies of Program Constructs—Type structure, D.3.3 Language Constructs and Features—Classes and objects.

Key words and phrases: Static analysis, Shape analysis, Type system, Java bytecode, Secure data copying.

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

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

(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:

```
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. Intuitively, policies impose non-sharing constraints between the structure referenced by a field of an object and the structure returned by the cloning method. Notice, that we do not enforce that a copy method will always return a target object that is functionally equivalent to its source. Nor does our method prevent a sub-class from making a copy of a structure using new fields that are not governed by the declared policy. For a more detailed example of these limitations, see Section 2.3.

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 behaviour 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*) upon

[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.

return from `clone`(), the object referenced by this field `f` is not 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:

```
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 unreachable 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 default 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.

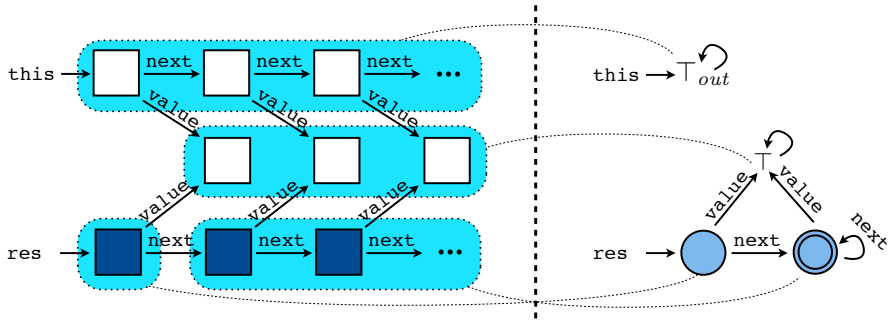


FIGURE 1. A linked structure (left part) and its abstraction (right part).

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 and not shared. 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.

This article is an extended version of a paper presented at ESOP'11 [13]. We have taken advantage of the extra space to provide improved and more detailed explanations, in particular of the inference mechanism and of what is exactly is being enforced by our copy policies. We have also added details of the proof of correctness of the enforcement mechanism. The formalism of copy policies and the correctness theorem for the core language defined in Section 2 have been implemented and verified mechanically in Coq [1]. The added details about the proofs should especially facilitate the understanding of this Coq development

2. Language and Copy Policies

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

$$x, y \in \mathit{Var} \qquad f \in \mathit{Field} \qquad m \in \mathit{Meth} \qquad cn \in \mathit{Class}_{\mathrm{id}} \qquad X \in \mathit{Policy}_{\mathrm{id}}$$

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

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.

FIGURE 2. Language Syntax.

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 $\mathit{Policy}_{\mathrm{id}}$, ranged over by X. A copy policy declaration has the form $X : \{\tau\}$ where X is the identifier of the policy signature and τ is a policy. The policy τ consists of a set of field annotations (X, f) ; ... 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 declaration of the class `List` presented in Sec. 1.2 writes: $\texttt{List.default} : \{(\texttt{List.default}, \texttt{next})\}$.

We assume that for a given program, all copy policies have been grouped together in a finite map $\Pi_p : \mathit{Policy}_{\mathrm{id}} \to \mathit{Policy}$. 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:

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

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

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

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

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

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

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

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

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

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

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 ρ 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.

FIGURE 3. Semantic Rules.

A program state consists of an environment ρ of local variables, a store h of locations mapping[3] to objects in a heap and a set A of *locally allocated locations*, *i.e.*, the locations that have been allocated by the current method invocation or by one of its callees. This last component does not influence the semantic transitions: it is used to express the type system interpretation defined in Sec. 3, but is not used in the final soundness theorem. Each object is modeled in turn as a pair composed with its dynamic class and 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 $Comm \times State$ and resulting states $State$. The set of 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 $\rho(y)$. It assigns the variable x a reference

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

that was either reachable from $\rho(y)$ 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 2.1 (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

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

Intuitively, this definition imposes that the overriding copy policy is stronger than the policy that it overrides. Lemma 2.4 below states this formally.

Example 2.2. 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 $\{(X, \mathtt{g})\}$. If a class `B` extends `A` and overrides `clone`(), it must assign it a policy of the form $\{(X, \mathtt{g});\ \ldots\ \}$ 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 π can be evaluated to a value v in a context $\langle \rho, h\rangle$ with a judgement $\langle \rho, h\rangle \vdash \pi \Downarrow v$. Each path π has a root variable $\downarrow\pi \in \mathit{Var}$. A judgement $\vdash \pi : \tau$ holds when a path π follows only deep fields in the policy τ. The rule defining the semantics of copy policies can be paraphrased as follows: For any path π starting in x and leading to location l only following deep fields in policy τ, there cannot be another path leading to the same location l which does not start in x.

Definition 2.3 (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 $x, y \in \mathit{Var}$,

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

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 τ.

Access path syntax

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

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

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

Access path satisfying a policy

We suppose given $\Pi_p : \mathit{Policy}_{\mathrm{id}} \to \mathit{Policy}$ 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

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

FIGURE 4. Copy Policy Semantics

Lemma 2.4 (Monotonicity of Copy Policies wrt. Overriding)**.**

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

Proof. [See Coq proof `Overriding.copy_policy_monotony` [1]]

Under these hypotheses, for all access paths π, $\vdash \pi : \tau_1$ 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 wrt. that copy signature.

2.3. **Limitations of Copy Policies.** The enforcement of our copy policies will ensure that certain sharing constraints are satisfied between fields of an object and its clone. However, in the current formalism we restrict the policy to talk about fields that are actually present in a class. The policy does not ensure properties about fields that are added in sub-classes. This means that an attacker could copy *e.g.*, a list by using a new field to build the list, as in the following example.

```
public class EvilList<V> extends List<V>
{
    @Shallow public List<V> evilNext;

    public EvilList(V val, List<V> next) {
        super(val,null);
        this.evilNext = next; }

    public List<V> clone() {
        return new EvilList(value,evilNext); }

   // redefinition of all other methods to use the evilNext field
   // instead of next
}
```

The enforcement mechanism described in this article will determine that the `clone`() method of class `EvilList` respects the copy policy declared for the `List` class in Section 1.2 because this policy only speaks about the `next` field which is set to `null`. It will fail to discover that the class `EvilList` creates a shallow copy of lists through the `evilNext` field. In order to prevent this attack, the policy language must be extended, *e.g.*, by adding a facility for specifying that all fields *except* certain, specifically named fields must be copied deeply. The enforcement of such policies will likely be able to reuse the analysis technique described below.

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 and is not shared. 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. As is standard in analysis of memory structures, we distinguish between nodes that represent exactly one memory cell and nodes that may represent several cells. If a node representing only one cell has an edge to another node, then this edge can be forgotten and *replaced* when we assign a new value to the node—this is called a *strong* update. If the node represents several cells, then the assignment may not concern all these cells and edges cannot be forgotten. We can only *add* extra out-going edges to the node—this is termed a *weak* update. In the graphical representations of types, we use singly-circled nodes to designate "weak" nodes and doubly-circled nodes to represent "strong" nodes.

A type is a triplet $T = (\Gamma, \Delta, \Theta) \in \boldsymbol{T}$ where

- **Γ:** is a typing environment that maps (local) variables to base types.
- **Δ:** 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.
- **Θ:** is a set of nodes that represents necessarily only one concrete cell each. Nodes in Θ are eligible to strong update while others (weak nodes) can only be weakly updated.

Example 3.1. The default `List` policy of Sec. 1.2 translates into the type

$$\Gamma = [\texttt{res} \mapsto n_1, \texttt{this} \mapsto \top_{out}]$$
$$\Delta = [(n_1, \texttt{next}) \mapsto n_2, (n_2, \texttt{next}) \mapsto n_2, (n_1, \texttt{value}) \mapsto \top, (n_2, \texttt{value}) \mapsto \top]$$
$$\Theta = \{n_1\}.$$

As mentioned in Sec 1.3, this type enjoys a graphic representation corresponding to the right-hand side of Fig. 1.

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 $\Pi_p \subseteq \mathit{PolicyDecl}$ can be translated into a graph Δ_p as described hereafter. We assume a naming process that associates to each policy name $X \in \mathit{Policy}_{\text{id}}$ 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 Δ_p with an extra node n_τ in order to be able to catch an alias information between this node and the result of a method, and hence declare n_τ 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 Δ_p in $\Phi(\tau)$ but only the part that is reachable from n_τ.

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 ρ are coherent with a type (Γ, Δ, Θ). The interpretation judgement amounts to checking that (i) for every path π that leads to a value v in the concrete memory and to a base type t in the graph, t is a correct description of v, as formalized by the auxiliary type interpretation $\langle \rho, h, A \rangle, (\Gamma, \Delta) \Vdash v \sim t$; (ii) every strong node in Θ represents a uniquely reachable value in the concrete memory. The auxiliary judgement $\langle \rho, h, A \rangle, (\Gamma, \Delta) \Vdash v \sim t$ 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

[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

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

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

Main type interpretation

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

FIGURE 5. Type Interpretation

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

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

FIGURE 6. Sub-typing

represents a locally allocated memory location l such that every concrete path π that leads to l in $\langle \rho, h\rangle$ leads to node n in (Γ, Δ).

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 τ.

Theorem 3.2. *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 Γ' 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.*

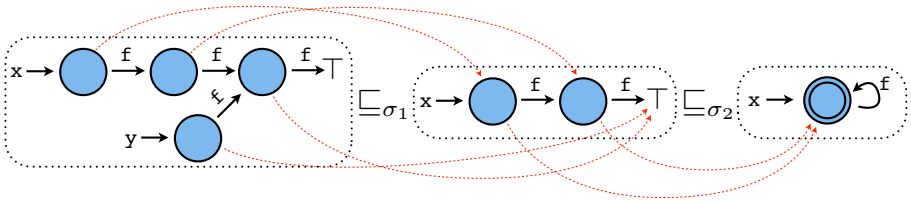
Proof. [See Coq proof `InterpAnnot.sound_annotation_to_type` [1]]

We consider two paths π' and $x.\pi$ such that $\downarrow\pi' \neq x$, $\langle \rho, h\rangle \vdash \pi' \Downarrow l$, $\vdash x.\pi : \tau$, $\langle \rho, h\rangle \vdash x.\pi \Downarrow l$ and look for a contradiction. Since $\vdash x.\pi : \tau$ and $\Gamma'(\mathsf{res}) = n_\tau$, there exists a node $n \in \Delta_\tau$ such that $(\Gamma', \Delta_\tau) \vdash \mathsf{res}.\pi \Downarrow n$. Furthermore $\langle \rho', h\rangle \vdash \mathsf{res}.\pi \Downarrow l$ 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.** 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 (ST_1) there exists a fusion function σ from $\mathrm{dom}(\Delta_1)$ to $\mathrm{dom}(\Delta_2) + \{\top\}$. σ is a mapping that

merges nodes and edges in Δ_1 such that (ST_2) every element t_1 of Δ_1 accessible from a path π is mapped to an element t_2 of Δ_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 Δ_1 maps to a strong node in Δ_2, this image node cannot be the image of any other node in Δ_1—in other terms, σ is injective on strong nodes (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 two intermediate lemmas. The first one states that paths are preserved between subtypes, and that they evaluate into basetypes that are related by the subtyping function.

Lemma 3.3 (Pathing in subtypes)**.** *Assume $\langle \rho, h, A\rangle \sim (T_1)$, and let σ be the fusion map defined by the assertion $T_1 \sqsubseteq T_2$. For any π, r such that $\langle \rho, h\rangle \vdash \pi \Downarrow r$, for any t_2 such that $(\Gamma_2, \Delta_2) \vdash \pi \Downarrow t_2$:*

$$\exists t_1 \leq_\sigma t_2, \quad (\Gamma_1, \Delta_1) \vdash \pi \Downarrow t_1.$$

Proof. [See Coq proof `Misc.Access_Path_Eval_subtyp` [1]]

The proof follows directly from the definition of type interpretation and subtyping. □

The second lemma gives a local view on logical soundness of subtyping.

Lemma 3.4 (Local logical soundness of subtyping)**.** *Assume $(\Gamma_1, \Delta_1, \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)$, and let v be a value and t_1, t_2 some types.*

$$\langle \rho, h, A\rangle, (\Gamma_1, \Delta_1) \Vdash v \sim t_1 \text{ and } t_1 \sqsubseteq_\sigma t_2 \text{ implies } \langle \rho, h, A\rangle, (\Gamma_2, \Delta_2) \Vdash v \sim t_2.$$

Proof. [See Coq proof `Misc.Interp_monotone` [1]]

We make a case for each rules of $t_1 \sqsubseteq_\sigma t_2$. The only non-trivial case is for $v = l \in Loc$, $t_1 = n \in \boldsymbol{N}$ and $t_2 = \sigma(n) \in \mathrm{dom}(\Delta_2)$. In this case we have to prove $\forall \pi,\ \langle \rho, h\rangle \vdash \pi \Downarrow l \Rightarrow (\Gamma_2, \Delta_2) \vdash \pi \Downarrow \sigma(n)$. Given such a path π, the hypothesis $\langle \rho, h, A\rangle, (\Gamma_1, \Delta_1) \Vdash l \sim n$ gives us $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow n$. Then subtyping hypothesis ST_2 gives us a base type t_2' such that $n \sqsubseteq_\sigma t_2'$ and $(\Gamma_2, \Delta_2) \vdash \pi \Downarrow t_2'$. But necessarily $t_2 = t_2'$ so we are done. □

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

Theorem 3.5. *For any type $T_1, T_2 \in \boldsymbol{T}$ and $\langle \rho, h, A\rangle \in State$, $T_1 \sqsubseteq T_2$ and $\langle \rho, h, A\rangle \sim (T_1)$ imply $\langle \rho, h, A\rangle \sim (T_2)$.*

Proof. [See Coq proof `InterpMonotony.Interpretation_monotone` [1]]

We suppose T_1 is of the form $(\Gamma_1, \Delta_1, \Theta_1)$ and T_2 of the form $(\Gamma_2, \Delta_2, \Theta_2)$. From the definition of the main type interpretation (Fig 5), we reduce the proof to proving the following two subgoals.

First, given a path π, a base type t_2 and a value v such that $\langle \Gamma_2, \Delta_2 \rangle \vdash \pi \Downarrow t_2$ and $\langle \rho, h \rangle \vdash \pi \Downarrow v$, we must prove that $\langle \rho, h, A \rangle, (\Gamma_2, \Delta_2) \Vdash v \sim t_2$ holds. Since $(\Gamma_1, \Delta_1, \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)$, there exists, by Lemma 3.3, a base type t_1 such that $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow t_1$ and $t_1 \sqsubseteq_\sigma t_2$. Since $\langle \rho, h, A \rangle \sim (T_1)$ holds we can argue that $\langle \rho, h, A \rangle, (\Gamma_1, \Delta_1) \Vdash v \sim t_1$ holds too and conclude with Lemma 3.4.

Second, given a strong node $n_2 \in \Theta_2$, two paths π and π' and two locations l and l' such that $(\Gamma_2, \Delta_2) \vdash \pi \Downarrow n_2$, $(\Gamma_2, \Delta_2) \vdash \pi' \Downarrow n_2$, $\langle \rho, h \rangle \vdash \pi \Downarrow l$ and $\langle \rho, h \rangle \vdash \pi \Downarrow l'$, we must prove that $l = l'$. As previously, there exists by Lemma 3.3, t_1 and t_1' such that $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow t_1$, $t_1 \sqsubseteq_\sigma n_2$, $(\Gamma_1, \Delta_1) \vdash \pi' \Downarrow t_1'$ and $t_1' \sqsubseteq_\sigma n_2$. But then, by (ST_3), there exists some strong node n_1 such that $t_1 = t_1' = n_1$ and we can obtain the desired equality from the hypothesis $\langle \rho, h, A \rangle \sim (\Gamma_1, \Delta_1, \Theta_1)$. □

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 (Γ, Δ, Θ) 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 below. The rules for *if* $(*)$ *then else fi*, *while* $(*)$ *do done*, sequential composition and most of the assignment rules are standard for flow-sensitive type systems. The rule for $x := new$ "allocates" a fresh node n with no edges in the graph Δ and let $\Gamma(x)$ references this node.

There are two rules concerning the instruction $x.f{:=}y$ for assigning values to fields. Assume that the variable x is represented by node n (*i.e.*, $\Gamma(x) = n$). In the first case (strong update), the node is strong and we update destructively (or add) the edge in the graph Δ from node n labeled f to point to the value of $\Gamma(y)$. The previous edge (if any) is lost because $n \in \Theta$ ensures that all concrete cells represented by n are affected by this field update. In the second case (weak update), the node is weak. In order to be conservative, we must merge the previous shape with its updated version since the content of $x.f$ is updated but an other cell mays exist and be represented by n without being affected by this field update.

As for method calls $m(y)$, two cases arise depending on whether the method m is copy-annotated or not. In each case, we also reason differently depending on the type of the argument y. If a method is associated with a copy policy τ, we compute the corresponding type (n_τ, Δ_τ) and type the result of $x{:=}m_{cn:X}(y)$ starting in (Γ, Δ, Θ) with the result type consisting of the environment Γ where x now points to n_τ, the heap described by the disjoint union of Δ and Δ_τ. In addition, the set of strong nodes is augmented with n_τ since a copy method is guaranteed to return a freshly allocated node. The method call may potentially modify the memory referenced by its argument y, but the analysis has no means of tracking this. Accordingly, if y is a locally allocated memory location of type n, we must remove all nodes reachable from n, and set all the successors of n to $\top$. The other case to consider is when the method is not associated with a copy policy (written $x{:=}?(y)$). If the parameter

Command typing rules

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\Gamma,\Delta,\Theta \vdash \mathit{return}\ x : \Gamma[\mathit{ret} \mapsto \Gamma(x)],\Delta,\Theta}$$

Method typing rule

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

Program typing rule

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

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

FIGURE 7. Type System

y is null or not locally allocated, then there is no way for the method call to access locally allocated memory and we know that x points to a non-locally allocated object. Otherwise, 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

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

```
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 }
```

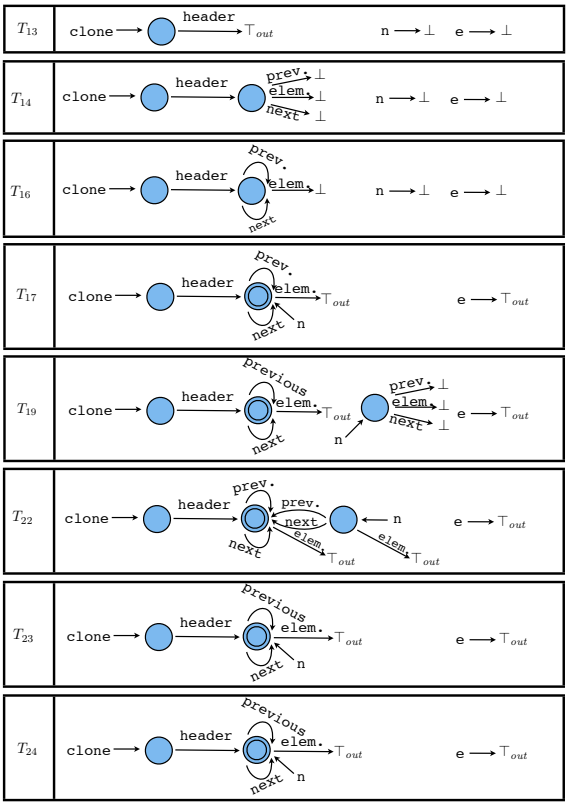


FIGURE 8. Intermediate Types for `java.util.LinkedList.clone()`

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 policy is strong but has 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 losing the modularity of the verification mechanism.

Example 3.6 (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 companion web page provides a more detailed explanation of this example [1]. 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 "semi-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 the method[5]. Note that there was one method call in the original code that was virtual and hence prevented inlining. It 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.

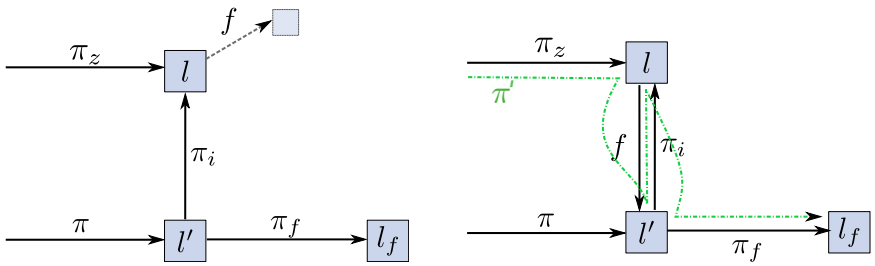
3.5. **Type soundness.** The rest of this section is devoted to the soundness proof of the type system. We consider the types $T = (\Gamma, \Delta, \Theta), T_1 = (\Gamma_1, \Delta_1, \Theta_1), T_2 = (\Gamma_2, \Delta_2, \Theta_2) \in \boldsymbol{T}$, a program $c \in \mathit{Prog}$, as well as the configurations $\langle \rho, h, A\rangle, \langle \rho_1, h_1, A_1\rangle, \langle \rho_2, h_2, A_2\rangle \in \mathit{State}$.

Assignments that modify the heap space can also modify the reachability properties of locations. This following lemma indicates how to reconstruct a path to l_f in the *initial* heap from a path π' to a given location l_f in the *assigned* heap.

Lemma 3.7 (Path decomposition on assigned states)**.** *Assume given a path π, field f, and locations l, l' such that $\langle \rho, h\rangle \vdash \pi \Downarrow l'$. and assume that for any path π' and a location l_f we have $\langle \rho, h[l, f \mapsto l']\rangle \vdash \pi' \Downarrow l_f$. Then, either*

$$\langle \rho, h\rangle \vdash \pi' \Downarrow l_f$$

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



(A) in $\langle \rho, h \rangle$ (B) in $\langle \rho, h[l, f \mapsto l'] \rangle$

FIGURE 9. An Illustration of Path Decomposition on Assigned States.

or $\exists \pi_z, \pi_1, \ldots, \pi_n, \pi_f$ *such that:*

$$\begin{cases} \pi' = \pi_z.f.\pi_1.f.\ldots.f.\pi_n.f.\pi_f \\ \langle \rho, h \rangle \vdash \pi_z \Downarrow l \\ \langle \rho, h \rangle \vdash \pi.\pi_f \Downarrow l_f \\ \forall \pi_1, \ldots, \pi_n, \langle \rho, h \rangle \vdash \pi.\pi_i \Downarrow l \end{cases}$$

The second case of the conclusion of this lemma is illustrated in Fig. 9.

Proof. [See Coq proof `Misc.Access_Path_Eval_putfield_case` [1]]
The proof is done by induction on π. □

We extend the previous lemma to paths in both the concrete heap and the graph types.

Lemma 3.8 (Pathing through strong field assignment)**.** *Assume* $\langle \rho, h, A \rangle \sim (\Gamma, \Delta, \Theta)$ *with* $\rho(x) = l_x \in A$, $\Gamma(x) = n_x \in \Theta$, $\rho(y) = l_y$, *and* $\Gamma(y) = t_y$. *Additionally, suppose that for some path* π, *value* v, *and type* t*:*

$$\langle \rho, h[l_x, f \mapsto l_y] \rangle \vdash \pi \Downarrow v$$
$$(\Gamma, \Delta[n_x, f \mapsto t_y]) \vdash \pi \Downarrow t.$$

Then at least one of the following four statements hold:

$$(\langle \rho, h \rangle \vdash \pi \Downarrow v \wedge (\Gamma, \Delta) \vdash \pi \Downarrow t) \tag{1}$$
$$\left(\exists \pi', \langle \rho, h \rangle \vdash y.\pi' \Downarrow v \wedge (\Gamma, \Delta) \vdash y.\pi' \Downarrow t\right) \tag{2}$$
$$t = \top \tag{3}$$
$$v = \diamond \tag{4}$$

Proof. [See Coq proof `Misc.strong_subst_prop` [1]]
The non-trivial part of the lemma concerns the situation when $t \neq \top$ and $v \neq \diamond$). In that case, the proof relies on Lemma 3.7. The two parts of the disjunction in Lemma 3.7 are used to prove one of the two first statements. If the first part of the disjunction holds, we can assume that $\langle \rho, h \rangle \vdash \pi \Downarrow v$. Then, since $\langle \rho, h, A \rangle \sim (\Gamma, \Delta, \Theta)$, we also have $(\Gamma, \Delta) \vdash \pi \Downarrow t$. This implies the first main statement. If the second part of the disjunction holds, then, by observing that $\langle \rho, h \rangle \vdash y \Downarrow l_y$, we can derive the sub-statement $\exists \pi_f, \langle \rho, h \rangle \vdash y.\pi_f \Downarrow v$. As previously, by assumption we also have $(\Gamma, \Delta) \vdash y.\pi_f \Downarrow t$, which implies our second main statement. □

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 3.9 (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* $\langle \rho_2, h_2, A_2 \rangle \sim T_2$*.*

Proof. [See Coq proof `Soundness.subject_reduction` [1]]

The proof proceeds by structural induction on the instruction c. For each reduction rule concerning c (Fig. 3), we prove that the resulting state is in relation to the type T_2, as defined by the main type interpretation rule in Fig. 5. This amounts to verifying that the two premises of the type interpretation rule are satisfied. One premise checks that all access paths lead to related (value,node) pairs. The other checks that all nodes that are designated as "strong" in the type interpretation indeed only represent unique locations.

We here present the most intricate part of the proof, which concerns graph nodes different from $\top$, $\bot$, or $\top_{out}$, and focus here on variable and field assignment. The entire proof has been checked using the Coq proof management system.

If $c \equiv x{:=}y$ then $\langle \rho_2, h_2, A_2 \rangle = \langle \rho_1[x \mapsto \rho_1(y)], h_1, A_1 \rangle$. Take $x.\pi$ a path in $(\rho_2, h_2, \Gamma_2, \Delta_2)$: since $\rho_2(x) = \rho_1(y)$ and $\Gamma_2(x) = \Gamma_1(y)$, $y.\pi$ is also a path in $(\rho_1, h_1, \Gamma_1, \Delta_1)$. Given that $\langle \rho_1, h_1, A_1 \rangle \sim (\Gamma_1, \Delta_1, \Theta_1)$, we know that $y.\pi$ will lead to a value v (formally, $\langle \rho_1, h_1 \rangle \vdash y.\pi \Downarrow v$) and a node n (formally, $(\Gamma_1, \Delta_1) \vdash y.\pi \Downarrow n$) such that $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash v \sim n$. The two paths being identical save for their prefix, this property also holds for $x.\pi$ (formally, $\langle \rho_1[x \mapsto \rho_1(y)], h_1, A_1 \rangle, (\Gamma_1[x \mapsto \Gamma_1(y)], \Delta_1) \Vdash v \sim n$, $\langle \rho_1[x \mapsto \rho_1(y)], h_1 \rangle \vdash x.\pi \Downarrow v$, and $(\Gamma_1[x \mapsto \Gamma_1(y)], \Delta_1) \vdash x.\pi \Downarrow n$). Paths not beginning with x are not affected by the assignment, and so we can conclude that the first premise is satisfied.

For the second premise, let $n \in \Theta_2$ and assume that n can be reached by two paths π and π' in Δ_2. If none or both of the paths begin with x then, by assumption, the two paths will lead to the same location in $h_2 = h_1$. Otherwise, suppose that, say, π begins with x and π' with a different variable z and that they lead to l and l' respectively. Since $\Gamma_2(x) = \Gamma_1(y)$ and $\Delta_2 = \Delta_1$, then by assumption there is a path π'' in $(\rho_1, h_1, \Delta_1, \Gamma_1)$ that starts with y, and such that $\langle \rho_1, h_1 \rangle \vdash \pi'' \Downarrow l$. As z is not affected by the assignment, we also have that $\langle \rho_1, h_1 \rangle \vdash \pi' \Downarrow l'$. Therefore, as $n \in \Theta_1$ and $\langle \rho_1, h_1, A_1 \rangle \sim (\Gamma_1, \Delta_1, \Theta_1)$, we can conclude that $l = l'$. This proves that $\langle \rho_1[x \mapsto \rho_1(y)], h_1, A_1 \rangle \sim (\Gamma_1[x \mapsto \Gamma_1(y)], \Delta_1, \Theta_1)$.

If $c \equiv x.f{:=}y$ then $\langle \rho_2, h_2, A_2 \rangle = \langle \rho_1, h_1[(\rho_1(x), f) \mapsto \rho_1(y)], A_1 \rangle$. Two cases arise, depending on whether $n = \Gamma(x)$ is a strong or a weak node. If $c \equiv x.f{:=}y$ and $n = \Gamma(x) \in \Theta$ the node n represents a unique concrete cell in h. To check the first premise, we make use of Lemma 3.8 on a given path π, a node n and a location l such that $(\Gamma_2, \Delta_2) \vdash \pi \Downarrow n$ and $\langle \rho_2, h_2 \rangle \vdash \pi \Downarrow l$. This yields one of two main hypotheses. In the first case π is not modified by the f-redirection (formally, $\langle \rho_1, h_1 \rangle \vdash \pi \Downarrow l \wedge (\Gamma_1, \Delta_1) \vdash \pi \Downarrow n$), and by assumption $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash l \sim n$. Now consider any path π_0 such that $\langle \rho_2, h_2 \rangle \vdash \pi_0 \Downarrow l$: there is a node n_0 such that $(\rho_2, h_2) \vdash \pi_0 \Downarrow n_0$, and we can reapply Lemma 3.8 to find that $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash l \sim n_0$. Hence $n_0 = n$, and since nodes in Δ_1 and Δ_2 are untouched by the field assignment typing rule, we can conclude that the first premise is satisfied. In the second case (π is modified by the f-redirection) there is a path π' in $(\rho_1, h_1, \Gamma_1, \Delta_1)$ such that $y.\pi'$ leads respectively to l and n (formally $\langle \rho_1, h_1 \rangle \vdash y.\pi' \Downarrow l \wedge (\Gamma_1, \Delta_1) \vdash y.\pi' \Downarrow n$). As in the previous case, for any path π_0 such that $\langle \rho_2, h_2 \rangle \vdash \pi_0 \Downarrow l$ and $(\rho_2, h_2) \vdash \pi_0 \Downarrow n_0$, we have $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash l \sim n_0$, which implies that $n = n_0$, and thus we can conclude that π_0 leads to the same node as π, as required.

For the second premise, let $n \in \Theta_2$ and assume that n can be reached by two paths π and π' in Δ_2. The application of Lemma 3.8 to both of these paths yields the following combination of cases:

neither path is modified by the f-redirection: formally, $\langle \rho_1, h_1 \rangle \vdash \pi \Downarrow l \wedge (\Gamma_1, \Delta_1) \vdash \pi \Downarrow n \wedge \langle \rho_1, h_1 \rangle \vdash \pi' \Downarrow l' \wedge (\Gamma_1, \Delta_1) \vdash \pi' \Downarrow n$. By assumption, $l = l'$.

one of the paths is modified by the f-redirection: without loss of generality, assume $\langle \rho_1, h_1 \rangle \vdash \pi' \Downarrow l' \wedge (\Gamma_1, \Delta_1) \vdash \pi \Downarrow n$, and there is a path $\pi_\star$ in $(\rho_1, h_1, \Gamma_1, \Delta_1)$ such that $y.\pi_\star$ leads to l in the heap, and n in the graph (formally, $\langle \rho_1, h_1 \rangle \vdash y.\pi_\star \Downarrow l \wedge (\Gamma_1, \Delta_1) \vdash y.\pi_\star \Downarrow n$). By assumption, $l = l'$.

both paths are modified by the f-redirection: we can find two paths $\pi_\star$ and $\pi'_\star$ such that $y.\pi_\star$ leads to l in the heap and n in the graph, and $y.\pi'_\star$ leads to l' in the heap and n in the graph (formally, $\langle \rho_1, h_1 \rangle \vdash y.\pi_\star \Downarrow l \wedge (\Gamma_1, \Delta_1) \vdash y.\pi_\star \Downarrow n \wedge \langle \rho_1, h_1 \rangle \vdash y.\pi'_\star \Downarrow l' \wedge (\Gamma_1, \Delta_1) \vdash y.\pi'_\star \Downarrow n$). By assumption, $l = l'$.

In all combinations, $l = l'$ in h_1. Since the rule for field assignment in the operational semantics preserves the locations, then $l = l'$ in h_2.

This concludes the proof that $\langle \rho_1, h_1[(\rho_1(x), f) \mapsto \rho_1(y)], A_1 \rangle \sim (\Gamma_1, \Delta_1[n, f \mapsto \Gamma(y)], \Theta_1)$ when $n \in \Theta$.

If $c \equiv x.f{:=}y$ and $n = \Gamma(x) \notin \Theta$ here n may represent multiple concrete cells in h. Let σ_1 and σ_2 be the mappings defined, respectively, by the hypothesis $(\Gamma_1, \Delta_1, \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)$ and $(\Gamma_1, \Delta_1[n, f \mapsto \Gamma_1(y)], \Theta_1) \sqsubseteq (\Gamma_2, \Delta_2, \Theta_2)$. The first premise of the proof is proved by examining a fixed path π in $(\rho_2, h_2, \Gamma_2, \Delta_2)$ that ends in l_0 in the concrete heap, and n'_0 in the abstract graph. Applying Lemma 3.7 to this path (formally, instantiating l by $\rho_1(x)$, l' by $\rho_1(y)$, l_f by l_0, π by y, and π' by π) yields two possibilities. The *first alternative* is when π is not modified by the f-redirection (formally, $\langle \rho_1, h_1 \rangle \vdash \pi \Downarrow l_0$). Lemma 3.3 then asserts the existence of a node n_0 that π evaluates to in $(\rho_1, h_1, \Gamma_1, \Delta_1)$ (formally, $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow n_0$ with $n'_0 = \sigma_1(n_0)$). Moreover, by assumption n_0 and l_0 are in correspondence (formally, $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash l_0 \sim n_0$). To prove that l_0 and n'_0 are in correspondence, we refer to the auxiliary type interpretation rule in Fig. 5, and prove that given a path π_0 that verifies $\langle \rho_2, h_2 \rangle \vdash \pi_0 \Downarrow l_0$, the proposition $(\Gamma_2, \Delta_2) \vdash \pi_0 \Downarrow n'_0$ holds. Using Lemma 3.7 on π_0 (formally, instantiating l by $\rho_1(x)$, l' by $\rho_1(y)$, l_f by l_0, π by y, and π' by π_0), the only non-immediate case is when π_0 goes through f. In this case, $\pi_0 = \pi_z.f.\pi_1.f.\ldots.f.\pi_n.f.\pi_f$, and we can reconstruct this as a path to n'_0 in (Γ_2, Δ_2) by assuming there are two nodes n'_x and n'_y such that $\sigma_1(\Gamma_1(x)) = n'_x$ and $\sigma_1(\Gamma_1(y)) = n'_y$, and observing:

- $\langle \rho_1, h_1 \rangle \vdash \pi_z \Downarrow \rho_1(x)$ thus $(\Gamma_1, \Delta_1) \vdash \pi_z \Downarrow \Gamma_1(x)$ by assumption. Because access path evaluation is monotonic wrt. mappings (a direct consequence of clause (ST_2) in the definition of sub-typing), we can derive $(\Gamma_2, \Delta_2) \vdash \pi_z \Downarrow n'_x$;
- for $i \in [1, n]$, $\langle \rho_1, h_1 \rangle \vdash y.\pi_i \Downarrow \rho_1(x)$ thus by assumption $(\Gamma_1, \Delta_1) \vdash y.\pi_i \Downarrow \Gamma_1(x)$. By again using the monotony of access path evaluation, we can derive $(\Gamma_2, \Delta_2) \vdash y.\pi_i \Downarrow n'_x$;
- $\langle \rho_1, h_1 \rangle \vdash y.\pi_f \Downarrow l_0$ thus by assumption $(\Gamma_1, \Delta_1) \vdash y.\pi_f \Downarrow n_0$. Hence $(\Gamma_2, \Delta_2) \vdash y.\pi_f \Downarrow n'_0$ by $n'_0 = \sigma_1(n_0)$ and due to monotonicity.
- in $\Delta_1[n, f \mapsto \Gamma_1(y)]$, $n = \Gamma_1(x)$ points to $\Gamma_1(y)$ by f. Note that because the types $\Gamma_1, \Delta_1, \Theta_1$ and $\Gamma_1, \Delta_1[n, f \mapsto \Gamma_1(y)], \Theta_1$ share the same environment Γ_1, we have $\sigma_2(\Gamma_1(x)) = \sigma_1(\Gamma_1(x)) = n'_x$ and $\sigma_2(\Gamma_1(y)) = \sigma_1(\Gamma_1(y)) = n'_y$. Hence in Δ_2, n'_x points to n'_y by f, thanks to the monotonicity of σ_2.

This concludes the proof of $(\Gamma_2, \Delta_2) \vdash \pi_0 \Downarrow n'_0$. The cases when n'_x and n'_y do not exist are easily dismissed; we refer the reader to the Coq development for more details.

We now go back to our first application of Lemma 3.7 and tackle the *second alternative* – when π is indeed modified by the f-redirection. Here π can be decomposed into $\pi_z.f.\pi_1.f.\ldots.f.\pi_n.f.\pi_f$. We first find the node in Δ_1 that corresponds to the location l_0: for this we need to find a path in $(\rho_1, h_1, \Gamma_2, \Delta_2)$ that leads to both l_0 and n'_0 (in order to apply Lemma 3.3, we can no longer assume, as in the first alternative, that π leads to l_0 in $\langle \rho_1, h_1 \rangle$). Since $\langle \rho_1, h_1 \rangle \vdash y.\pi_f \Downarrow l_0$, the path $y.\pi_f$ might be a good candidate: we prove that it points to n'_0 in Δ_2. Assume the existence of two nodes n'_x and n'_y in Δ_2 such that $n'_x = \sigma_1(\Gamma_1(x)) = \sigma_2(\Gamma_1(x))$ and $n'_y = \sigma_1(\Gamma_1(y)) = \sigma_2(\Gamma_1(y))$ (the occurrences of non-existence are dismissed by reducing them, respectively, to the contradictory case when $\Gamma_2(x) = \top$, and to the trivial case when $n'_0 = \top$; the equality between results of σ_1 and σ_2 stems from the same reasons as in bullet 4 previously). From the first part of the decomposition of π one can derive, by assumption, that π_z leads to $\Gamma_1(x)$ in Δ_1, and, by monotonicity, to n'_x in Δ_2 (formally, $\langle \rho_1, h_1 \rangle \vdash \pi_z \Downarrow \rho_1(x)$ entails $(\Gamma_1, \Delta_1) \vdash \pi_z \Downarrow \Gamma_1(x)$ implies $(\Gamma_2, \Delta_2) \vdash \pi_z \Downarrow n'_x$). Similarly, we can derive that for any $i \in [1, n]$, $(\Gamma_1, \Delta_1) \vdash y.\pi_i \Downarrow \Gamma_1(x)$. We use monotonicity both to infer that in Δ_2, n'_x points to n'_y by f, and that for any $i \in [1, n]$ $(\Gamma_2, \Delta_2) \vdash y.\pi_i \Downarrow n'_x$. From these three statements on Δ_2, we have $(\Gamma_2, \Delta_2) \vdash \pi_z.f.\pi_1.f.\ldots.f.\pi_n \Downarrow n'_x$, and by path decomposition $(\Gamma_2, \Delta_2) \vdash y.\pi_f \Downarrow n'_0$. This allows us to proceed and apply Lemma 3.3, asserting the existence of a node n_0 in $(\rho_1, h_1, \Delta_1, \Gamma_1)$ that $y.\pi_f$ evaluates to (formally, $(\Gamma_1, \Delta_1) \vdash y.\pi_f \Downarrow n_0$ with $n'_0 = \sigma_1(n_0)$). Moreover, by assumption n_0 and l_0 are in correspondence (formally, $\langle \rho_1, h_1, A_1 \rangle, (\Gamma_1, \Delta_1) \Vdash l_0 \sim n_0$). The proof schema from here on is quite similar to what was done for the first alternative. As previously, we demonstrate that l_0 and n'_0 are in correspondence by taking a path π_0 such that $\langle \rho_2, h_2 \rangle \vdash \pi_0 \Downarrow l_0$ and proving $(\Gamma_2, \Delta_2) \vdash \pi_0 \Downarrow n'_0$. As previously, using Lemma 3.7 on π_0 (formally, instantiating l by $\rho_1(x)$, l' by $\rho_1(y)$, l_f by l_0, π by y, and π' by π_0), the only non-immediate case is when π_0 goes through f. In this case, $\pi_0 = \pi'_z.f.\pi'_1.f.\ldots.f.\pi'_n.f.\pi'_f$, and we can reconstruct this as a path to n'_0 in (Γ_2, Δ_2) by observing:

- $\langle \rho_1, h_1 \rangle \vdash \pi'_z \Downarrow \rho_1(x)$ thus $(\Gamma_1, \Delta_1) \vdash \pi'_z \Downarrow \Gamma_1(x)$ by assumption. Monotonicity yields $(\Gamma_2, \Delta_2) \vdash \pi'_z \Downarrow n'_x$;
- for $i \in [1, n]$, $\langle \rho_1, h_1 \rangle \vdash y.\pi'_i \Downarrow \rho_1(x)$ thus by assumption $(\Gamma_1, \Delta_1) \vdash y.\pi'_i \Downarrow \Gamma_1(x)$, and by monotonicity we can derive $(\Gamma_2, \Delta_2) \vdash y.\pi'_i \Downarrow n'_x$;
- $\langle \rho_1, h_1 \rangle \vdash y.\pi_f \Downarrow l_0$ thus since l_0 and n_0 are in correspondence, $(\Gamma_1, \Delta_1) \vdash y.\pi_f \Downarrow n_0$. Hence $(\Gamma_2, \Delta_2) \vdash y.\pi_f \Downarrow n'_0$ by $n'_0 = \sigma_1(n_0)$ and by monotonicity.

This concludes the proof of $(\Gamma_2, \Delta_2) \vdash \pi_0 \Downarrow n'_0$, hence the demonstration of the first premise.

We are now left with the second premise, stating the unicity of strong node representation. Assume $n \in \Theta_2$ can be reached by two paths π and π' in $\rho_2, h_2, \Gamma_2, \Delta_2$, we use Lemma 3.7 to decompose both paths. The following cases arise:

neither path is modified by the f-redirection: formally, $\langle \rho_1, h_1 \rangle \vdash \pi \Downarrow l \wedge \langle \rho_1, h_1 \rangle \vdash \pi' \Downarrow l'$. Lemma 3.3 ensures the existence of n' such that $n = \sigma_1(n')$ and $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow n' \wedge (\Gamma_1, \Delta_1) \vdash \pi' \Downarrow n'$ Thus by assumption, $l = l'$.

one of the paths is modified by the f-redirection: without loss of generality, assume $\langle \rho_1, h_1 \rangle \vdash \pi \Downarrow l$, and $\pi' = \pi_0.f.\pi_1.f.\ldots.f.\pi_n.f.\pi_f$ with $\langle \rho_1, h_1 \rangle \vdash y.\pi_f \Downarrow l'$. From the first concrete path expression, Lemma 3.3 ensures that there is n' such that $n = \sigma_1(n')$, and $(\Gamma_1, \Delta_1) \vdash \pi \Downarrow n'$. Moreover, by assumption there exists a strong node n'' in Δ_1 that $y.\pi_f$ leads to, and that is mapped to n by σ_1 (formally, $n = \sigma_1(n'')$ and $(\Gamma_1, \Delta_1) \vdash y.\pi_f \Downarrow n''$). Clause ($\mathrm{ST}_3$) of the definition of subtyping

states the uniqueness of strong source nodes: thus $n' = n''$, and we conclude by assumption that $l = l'$.

both paths are modified by the f-redirection: Lemma 3.7 provides two paths π_f and π'_f such that $y.\pi_f$ leads to l, and $y.\pi'_f$ leads to l' in the heap (formally, $\langle \rho_1, h_1 \rangle \vdash y.\pi_\star \Downarrow l \wedge \langle \rho_1, h_1 \rangle \vdash y.\pi'_\star \Downarrow l'$). As in the previous case, we use the assumption and clause (ST_3) of main sub-typing definition to exhibit the common node n' in Δ_1 that is pointed to by both paths. By assumption, $l = l'$.

This concludes the demonstration of the second premise, and of the $c \equiv x.f{:=}y$ case in our global induction. □

Theorem 3.10 (Type soundness)**.** *If $\vdash p$ then all methods m declared in the program p are secure,* i.e.*, respect their copy policy.*

Proof. [See Coq proof `Soundness.soundness` [1]]

Given a method m and a copy signature $\mathsf{Copy}(X)\{\tau\}$ attached to it, we show that 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 $x, y \in \mathit{Var}$,

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

Following the rule defining the semantics of calls to copy methods, we consider the situation where there exists a potential overriding of m, $\mathsf{Copy}(X')\ m(a){:=}c$ such that

$$(c, \langle \rho_\diamond[a \mapsto \rho_1(y)], h_1, \emptyset \rangle) \rightsquigarrow \langle \rho', h_2, A' \rangle.$$

Writing τ' for the policy attached to X', we know that $\tau \subseteq \tau'$. By Lemma 2.4, it is then sufficient to prove that $\rho_2, h_2, x \models \tau'$ holds.

By typability of $\mathsf{Copy}(X')\ m(a){:=}c$, there exist $\Gamma', \Gamma, \Delta, \Theta$ such that $\Gamma'(\mathit{ret}) = n_\tau$, $(\Gamma, \Delta, \Theta) \sqsubseteq (\Gamma', \Delta_\tau, \{n_\tau\})$ and $[\ \cdot \mapsto \bot][x \mapsto \top_{\mathit{out}}], \emptyset, \emptyset \vdash c : \Gamma, \Delta, \Theta$.

Using Theorem 3.9, we know that $\langle \rho', h_2, A' \rangle \sim (\Gamma, \Delta, \Theta)$ holds and by subtyping between (Γ, Δ, Θ) and $(\Gamma', \Delta_\tau, \{n_\tau\})$ we obtain that $\langle \rho', h_2, A' \rangle \sim (\Gamma', \Delta_\tau, \{n_\tau\})$ holds. Theorem 3.2 then immediately yields that $\rho_2, h_2, x \models \tau$. □

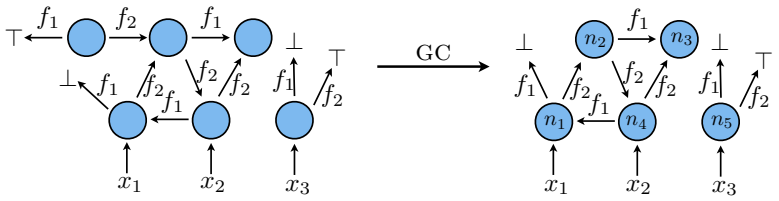
4. Inference

In order to type-check a method with the previous type system, it is necessary to infer intermediate types at each loop header, conditional junction points and weak field assignment point. 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.

Lemma 4.1. *The binary relation $\sqsubseteq$ is a preorder on $\boldsymbol{T}$.*

Proof. The relation is reflexive because for all type $T \in \boldsymbol{T}$, $T \sqsubseteq_{id} T$. The relation is transitive because if there exists types $T_1, T_2, T_3 \in \boldsymbol{T}$ such that $T_1 \sqsubseteq_{\sigma_1} T_2$ and $T_2 \sqsubseteq_{\sigma_2} T_3$ for some fusion maps σ_1 and σ_2 then $T_1 \sqsubseteq_{\sigma_3} T_3$ for the fusion map σ_3 define by $\sigma_3(n) = \sigma_2(\sigma_1(n))$ if $\sigma_1(n) \in \boldsymbol{N}$ or $\top$ otherwise. □

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 $(\boldsymbol{T}, \equiv)$, 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 write n_i (resp. x_i and f_i) for the ith node (resp. variable and field). A type (Γ, Δ, Θ) is *well-formed* if every node in Δ is reachable from a node in Γ and the nodes in Δ 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 call this transformation GC. 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 that an 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 whether a fixpoint iteration has ended.

Lemma 4.2. *For all well-formed types $T_1, T_2 \in \boldsymbol{T}$, $T_1 \equiv T_2$ iff $T_1 = T_2$.*

Definition 4.3. Let $\sqcup$ be an operator that merges two types according to the algorithm in Fig. 10.

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 Δ_1 and Δ_2 into a non-deterministic graph (NDG) α, 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 α referenced by Γ_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 Δ from the now-*deterministic* graph α. This function operates by pushing a node set to a node labelled by the $\leq_\sigma$-sup of the set. The result environment Γ is derived from Γ_i and α before the Δ-reconstruction.

All state fusions are recorded in a map σ which binds nodes in $\Delta_1 \cup \Delta_2$ to nodes in Δ. Figure 11 contains the auxiliary functions used in the above procedure. Figure 12 unfolds the algorithm on a small example.

Theorem 4.4. *The operator $\sqcup$ defines a sup-semi-lattice on types.*

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

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

// Determinize α–transitions:
//    start with the entry points.
for {(x,t);(x,t')} ⊆ (Γ1 × Γ2) {
  fusion({t,t'})
}

// Determinize α–transitions:
//    propagate inside the graph.
while ∃u ∈ α, ∃f ∈ Field, |succ(u,f)| > 1 {
  fusion(succ(u,f))
}

// α is now fully determinized:
//    convert it back into a type.
(Γ, Δ, Θ) = ground(Γ1, Γ2, α)
```

```
// S is a set of t ∈ t.
// (|S|) denotes a node labelled by S.
void fusion (S) {
  // Create a new α–node.
  NDG node n = (|S|)
  α ← α + n
  // Recreate all edges from the fused
  //    nodes on the new node.
  for t ∈ S {
    for f ∈ Field {
      if ∃u, α(t,f) = u {
        // Outbound edges.
        α ← α with (n,f) ↦ u
      }
      if ∃n', α(n',f) = t {
        // Inbound edges.
        α ← α with (n',f) ↦ n
    }}
  // Delete the fused node.
  α ← α − t
}}
```

FIGURE 10. Join Algorithm

```
// Convert a type to an NDG
NDG lift (Γ1, Γ2, Δ) {
  NDG α = undef
  for x ∈ Var {
    α ← α + (|{Γ1(x)}|) + (|{Γ2(x)}|)
  }
  for n ∈ Δ {
    if ∃f ∈ Fields, ∃b ∈ BaseType, Δ[n,f] = b {
      α ← α + (|{n}|) + (|{b}|)
      α ← α with ((|{n}|), f) ↦ (|{b}|)
  }}
  return α
}
```

```
// Convert an NDG to a type
T ground (Γ1, Γ2, α) {
  (Var → t) Γ = λx.⊥
  Δ Δ = undef
  for N ∈ α {
    for x ∈ Var, Γ1(x) ∈ N ∨ Γ2(x) ∈ N {
      Γ ← Γ with x ↦ N↓
  }}
  for N, N' ∈ α {
    if α(N,f) = N' {
      Δ ← Δ with (N↓, f) ↦ N'↓
  }}
  return (Γ, Δ)
}
```

```
// ≤σ–sup function
BaseType ↓ (N) {
  if ⊤ ∈ N return ⊤
  else if ∀c ∈ N, c = ⊥ return ⊥
  else return freshNode()
}
```

FIGURE 11. Auxiliary join functions

Proof. First note that σ_1 and σ_2, the functions associated with the two respective sub-typing relations, can easily be reconstructed from σ.

Upper bound: Let $(T, \sigma) = T_1 \sqcup T_2$: we prove that $T_1 \sqsubseteq T$ and $T_2 \sqsubseteq T$. Hypothesis (ST_2) is discharged by case analysis, on t_1 and on t_2. The general argument

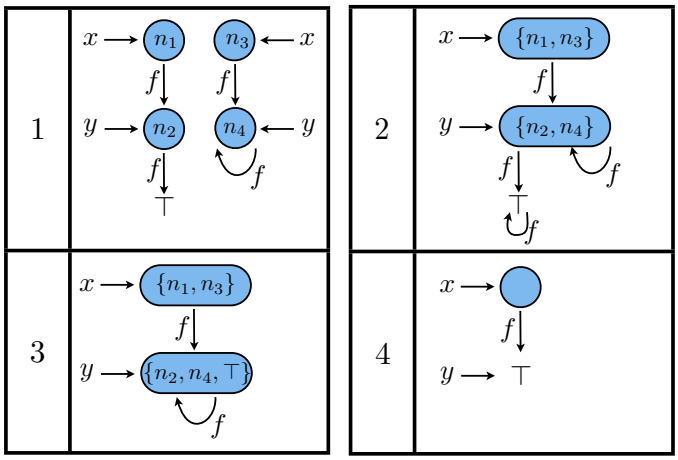


FIGURE 12. An example of join

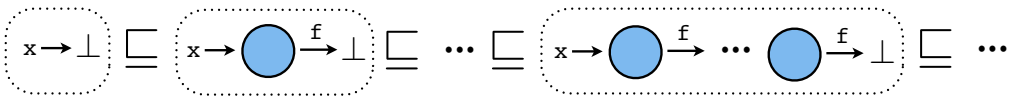
used is that the join algorithm does not delete any edges, thus preserving all paths in the initial graphs.

Least of the upper bounds: Let $(T, \sigma) = T_1 \sqcup T_2$. Assume there exists T' such that $T_1 \sqsubseteq T'$ and $T_2 \sqsubseteq T'$. Then we prove that $T \sqsubseteq T'$. The proof consists in checking that the join algorithm produces, at each step, an intermediary pseudo-type T such that $T \sqsubseteq T'$. The concrete nature of the algorithm drives the following, more detailed decomposition.

(1) Given a function σ, define a state mapping function $\leq_\tau$, and a $\sqsubseteq_\sigma$-like relation $\sqsubseteq_\tau$ on non-deterministic graphs. The aim with this relation is to emulate the properties of $\sqsubseteq_\sigma$, lifting the partial order $\leq_\sigma$ on nodes to sets of nodes (cells). Lift T' into an NDG α'.
(2) Using the subtyping relations between T_1, T_2, and T', establish that the disjunct union and join steps produce an intermediary NDG β such that $\beta \sqsubseteq_\tau \alpha'$.
(3) Ensure that the fusions operated by the join algorithm in β produce an NDG γ such that $\gamma \sqsubseteq_\tau \alpha'$. This is done by case analysis, and depending on whether the fusion takes place during the first entry point processing phase, or if it occurs later.
(4) Using the $\boldsymbol{t}$-lattice, show that the ground operation on the NDG γ produces a type T that is a sub-type of T'.

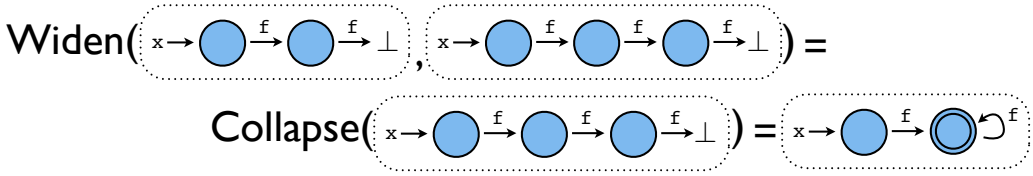
□

The poset structure does have infinite ascending chains, as shown by the following example.



Fixpoint iterations may potentially result in such an infinite chain so we have then to rely on a widening [9] operator to enforce termination of fixpoint computations. Here we follow a 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. This is illustrated by the following example.



This ensures the termination of the fixpoint iteration because the number of nodes is then bounded by $2N$ with N the number of local variables in the program.

5. Experiments

The policy language and its enforcement mechanism have been implemented in the form of a security tool for Java bytecode. Standard Java `@interface` declarations are used to specify native annotations, which enable development environments such as Eclipse 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, and to facilitate the creation of an experimental Eclipse plugin.

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. 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 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. As expected, some cloning methods are 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.

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 was used to test the tool on two large code bases. The 17000 classes in Sun's `rt.jar` and the 7000 others 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 assignment or method call on non-local memory, as explained in Section 3.4. Assignment on non-local memory means here that the copying method is updating fields of other objects than the result of

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

the copy itself. For such examples, our shape analysis seems too coarse to track the dependencies between sources and copy targets. For 78 methods we were unable to infer the minimal, shallow signature {} (the same signature as `java.lang.Object.clone()`). In some cases, for instance in the `DomAttr` class, this will happen when the copy method returns the result of another, unannotated method call, and can be mitigated with additional copy annotations. In other cases, merges between abstract values result in precision losses: this is, for instance, the case for the `clone` method of the TreeMap class, as explained above.

Our prototype confirms the efficiency of the enforcement technique: these verifications took about 25s to run on stock hardware. The 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 [5]. 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 [11] for which a precise semantic characterization has emerged [12]. 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.* [3] 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 [2]. 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 [4, 7] 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 [18], separation logic [15] and shape analysis [16]. 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`.

Noble *et al.* [14] propose a prescriptive technique for characterizing the aliasing, and more generally, the topology of the object heap in object-oriented programs. This technique is based on alias modes which have evolved into the notion of ownership types [8]. In this setting, the annotation `@Repr` is used to specify that an object is *owned* by a specific object. It is called a *representation* of its owner. After such a declaration, the programmer must manipulate the representation in order to ensure that any access path to this object should pass trough its owner. Such a property ensures that a `@Repr` field must be a `@Deep` field in any copying method. Still, a `@Deep` field is not necessarily a `@Repr` field since a copying method may want to deeply clone this field without further interest in the global alias around it. Cloning seems not to have been studied further in the ownership community and ownership type checking is generally not adapted to flow-sensitive verification, as required by the programming pattern exhibited in existing code. In this example, if we annotate the field `next` and `previous` with `@Repr`, the `clone` local variable will not be able to keep the same ownership type at line 12 and at line 26. Such a an example would require ownership type systems to track the update of a reference in order to catch that any path to a representation has been erased in the final result of the method.

We have aimed at annotations that together with static analysis allows to verify existing cloning methods. Complementary to our approach, Drossopoulou and Noble [10] propose a system that generate cloning methods from annotation inpired by ownership types.

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 deals with dynamic method dispatch and addresses some of the problems posed by redefinition of cloning methods in inheritance-based object oriented programming language (but see Section 2.3 for a discussion of current limitations). The verification technique is designed to enable modular verification of individual classes. 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. This means that our method can form part of an extended, security-enhancing Java byte code verifier which of course would have to address, in addition to secure cloning, a wealth of other security policies and security guidelines as *e.g.*, listed on the CERT web site for secure Java programming [6].

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 was particularly challenging because of the storeless nature of our type interpretation but in the end proved to be of great help to get the soundness arguments right.

Several issues merit further investigations in order to develop a full-fledged software security verification tool. The extension of the policy language to be able to impose policies on fields defined in sub-classes should be developed (*cf.* discussion in Section 2.3). We believe that the analysis defined in this article can be used to enforce such policies but their precise semantics remains to be defined. 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 conjecture that the extension to interleaving multi-threading semantics is feasible and that it can be done without making major changes to the type system because we only manipulate thread-local pointers.

An other line of work could be to consider the correctness of `equals`() methods with respect to copying methods, since we generally expect `x.clone().equals(x)` to be `true`. The annotation system is already in good shape for such a work but a static enforcement may require a major improvement of our specifically tailored shape analysis.

8. Acknowledgement

We wish to thank the ESOP11 and LMCS anonymous reviewers for their helpful comments on this article. We specially thank the anonymous reviewer who suggested the `EvilList` example presented in Section 2.3.

References

[1] Secure cloning webpage. `http://www.irisa.fr/celtique/ext/clones`, September 2011.

[2] 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.

[3] 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.

[4] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proc. of OOPSLA*, pages 20–34. ACM Press, 1999.

[5] J. Bloch. *JSR 175: A metadata facility for the Java programming language*. `http://jcp.org/en/jsr/detail?id=175`, September 30, 2004.

[6] CERT. *The CERT Sun Microsystems Secure Coding Standard for Java*, 2010. `https://www.securecoding.cert.org`.

[7] 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.

[8] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[9] 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.
[10] S. Drossopoulou and J. Noble. Trust the Clones. In *FoVEOOS 2011 - preproceedings*, September 2011.
[11] 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.
[12] 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.
[13] T. Jensen, F. Kirchner, and D. Pichardie. Secure the clones: Static enforcement of secure object copying. In G. Barthe, editor, *Proc. of 20th European Symposium on Programming (ESOP 2011)*. Springer LNCS vol. 6602, 2011.
[14] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In *Proc. of ECOOP'98*, pages 158–185. Springer LNCS vol. 1445, 1998.
[15] 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.
[16] 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.
[17] Sun Developer Network. *Secure Coding Guidelines for the Java Programming Language, version 3.0*, 2010. `http://java.sun.com/security/seccodeguide.html`.
[18] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.